Programação Concorrente 2019/1: Simulação de uma Maratona de Programação

Gabriel N. R. Fonseca - 16/0006597
Departamento de Ciência da Computação
Universidade de Brasília - UnB
Brasília, Brasil
nunesgrf@gmail.com

Este relatório trata do estudo de uma proposta de um problema envolvendo concorrência por recursos computacionais com a temática de Maratonas de Programação e programação competitiva, envolvendo uma análise sobre seu caráter educacional e sua relevância, bem como a descrição de um algoritmo para a solução do mesmo.

***Index Terms*—concorrência, maratona, programação, algoritmo, problema**

I. Introdução

Programação competitiva é um esporte onde times de entusiastas por programação e por algoritmos são postos diante de uma coleção de problemas e vence o time que melhor pontuar ao final da prova, sendo esta pontuação determinada pelo edital divulgado pelo organizador. A exemplo disso, temos as Maratonas de Programação organizadas aos moldes do ICPC, onde vence o time que mais resolver questões em um período de cinco horas corridas sendo o critério de desempate o tempo acumulado em que as soluções corretas são apresentadas diante do corretor automático.

Cada time de uma maratona aos moldes do ICPC [3] é composto por três indivíduos e uma única máquina o qual o time deve utilizar para implementar o código-fonte de cada problema, ou seja, o desafio não limita-se a resolver o caderno de questões incluindo também como o time vai lidar com o uso da maquina sem que haja perda de tempo. Alguns times adotam a estratégia de formar uma fila, outros preferem gastar os minutos iniciais para ordenar o caderno de questões em nível de dificuldade e outros simplesmente decidem caso a caso quem será o programador que irá sentar e utilizar a máquina.

Quanto ao corretor automático, este é único e roda em um servidor local para onde todos os times enviam suas submissões e recebem de retorno um veredito AC (*Accepted*) para submissões corretas, WA (*Wrong Answer*) para incorretas, RTE (*Run Time Error*) para submissões com erro de execução, TLE (*Time limit error*) para casos em que o tempo limite de execução é estourado e CE (*Compilation Error*) que é quando o programa sequer compila.

Visto é natural pensar que com ajustes, simular uma Maratona de Programação pode ser um bom exemplo para a prática do paradigma concorrente de programação e justamente isto será discutido dentro deste relatório, bem como as limitação e as dificuldades de tradução em liguagem de programação.

Na seção Formalização do Problema Proposto será discutido o problema em si, uma espécie de destrinchamento e redução à problemas clássicos, serão mostradas as limitações e degenerações bem como as decisões arbitrárias necessárias para tornar o problema menos complexo e mais didático. Na seção Algoritmo será apresentadas as estruturas da biblioteca POSIX Threads utilizadas para a resolução do problema bem como a razão de utilizar cada uma delas. A conclusão tratará sobre o que foi identificado de positivo e problemático na implementação proposta bem como eventuais evoluções.

II. Formalização do Problema Proposto

Nas 3 subsessões seguintes serão descritos todas as modificações em relação ao problema original para fim de torná-lo mais didático e focado nos conceitos de concorrência.

A. O problema

A UnC - Universidade da Competitiva modelou uma novo edital para programação competitiva inspirado no ICPC. Neste modelo, N times de K pessoas são postos diante de um conjunto infinito de problemas com dificuldade variadas e ganha o time que resolver M problemas de qualquer dificuldade primeiro; Cada time tem a sua disposição um computador e um despachador de código-fonte sendo possível a situação onde um estudante escreve código ao mesmo tempo que outro envia.

Além disso, existe um corretor automatizado que só funciona se há submissões à serem julgadas. Estudantes da UnC são muito preciosistas com o código no quesito complexidade e teste, logo, nunca recebem os vereditos RTE, CE e TLE, O único veredito negativo possível é o WA que acusa falha de lógica.

No que diz a cada estudante individualmente, cada um possui um nível de inteligência aleatório que é determinante para sabermos quanto tempo ele demora para preparar uma submissão (O processo de pensar e em seguida escrever o código) Quando alguém vence a competição, os organizadores anunciam a todos e então encerra-se a competição e submissões pendentes de julgamento são descartadas.

B. Degenerações

No problema descrito acima, foi determinado que o conjunto de questões seria infinito e com dificuldade aleatório pois não

entra no escopo de concorrência definir uma lista igual de questões para todos os N times, o número infinito de questões com dificuldade aleatória já é suficiente para que haja uma simulação aproximada do que vá ser a realidade.

Quanto encerrar-se baseado em questões resolvidas e não em tempo, é uma simples mudança para tornar mais dinâmica as simulações e tornar mais fácil aferir que esta deve ser encerrada.

Outro ponto é que os vereditos são limitados a AC e WA pois neste problema é irrelevante o qualitativo de cada submissão, somente se esta foi aceita ou não, futuramente, já na seção algoritmo, uma nova degeneração será mostrada onde uma submissão qualquer tem probabilidade 0.75 de ser aceita e 0.25 de ser recusada, somente para fins de simulação.

Cada time tem K integrantes que se revezam no computador, a estratégia de todos os times consiste em deixar quem chegar primeiro utilizar a máquina.

C. Destrinchamento

Dado o problema e as degenerações, trivialmente, podemos perceber que trata-se de uma variação do problema do produtor-consumidor junto com o problema dos leitores e escritores. Cada time produz constantemente submissões que são escritas em uma fila e consumidas pelo Juiz.

Internamente a cada time, há uma competição por qual estudante desempenhará o papel de produtor, sendo bem fácil de abstrair uma solução quanto pensado desta forma.

III. Algoritmo

Nesta seção serão descritos as estruturas usadas bem como as razões das mesmas estarem sendo posicionadas naquele trecho de código.

A. pthread_mutex_t

[1]
```
pthread_mutex_lock(&st→computer);
implementing(st);
pthread_mutex_unlock(&st→computer);

pthread_mutex_lock(&queue);
submitting(st);
pthread_mutex_unlock(&queue);
```

Está é a estrutura mutex é uma abreviação do inglês exclusão mútua, esta estrutura pode possuir dois estados *lock* quando pertence a uma thread e *unlock* quando não pertence a nenhuma thread. O mutex não pode pertencer a duas threads simultaneamente, sendo interessante para problemas de acesso à região crítica [1] pois quando uma thread tenta adquirir a permissão de um mutex quando outra thread já o obteve a primeira thread aguarda naquele trecho do código a liberação do mutex.

Neste algoritmo, o mutex foi utilizado com dois intuitos, primeiro, garantir que nenhuma thread estudante de um time qualquer acesse o computador ao mesmo tempo que outro estudante. O mesmo vale para o despachador de código-fonte, o objetivo foi respeitar as condições impostas pelo problema na origem, como pode ser observado código acima.

O outro ponto onde utilizou-se o mutex foi para gerenciar o acesso do Juiz e dos Times à fila de de submissões, em teoria, não seria necessário, já que os Juiz nunca está operante quando a fila está vazia e as submissões são feitas uma a uma por ordem de chegada. Porém, a estrutura de dados utilizada para implementar esta fila foi a *std::queue* da biblioteca padrão do C++, pela fato de ser dinâmica, toda vez que é realizado uma uma operação *pop()* nesta fila, todos os objetos tem o índice atualizado para uma posição anterior à atual [4], e como não se trata de uma operação atômica caso algum elemento seja inserido na fila no momento em que esta está sendo atualizada o programa encerra com *segmentation fault*

B. pthread_cond_t

pthread_cond_t ou simplesmente variável de condição trata-se um dispositivo de sincronização entre threads onde quando utilizada, a thread entre em estado de espera não-ocupada até que uma condição qualquer seja satisfeita em outra thread [2]. Quando a situação é satisfeita, a thread o qual ocorreu esse evento deve sinalizar o que torna esta operante novamente e concorrente aos recursos da máquina.

Neste código foi utilizado uma variável de condição para garantir que Juiz não tente julgar nada enquanto a fila estiver vazia e também para reforçar a ideia de proteção contra o *segmentation fault* evitando comparações desnecessárias no futuro. O uso pode ser observado no pseudo código abaixo.

[2]
```
if(submissions.size() == 0) {
    printf(MAG "NÃO HÁ SUBMISSÕES AGUARDANDO JULGAMENTO!\n" RESET);
    pthread_cond_wait(&solution, &cond_lock);
}
```

C. Função principal

A função principal neste algoritmo fica responsável somente pela inicialização das threads que representam os times, como o encerramento das mesmas utilizando o método join e ao final a exibição do resultado final da simulação.

D. Código das Threads Time

Cada time é responsável por gerenciar K estudantes, logo similar ao comportamento da função principal, o código executado pelas threads Time criam e encerram K estudantes. Diferentemente da função principal, este código aloca também uma struct de meta-dados e passa como argumento para cada uma das threads-filha contendo o identificar próprio, o identificador da filha, o nível de inteligência da thread-filha e uma referência constante a um mutex visível somente a ela e as filhas.

A razão de fazer o mutex ser criado no escopo da team_thread se dá pelo fato de somente estudantes de um mesmo time concorrerem pelo acesso ao computador, não tendo nada haver com os outros N-1 times.

E. Código das Threads Estudantes

As threads estudante é onde ocorre de fato a lógica de concorrência por um recurso, esta thread é executada até que exista um vencedor, mais do comportamento da mesma será descrito na análise dos casos das subseções

F. Execução do caso atômico

[3]0.2

```
NÃO HÁ SUBMISSÕES AGUARDANDO JULGAMENTO!
T0E0 está pensando em uma questão.
T0E0 teve uma ideia e quer o computador.
T0E0 está implementado e levará 1.00
T0E0 terminou de implementar
T0E0 está submetendo a solução ao juiz.
T0E0 está pensando em uma questão.
JUIZ ESTÁ LIGADO E PRONTO PARA JULGAR!
JUIZ ESTÁ JULGANDO SUBMISSÃO DO TIME 0!
AC PARA O TIME 0!
T0E0 teve uma ideia e quer o computador.
T0E0 está implementado e levará 0.67
T0E0 terminou de implementar
T0E0 está submetendo a solução ao juiz.
TIME 0 FOI VENCEDOR DA MARATONA DE PROGRAMAÇÃO!
```

Como forma de testar, é interessante observar a execução no caso de base, para isso, teremos 1 único time, com 1 único integrante e 1 único problema para resolver e ser declarado vitorioso.

É possível observar que mesmo após o veredito AC, o estudante faz outra submissão antes de ser declarado vencedor, isto acontece,pois adentrando a lógica do problema, é como se o julgamento da primeira submissão e a implementação de uma segunda solução fossem simultâneos.

G. Execução para um time de 3 integrantes

[4]0.2

```
NÃO HÁ SUBMISSÕES AGUARDANDO JULGAMENTO!
T0E0 está pensando em uma questão.
T0E0 teve uma ideia e quer o computador.
T0E2 está pensando em uma questão.
T0E1 está pensando em uma questão.
T0E0 está implementado e levará 1.50
T0E2 teve uma ideia e quer o computador.
T0E1 teve uma ideia e quer o computador.
T0E0 terminou de implementar
T0E0 está submetendo a solução ao juiz.
T0E2 está implementado e levará 1.00
T0E0 está pensando em uma questão.
JUIZ ESTÁ LIGADO E PRONTO PARA JULGAR!
T0E0 teve uma ideia e quer o computador.
JUIZ ESTÁ JULGANDO SUBMISSÃO DO TIME 0!
WA PARA O TIME 0!
NÃO HÁ SUBMISSÕES AGUARDANDO JULGAMENTO!
```

Semelhante ao caso atômico, o time tenta mais uma submissão mesmo em uma situação o qual já estaria vitorioso, da-se pelo mesmo motivo descrito anteriormente, ainda respeitando a lógica do problema.

H. Execução do caso geral

Executando o caso geral com vários times e vários estudantes por time, nota-se uma ocorrência de vitórias enviesada para alguns times, para o caso do *Ubuntu 18.04 LTS*, o time de id N-1 (O time com a ultima thread criada) geralmente é o vencedor, isso se dá pela forma como o Sistema Operacional lida com o escalonamento das threads e também o estado atual da maquina no momento exato dos testes. Em outro sistema operacionais é possível que haja enviesamento para outro time ou até mesmo não haver enviesamento.

IV. Conclusão

Apesar de ser um problema interessante para reforçar os conceitos básicos dos mecanismos de sincronização, é importante salientar que o mesmo apresenta resultados enviesados e necessita adaptações no modelo para funcionar. Eventualmente, a implementação da thread juiz na forma de um servidor e N máquinas representando os N times traga uma simulação fidedigna do que é uma Maratona de Programação.

Dado isto, o exercício pode ser interessante para demonstrar que a execução de threads se dá por tempo lógico e não por tempo físico.

Referências

[1] Documentação pthread_mutex_t. Acessado em 3 de junho de 2019 - http://sourceware.org/pthreads-win32/manual/pthread_mutex_init.html

[2] Documentação pthread_cond_t. Acessado em 3 de junho de 2019 - https://sourceware.org/pthreads-win32/manual/pthread_cond_init.html

[3] The ICPC International Collegiate Programming Contest - Acessado em 3 de junho de 2019 - https://icpc.baylor.edu/

[4] std::queue cpp reference - Acessado em 3 de junho de 2019 - https://en.cppreference.com/w/cpp/container/queue