Trabalho Prático – Apuração de Votos

Bárbara Luíza do Nascimento[1], Gustavo Bandeira da Silva[1]

[1]Departamento de Ciência da Computação – Universidade do Estado de Santa Catarina (UDESC) – Joinville, SC – Brasil

barbaraluizanascimento@gmail.com, guzaop@gmail.com

1. Introdução

O presente relatório tem o objetivo de descrever como foi desenvolvida uma aplicação que realiza a apuração de votos de uma eleição.

O trabalho foi desenvolvido no âmbito da disciplina de Sistemas Operacionais, portanto foram utilizadas as APIs para programação concorrente, estudadas em sala de aula, como *POSIX Pthreads* e para controle de concorrências, *Pthreads Mutexes*.

O código encontra-se implementado em C, no Sistema operacional Linux, e suas funcionalidades são, basicamente, processar um arquivo com votos, validá-los, colocá-los em uma fila e ainda contabilizá-los por *threads*.

2. Funções e funcionamento da aplicação

A biblioteca *Pthread*, fornece APIs para utilização das *threads* e implementa a *POSIX threads*. As estruturas e funções utilizadas desta biblioteca foram `pthread_t`, `pthread_create`, `pthread_join`, `pthread_exit`, entre outras. Portanto, segue explicação detalhada das demais funções do programa.

2.1. Função `processaVotos`

A função `processaVotos` é executada por cada uma das *threads*. As *threads* precisam contar todos os números de votos, válidos e inválidos e também inserir votos válidos na lista de votos.

Nessa função, temos um *if* que verifica se o arquivo é nulo. Quando o arquivo é aberto, logo após de ter concatenado com o nome e *tid*, se ele for nulo, ou seja, se ele não existir, o sistema aponta erro e finaliza a *thread* atual. Ao ser finalizada, seu vetor `t_inserindo` é alterado e retornamos a lista de totais de votos, sempre na saída das *threads*.

Os votos totais e votos inválidos contabilizados pela *thread*, são armazenados em um vetor chamado *totais*, que possui duas posições para guardar os valores de votos totais e inválidos. A primeira posição é `totais[0]`, que armazena o número total de votos que a *thread* contou e a segunda é `totais[1]`, que é a quantidade de votos inválidos que a *thread* processou.

Caso o arquivo exista, o programa lê linha por linha do arquivo e armazena na variável `line`, lê o valor guardado pela função `fgets` e armazena na variável temporária `votoString`. Tendo isso como *string*, garantimos que a aplicação trate os votos inválidos, porque a função `strtoul` retorna zero, converte uma *string* para um *int*, podendo assim tratar esses valores como voto inválido.

A função `strncmp` compara se o valor da variável linha, é zero, em forma de *string*. Se for igual, a função retorna zero. Se retornar um valor diferente de zero e voto for zero (já convertido pela função `strtoul`), então significa que na variável linha, realmente tem uma *string*.

Apenas os votos válidos são adicionados à lista de votos. Porém, antes de verificar se o voto é válido ou não, somamos todos os votos em *totais,* porque independente de ser válido ou não, a *thread* deve contabilizar o recebimento e leitura deste voto do arquivo.

3. Controle de Concorrências

Em *else* temos o início da região crítica das *threads*, que ao percorrerem a lista de votos, podem gerar inconsistências ou perda de votos, pois irão adicionar novos nós a lista de votos. Por isso, o mutex foi inicializado aqui, para que duas *threads* ou a `main` não percam o ponteiro da cabeça da lista de candidatos, mexendo nessa lista ao mesmo tempo. O mutex evita também que as *threads* continuem uma inserindo e outra removendo, por exemplo. Temos uma lista auxiliar para ir retirando nesta região crítica.

Ao entrar nesta condição, provavelmente o voto válido será encontrado e através do *for*, o sistema percorrerá a lista de `cabeçaLcand`. Esta lista é global e armazena a lista de candidatos, inicializada em `main`, pela função `inicializa_lista_candidatos`. Então, se o *id* que está nesta lista, for igual a algum voto, ele é adicionado na lista de votos, porque é válido, parando esta iteração e seguindo para o próximo nó. Se "próximo" for nulo e *current id*, que é o *id* do candidato atual, for diferente de voto, entramos na condição para os votos inválidos, porque não há candidatos com o mesmo *id* inserido na lista de votos.

No final de toda *thread*, atribuímos o valor da posição de `t_inserindo`, que é um vetor, alterando o valor da posição atual (número da *thread*), temos isso na linha `*(t_inserindo + (tid-1)) = 0`. São retornados os vetores *totais*, com o total de votos que a *thread* leu em *totais[0]* e votos inválidos em *totais[1]*. Este vetor é global, porque tanto a função `processaVotos` executado pelas *thread*, quanto a `main` trabalham com ele.

A `main` deve ler a lista de votos, tendo a *thread* inserido o voto ou não, por isso ela tem concorrência e é definida como a *thread* "consumidora", enquanto as *threads* que executam a função `processaVotos`, são as "produtoras". A lista de votos pode chegar a ser nula, mas a *thread* ainda pode estar inserindo votos, sendo assim, a `main` não pode parar.

No *for* dentro do *while*, temos a variável `soma_vetor_threads`, que recebe a soma de todos os valores das posições do vetor global `t_inserindo`, inicializado em zero, que ao final de cada *thread* é atribuído o valor 1. Quando esse valor for zero, todas as *threads* terão acabado de executar. Precisamos desse *while*, porque se todas as *threads* começassem com a `main`, elas inseririam na lista e não teríamos problema na `main`, poderíamos ler até a lista de votos ser nula e nunca iriam faltar voto processando a *thread*, porém isso não acontece, por causa da concorrência. Então, quando a soma for nula, nos certificamos de que não há nenhum voto sendo ainda processado, por isso utilizamos este *while*.

O primeiro *if* dentro do mutex, retorna a função `tamanho_lista_votos`, que conta o maior número de nós que a lista alcançou. Ele está em uma região crítica, porque antes de removermos algo, precisamos verificar o tamanho da lista. Então, iniciamos o mutex, para que as *threads* parem de adicionar na lista, entramos então no *while* para verificar o tamanho atual da lista. Se for menor, passa para a variável `maior_tamanho_lista`.

O próximo *for* percorre a lista de candidatos, comparando os *ids*, de acordo com a posição do nó auxiliar e atribui o voto ao determinado candidato. *Current* é o nó da lista de candidatos que ele está e o auxiliar é o nó da lista de votos.

Saindo da região crítica, damos `pthread_join`. Retornamos *ret*, que é um ponteiro do vetor *totais* que a *thread* retorna e contabilizamos a quantidade total de votos e de votos inválidos de cada *thread*.

Por último, temos os *prints* que exibem os resultados, utilizando `fmtcabvoto` e `fmtvoto` de `utils.h`, com o cabeçalho definido pelo professor.

4. Conclusão

Com a execução deste trabalho prático, percebemos que realmente as *threads* facilitam os eventuais problemas decorrentes na aplicação, porque faz com que as soluções possam executar em paralelo. Uma *thread* pode executar algo sozinha, enquanto as outras esperam para poder executar, quando uma é bloqueada, a outra já está esperando para executar.

O uso de *threads* também ajuda a diminuir o tempo de execução de um programa, porque ao inseri-las na aplicação, elas executam quase que em paralelo. Porém *threads* também se sobrepõem muitas vezes, fazendo com que o andamento da aplicação seja mais acelerado e mais propício a falhas, quando se trabalha com manipulação de estruturas de dados compartilhadas, sem realizar o devido tratamento das regiões críticas de concorrências.

5. Referências

Park, Alfred (1999-2018) “Multithreaded Programming (POSIX pthreads Tutorial)”, http://randu.org/tutorials/threads/

Lawrence Livermore National Laboratory (2017) “POSIX Threads Programming”, http://computing.llnl.gov/tutorials/pthreads/index.html

Mesquita, Renato Cardoso (2000) “Apostila do Curso de Linguagem C / UFMG”, http://moodle.joinville.udesc.br/pluginfile.php/2699/mod_resource/content/1/ApostLinguagemC_UFMG.pdf