Relatório do Trabalho de Sistemas Operacionais

Peterson Carvalho (GRR20163053)
pwkc16@inf.ufpr.br

Luiz Chiconato (GRR20166814)
lfac16@inf.ufpr.br

01 de Dezembro de 2017

1 Introdução

Esse trabalho tem como objetivo implementar uma hash multi-threaded criptografada com as operações de inserção e busca em C, usando a biblioteca OPENSSL e Pthreads. As requisições de busca e inserção são enviadas para o programa via Unix Sockets.

2 Estrutura de Dados

Na Tabela Hash, as eventuais colisões são resolvidas com árvores AVL para ter inserções e buscas em $\log_2 n$. A Hash é estruturada em um array de 1.000.000 de elementos sendo que cada elemento é um ponteiro para uma árvore. Cada nodo da árvore contém a chave, o nome criptografado, o telefone criptografado e os ponteiros para as sub-árvores.

Para que as operações sejam multi-threaded, há um mutex para cada entrada da tabela que as operações de get e put disputam o seu acesso caso haja colisão e acesso simultâneo.

3 Implementação

O programa principal se conecta com os sockets para obter as requisições, inicializa os semáforos, mutexes e estrutura de dados e cria 8 threads de inserção e 8 threads de busca e depois espera pelo término dessas threads.

3.1 Threads de Busca

Cada thread de busca tem seu próprio buffer de mensagens e variáveis locais. A cada mensagem processada, é verificado se o relógio lógico da mensagem de *get* for menor que o menor relógio lógicos de todos os *puts*, essa thread é bloqueada até que algum put a libere novamente com um *signal* no semáforo.

```
1  inicialização;
2  enche o buffer com mensagens do socket;
3  while not fim das requisições do
4  |  for cada mensagem do
5  |  |  if meu_get >= menor_put then
6  |  |  |  get_ahead[meu_indice] = 1;
7  |  |  |  wait(sem_put_na_frente);
8  |  |  end
9  |  |  criptografa nome para buscar na hash;
10 |  |  busca elemento na hash;
11 |  |  escreve resultado no arquivo;
12 |  end
13 end
```

Algorithm 1: Pseudo-código da busca

3.2 Threads de Inserção

Cada thread de inserção também tem seu próprio buffer de mensagens e variáveis locais. Depois de inserir na hash, cada thread tenta ganhar acesso ao mutex de "ressuscitador de gets", se não ganhar o acesso ao mutex, a thread simplesmente continua sua execução adiante. Se a thread conseguir acesso a esse mutex, ela atualiza a variável *smallest_put_clock*. Depois disso, verifica se tem algum get bloqueado e se tiver, verifica se ele pode voltar a execução novamente comparando o seu clock com o menor dos relógios dos puts. O pseudo-código 2 resume a funcionalidade.

```
1  inicialização;
2  enche o buffer com mensagens do socket;
3  while not fim das requisições do
4  |  for cada mensagem do
5  |  |  criptografa nome;
6  |  |  criptografa telefone;
7  |  |  insere elemento na hash;
8  |  |  if sem_trywait (mutex_reviver) == lock_acquired then
9  |  |  |  atualiza a variável menor_put;
10 |  |  |  for cada thread do
11 |  |  |  |  if get_ahead(thread_num) == 1 then
12 |  |  |  |  |  if get_clock[thread_num] < menor_put then
13 |  |  |  |  |  |  get_ahead[thread_num] = 0;
14 |  |  |  |  |  |  signal(sem_put_na_frente);
15 |  |  |  |  |  end
16 |  |  |  |  end
17 |  |  |  end
18 |  |  end
19 |  end
20 end
```

Algorithm 2: Pseudo-código da inserção

4 Execução

A performance do programa pode ser vista na Tabela 1. É possível ver que o programa fica mais rápido com o aumento do número de 4 para 8 cores de CPU. Nota-se também que um número de threads maior que o número de cores disponíveis pode gerar um overhead e diminuir a performance. Se o número de threads for menor que o número de cores, o tempo se aproxima mais do tempo de uma performance sequencial. Para fins de comparação, o código de referência sequencial do professor roda, em média, em 1m10s na máquina de 8 cores e em 1m16s na máquina de 4 cores. Não foi possível executar na servidora orval porque ela esteve congestionada durante todas as vezes que tentamos (mesmo no meio do semestre).

Table 1: Performance do programa

Num. de cores	Num. de GETs	Num. de PUTs	Tempo
8	2	2	0m50s
8	4	4	0m48s
8	8	8	0m51s
8	16	16	0m51s
4	4	4	1m09s
4	8	8	1m14s
4	16	16	1m15s

5 Primeiras Versões

Em tentativas anteriores, foram feitas implementações como no diagrama abaixo. Nessa versão, havia 1 thread de leitura do socket para cada operação e várias threads de processamento. As threads de leitura faziam a divisão do buffer para cada thread de processamento. Essa implementação ficou, em média, com 10 segundos a mais de tempo de execução que a implementação final. Acreditamos que isso se deu pela quantidade de semáforos que foram utilizados no código, gerando um overhead e também por causa da sequencialização do código quando as threads de leitura de socket tem que esperar pelo término do processamento.

Outro experimento feito anteriormente foi o de fazer uma sincronização mais complexa pros elementos da hash, baseando-se na solução Readers-Writers (com priorização para os Writers) do livro The Little Book of Semaphores. Essa implementação acabou ficando significativamente mais lenta que usar apenas um mutex pra proteger a entrada tanto para inserção quanto para busca. Essa solução seria mais viável se as operações protegidas por esses semáforos adicionais levassem mais tempo a ponto do overhead da sincronização não ser mais tão significativo proporcionalmente.

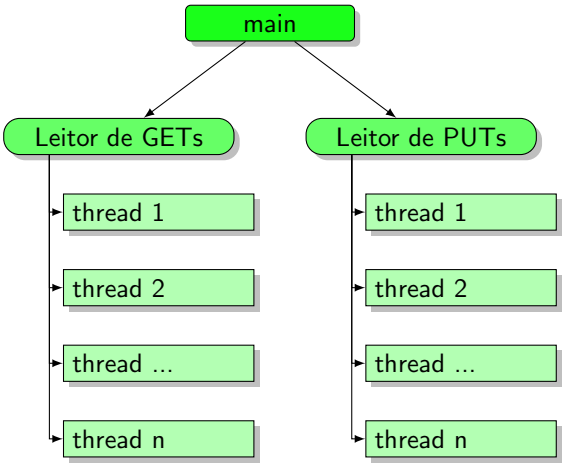

Figure 1: Diagrama da tentativa anterior.

6 Conclusão

Vimos que o desempenho de um programa com múltiplas linhas de execução depende de muitos fatores como o número de *cores* disponíveis na máquina, número de threads do programa, acesso/escrita em memória compartilhada e a sincronização.

O desempenho do nosso programa ainda poderia melhorar no quesito de gerenciamento de memória, já que usamos a função malloc para cada nova inserção e por isso, a memória não fica contígua para a busca nas árvores de colisão, gerando um custo a mais para inserir um elemento dentro da tabela.