Universidade Federal do Paraná
Departamento de Informática

Relatório do Trabalho de Sistemas Operacionais

Alunos: Giovanni Rosa da Silva
Julio Cesar Luppi Doebeli
Luiz Eduardo Cavalheiro
Roberta Samistraro Tomigian
Professor orientador: Luis Carlos Erpen de Bona

Dezembro
2018

Universidade Federal do Paraná
Departamento de Informática

Relatório do Trabalho de Sistemas Operacionais

Trabalho apresentado como requisito parcial à disciplina de Sistemas Operacionais, Departamento de Informática, Universidade Federal do Paraná

Alunos: Giovanni Rosa da Silva
Julio Cesar Luppi Doebeli
Luiz Eduardo Cavalheiro
Roberta Samistraro Tomigian

Professor orientador: Luis Carlos Erpen de Bona

Dezembro
2018

Conteúdo

1 Introdução

Esse relatório tem como principal objetivo otimizar a implementação da agenda telefonica do Professor Casanova, cuja melhoria se dará pelo paralelismo de multi-threads.
Os elementos principais são: Nome, que possui um limite de 15 caraceteres, e Telefone, que possui limite de 9 algarismos. Os dados são criptografados e colocados em uma tabela hash.
O projeto de referência, disponibilizado pelo professor Bona, tem como base um programa com 2 threads, uma para o put (escrita na agenda) e outra pro get (leitura da agenda).
A melhoria sera feita de tal maneira que dados possam ser consultados bem como inseridos na agenda de forma paralela, ou seja, telefones podem ser consultados durante e apos a insercao, respeitando a restricao de que se em uma busca G (numero sequencial), se houverem insercoes (PUTs) com numero de sequencia menor ou igual G, estes tenham sido realizadas.
Para atingir o objetivo desta proposta, utilizamos a biblioteca pthreads, de modo a executar varias threads e permitir a execucao paralela das operacoes.

2 Arquitetura do Sistema

Os principais elementos criados ou modificados foram: a tabela hash, os gerenciamentos do PUT e do GET.

2.1 Tabela Hash

O programa possui uma tabela hash que, em caso de colisão, tem em seu bloco ponteiro para uma lista encadeada, como mostra a estrutura de dados:

```
#include <semaphore.h>

typedef struct linked_list_t *linked_list_t;
typedef struct hash_table_t *hash_table_t;

struct linked_list_t {
  unsigned char key[SIZE_OF_ENCRYPTED_DATA]; // chave da tabela hash
  unsigned char value[SIZE_OF_ENCRYPTED_DATA]; // conte do da tabela hash
  linked_list_t previous, next; // lista duplamente encadeada
};

struct hash_table_t {
  unsigned int size; // tamanho da hash
  linked_list_t *list; // posi   es da hash
};

sem_t *hash_table_mutex; // vetor de mutex para cada posi   o da hash
```

Gera Hash a partir do nome encriptado e faz mod para que o indice fique dentro da tabela.

```
unsigned int hash_table_index(hash_table_t hashTable, unsigned char key[SIZE_OF_ENCRYPTED_DATA]) {
  unsigned int magic = 0x9f3a5d;
  for(int i = 0; i < SIZE_OF_ENCRYPTED_DATA; ++i) {
    magic = ((magic << 5) - magic) +  key[i];
  }
  magic = magic % hashTable->size;
  return magic;
}
```

Alocação inicial de memoria para a tabela hash.

```
hash_table_t hash_table_malloc(unsigned int size) {
  hash_table_t hashTable = (hash_table_t)malloc(sizeof(struct hash_table_t));
```

```
  hash_table_mutex = (sem_t*)malloc(sizeof(sem_t)*size);
  hashTable->list = (linked_list_t*)malloc(sizeof(struct
    linked_list_t)*size);

  hashTable->size = size;
  for(int i = 0; i < size; ++i) {
    sem_init(&hash_table_mutex[i], 0, 1);

    hashTable->list[i] = (linked_list_t)malloc(sizeof(struct
    linked_list_t));
    hashTable->list[i]->previous = NULL;
    hashTable->list[i]->next = NULL;
    for(int j = 0; j < SIZE_OF_ENCRYPTED_DATA; ++j) {
      hashTable->list[i]->key[j] = '\0';
      hashTable->list[i]->value[j] = '\0';
    }
  }
  return hashTable;
}
```

Insere entrada na tabela hash.

```
void hash_table_put(hash_table_t hashTable, unsigned char key[
    SIZE_OF_ENCRYPTED_DATA], unsigned char value[
    SIZE_OF_ENCRYPTED_DATA]) {
  unsigned int index = hash_table_index(hashTable, key);
  linked_list_t auxNoh = hashTable->list[index];

  sem_wait(&hash_table_mutex[index]);

  linked_list_t newNoh = (linked_list_t)malloc(sizeof(struct
    linked_list_t));
  newNoh->previous = NULL;
  newNoh->next = auxNoh;
  for(int j = 0; j < SIZE_OF_ENCRYPTED_DATA; ++j) {
    newNoh->key[j] = key[j];
    newNoh->value[j] = value[j];
  }
  hashTable->list[index] = newNoh;
  sem_post(&hash_table_mutex[index]);
}
```

Pesquisa e retorna entrada da tabela hash, em caso de erro retorna NULL.

```
unsigned char* hash_table_get(hash_table_t hashTable, unsigned
    char key[SIZE_OF_ENCRYPTED_DATA]) {
  unsigned int index = hash_table_index(hashTable, key);
  linked_list_t auxNoh = hashTable->list[index];

  sem_wait(&hash_table_mutex[index]);
  while (auxNoh->next != NULL) {
```

```
      if(!memcmp(auxNoh->key, key, SIZE_OF_ENCRYPTED_DATA *
    sizeof(unsigned char))) {
        sem_post(&hash_table_mutex[index]);
        return auxNoh->value;
      }
      else auxNoh = auxNoh->next;
  }
  sem_post(&hash_table_mutex[index]);
  return NULL;
}
```

Imprime listas encadeadas de entradas da tabela hash.

```
void hash_table_print(hash_table_t hashTable, int showVaues) {
  linked_list_t auxNoh;
  unsigned int i, cont;

  for(i = 0; i < hashTable->size; ++i) {
    auxNoh = hashTable->list[i];
    printf("Hash[%d]: ", i);
    cont = 0;

    if(showVaues){
      while (auxNoh->next != NULL) {
        printf("\n---> key: %s - value: %s\n", auxNoh->key,
    auxNoh->value);
        auxNoh = auxNoh->next;
      }
    } else {
      while ( auxNoh->next != NULL ) {
        cont++;
        auxNoh = auxNoh->next;
      }
      printf("%d\n", cont);
    }
  }
}
```

Libera memória ocupada pelas listas encadeadas e a estrutura da dados da tabela hash.

```
void hash_table_free(hash_table_t hashTable) {
  for(int i = 0; i < hashTable->size; i++) {
    linked_list_t auxNoh = hashTable->list[i];
    linked_list_t nextNoh = auxNoh->next;
    free(auxNoh);
    while(nextNoh) {
      auxNoh = nextNoh;
      nextNoh = auxNoh->next;
      free(auxNoh);
    }
```

```
  }
  free(hashTable->list);
  free(hashTable);
  free(hash_table_mutex);
}
```

2.2 Gerenciamento do PUT

As threads de PUT tem a seguinte estrutura:

```
// estrutura de dados para thread PUT
struct putThread{
  pthread_t thread; // thread rodando PUT
  sem_t sem; // semaforo
  int busy; //  se esta sendo usada
  int m_avail; // numero de mensagens inteiras
  char actual_put[ID_SIZE]; // relogio logico da thread atual
  char buffer[PUT_MESSAGE_SIZE*170]; // mensagens
};
```

Para a sincronização foi feita uma pool de threads para o PUT. Configuramos cada thread para executar em um núcleo específico, dessa forma a cache pode ser aproveitada. Exemplo de inicialização de threads:

```
  // inicia threads
  sem_init(&putThreadsUsed, 0, N_PUT_THREADS);
  pthread_attr_t attr;
  cpu_set_t cpus;
  pthread_attr_init(&attr);
  // le o numero de processadores do pc atual
  int numberOfProcessors = sysconf(_SC_NPROCESSORS_ONLN);

  for (i = 0; i < N_PUT_THREADS; ++i)
  {
    putThreads[i].busy = FALSE;
    sem_init(&putThreads[i].sem, 0, 0);
    memset(putThreads[i].actual_put, 0, ID_SIZE - 1);
    CPU_ZERO(&cpus);
    CPU_SET(i%numberOfProcessors, &cpus);
    // define afinidade da thread para determinado n cleo
    pthread_attr_setaffinity_np(&attr, sizeof(cpu_set_t), &cpus)
    ;
    pthread_create(&putThreads[i].thread, &attr, (void *)&store,
    &putThreads[i]);
  }
```

A execução do PUT acontece da seguinte forma:

Encontra thread livre na pool de threads.

```
    // pool de threads PUT
    sem_wait(&putThreadsUsed);
    for (i = 0; i < N_PUT_THREADS; ++i)
    {
      if (!putThreads[i].busy)
      {
        // escolhe uma thread livre
        actual_thread = &putThreads[i];
      }
    }
```

Sinaliza para a thread que o conteudo do buffer esta pronto para ser consumido pelo método store.

```
    // libera a thread atual para rodar metodo store
    if (read_ret > 0)
    {
      actual_thread->m_avail = m_avail;
      actual_thread->busy = TRUE;
      sem_post(&actual_thread->sem);
    }
```

O método store executa os seguintes comandos:

```
   // roda enquanto PUT nao terminar
  while (!putOver)
  {
  // aguarda sinal para comecar PUT
    sem_wait(&self->sem);
  // sa da r pida
    if (putOver)
      return;

    for (n = 0; n < self->m_avail; n++)
    {
    // encripta nome para usar como chave na hash
      encrypte(ctx_crypt, (unsigned char *)self->buffer + (n *
   PUT_MESSAGE_SIZE) + ID_SIZE, NOME_SIZE, nome_crypt);
      // encripta telefone para inserir na hash
      encrypte(ctx_crypt, (unsigned char *)(self->buffer + (n *
   PUT_MESSAGE_SIZE) + ID_SIZE + NOME_SIZE), FONE_SIZE,
   telefone_crypt);

      // verifica se por algum motivo ja foi inserido o dado
      unsigned char *telefone_hash = searchHash(hashTable,
   nome_crypt);

      if (telefone_hash)
      {
    // ERRO: se foi inserido
```

```
23          fprintf(stderr, "ERRO: %.16s:", self->buffer + (n *
    PUT_MESSAGE_SIZE) + ID_SIZE);
24          BIO_dump_fp(stderr, (const char *)nome_crypt,
    CRYPTEDSIZE);
25          continue;
26        }
27      // insere na hash
28        insertHash(hashTable, nome_crypt, telefone_crypt);
29
30      // atualiza relogio logico da thread PUT atual
31        memcpy(self->actual_put, self->buffer + n *
    PUT_MESSAGE_SIZE, ID_SIZE - 1);
32      }
33
34    // acorda threads GET que eventualmente estejam dormindo
35      wakeupGets();
36
37      // libera thread PUT na pool
38      self->busy = FALSE;
39      sem_post(&putThreadsUsed);
40    }
```

Ao finalizar a leitura dos dados no socket, o PUT finaliza da seguinte maneira:

```
1   // sinaliza que o PUT terminou
2   putOver = TRUE;
3
4   // reseta rel gio l gico das threads PUT
5   for (i = 0; i < N_PUT_THREADS; ++i)
6   {
7     memset(putThreads[i].actual_put, 0xFF, ID_SIZE - 1);
8   }
9
10  // libera as threads GET que est o eventualmente esperando
11  for (i = 0; i < N_GET_THREADS; ++i)
12  {
13    getThreads[i].waitting = FALSE;
14    sem_post(&getThreads[i].sem_getahead);
15  }
16
17  // libera as threads PUT e finaliza
18  for (i = 0; i < N_PUT_THREADS; ++i)
19  {
20    sem_post(&putThreads[i].sem);
21    pthread_join(putThreads[i].thread, NULL);
22  }
```

2.3 Gerenciamento do GET

As threads de GET tem a seguinte estrutura:

```
// estrutura de dados para thread GET
struct getThread{
  pthread_t thread; // thread rodando GET
  sem_t sem;  // sem foro
  int busy; // se est  sendo usada
  int m_avail;  // numero de mensagens inteiras
  char actual_get[ID_SIZE]; // rel gio l gico da thread atual
  sem_t sem_getahead; // sinaliza se GET est  a frente do PUT
  int waitting; // se est  esperando PUT
  char buffer[GET_MESSAGE_SIZE*273]; // mensagens
};
```

Para a sincronização foi feita uma pool de threads para o GET. Configuramos cada thread para executar em um núcleo específico, dessa forma a cache pode ser aproveitada. Exemplo de inicialização de threads:

```
  // inicia threads
  sem_init(&getThreadsUsed, 0, N_GET_THREADS);
  sem_init(&mutexOutput, 0, 1);
  pthread_attr_t attr;
  cpu_set_t cpus;
  pthread_attr_init(&attr);
  // le o numero de processadores do pc atual
  int numberOfProcessors = sysconf(_SC_NPROCESSORS_ONLN);

  for (i = 0; i < N_GET_THREADS; ++i)
  {
    getThreads[i].busy = FALSE;
    sem_init(&getThreads[i].sem, 0, 0);
    sem_init(&getThreads[i].sem_getahead, 0, 0);
    memset(getThreads[i].actual_get, 0, ID_SIZE - 1);
    getThreads[i].waitting = FALSE;
    CPU_ZERO(&cpus);
    CPU_SET(i%numberOfProcessors, &cpus);
    // define afinidade da thread para determinado nucleo
    pthread_attr_setaffinity_np(&attr, sizeof(cpu_set_t), &cpus)
    ;
    pthread_create(&getThreads[i].thread, &attr, (void *)&
    retrieve, &getThreads[i]);
  }
```

A execução do GET acontece da seguinte forma:

Encontra thread livre na pool de threads.

```
    // pool de threads GET
    sem_wait(&getThreadsUsed);
```

```
    for (i = 0; i < N_GET_THREADS; ++i)
    {
      if (!getThreads[i].busy)
      {
        // escolhe thread livre
        actual_thread = &getThreads[i];
      }
    }
```

Sinaliza para a thread que o conteudo do buffer esta pronto para ser consumido pelo método store.

```
    // libera a thread atual para rodar metodo store
    if (read_ret > 0)
    {
      actual_thread->m_avail = m_avail;
      actual_thread->busy = TRUE;
      sem_post(&actual_thread->sem);
    }
```

O método store executa os seguintes comandos:

```
   // roda enquanto PUT nao terminar
  while (!putOver)
  {
  // aguarda sinal para comecar PUT
    sem_wait(&self->sem);
  // sa da r pida
    if (putOver)
      return;

    for (n = 0; n < self->m_avail; n++)
    {
    // encripta nome para usar como chave na hash
      encrypte(ctx_crypt, (unsigned char *)self->buffer + (n *
   PUT_MESSAGE_SIZE) + ID_SIZE, NOME_SIZE, nome_crypt);
      // encripta telefone para inserir na hash
      encrypte(ctx_crypt, (unsigned char *)(self->buffer + (n *
   PUT_MESSAGE_SIZE) + ID_SIZE + NOME_SIZE), FONE_SIZE,
   telefone_crypt);

      // verifica se por algum motivo ja foi inserido o dado
      unsigned char *telefone_hash = searchHash(hashTable,
   nome_crypt);

      if (telefone_hash)
      {
    // ERRO: se foi inserido
        fprintf(stderr, "ERRO: %.16s:", self->buffer + (n *
   PUT_MESSAGE_SIZE) + ID_SIZE);
```

```
24          BIO_dump_fp(stderr, (const char *)nome_crypt,
    CRYPTEDSIZE);
25          continue;
26        }
27      // insere na hash
28        insertHash(hashTable, nome_crypt, telefone_crypt);
29
30      // atualiza relogio logico da thread PUT atual
31        memcpy(self->actual_put, self->buffer + n *
    PUT_MESSAGE_SIZE, ID_SIZE - 1);
32      }
33
34    // acorda threads GET que eventualmente estejam dormindo
35      wakeupGets();
36
37      // libera thread PUT na pool
38      self->busy = FALSE;
39      sem_post(&putThreadsUsed);
40    }
```

Ao finalizar a leitura dos dados no socket, o PUT finaliza da seguinte maneira:

```
1   // sinaliza que o PUT terminou
2   putOver = TRUE;
3
4   // reseta rel gio l gico das threads PUT
5   for (i = 0; i < N_PUT_THREADS; ++i)
6   {
7     memset(putThreads[i].actual_put, 0xFF, ID_SIZE - 1);
8   }
9
10  // libera as threads GET que est o eventualmente esperando
11  for (i = 0; i < N_GET_THREADS; ++i)
12  {
13    getThreads[i].waitting = FALSE;
14    sem_post(&getThreads[i].sem_getahead);
15  }
16
17  // libera as threads PUT e finaliza
18  for (i = 0; i < N_PUT_THREADS; ++i)
19  {
20    sem_post(&putThreads[i].sem);
21    pthread_join(putThreads[i].thread, NULL);
22  }
```

Estruturas de dados (lista encadeada e habela hash)

```
typedef struct linked_list_t *linked_list_t;
typedef struct hash_table_t *hash_table_t;

struct linked_list_t {
  unsigned char key[SIZE_OF_ENCRYPTED_DATA];
  unsigned char value[SIZE_OF_ENCRYPTED_DATA];
  linked_list_t previous, next;
};

struct hash_table_t {
  unsigned int size;
  linked_list_t *list;
};
```

3 Experimentos

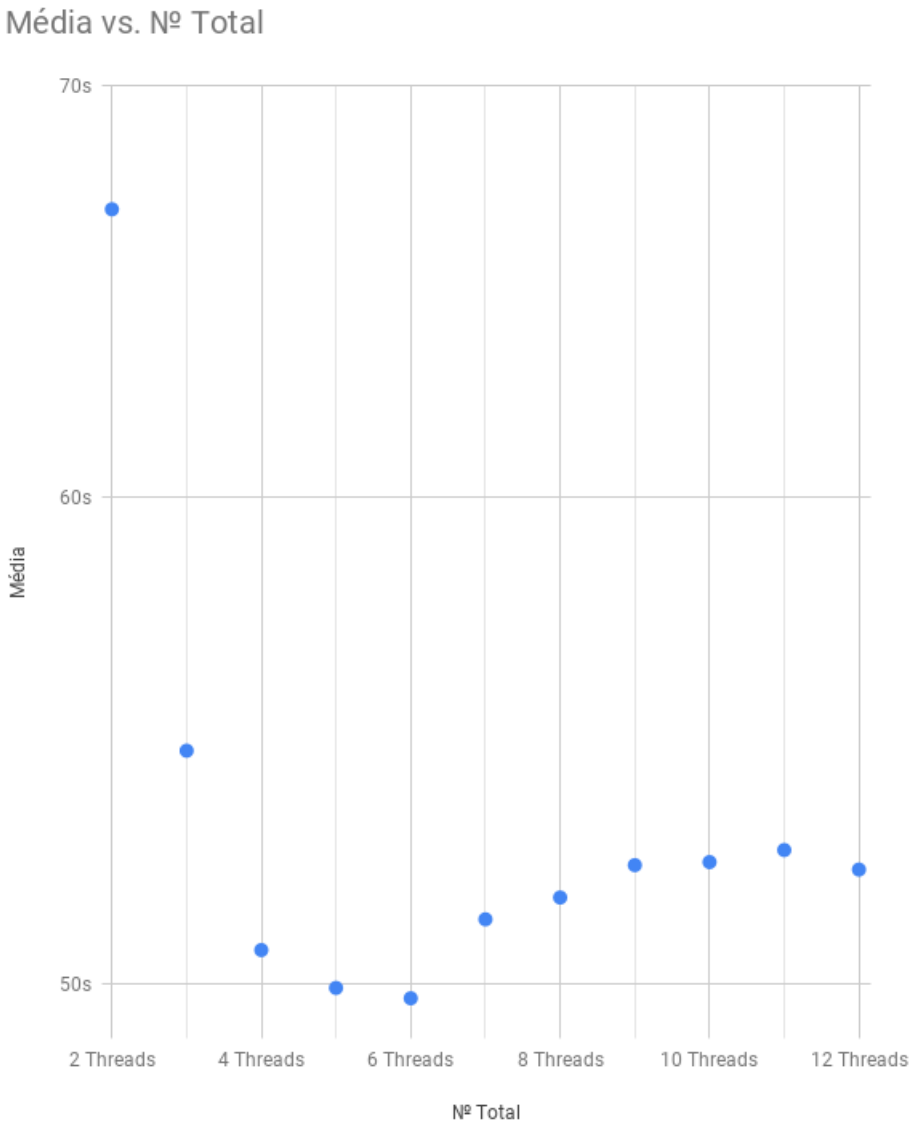

Figura 1: Legenda

Número de Threads			Resultado dos Testes				
№ Total	№ PUT	№ GET	Teste 1	Teste 2	Teste 3	Média	Menor Média
2	1	1	66,19	66,95	67,35	66,83	**66,83**
3	1	2	55,75	53,47	54,46	54,56	**54,56**
	2	1	62,27	62,61	61,97	62,28	
4	1	3	55,67	61,77	61,85	59,76	**50,63**
	2	2	50,25	51,32	50,33	50,63	
	3	1	70,1	68,34	68,98	69,14	
5	1	4	56,28	56,56	57,27	56,7	**49,92**
	2	3	50,69	52,77	51,49	51,65	
	3	2	49,88	50,03	49,85	49,92	
	4	1	63,92	63	63,63	63,52	
6	1	5	59,11	60	58,84	59,32	**49,73**
	2	4	51,81	51,77	53,03	52,2	
	3	3	50,13	50,32	50,13	50,19	
	4	2	49,89	49,36	49,93	49,73	
	5	1	65,98	66,91	67,11	66,67	
7	1	6	58,57	59,69	60,03	59,43	**51,22**
	2	5	52,26	53,22	52,67	52,72	
	3	4	51,25	51,81	52,18	51,75	
	4	3	51,14	51,48	51,04	51,22	
	5	2	52,64	52,86	53,07	52,86	
	6	1	65,37	64,14	64,05	64,52	
8	1	7	60,08	61,1	61,64	60,94	**51,64**
	2	6	53,3	53,72	54,38	53,8	
	3	5	51,67	51,39	51,92	51,66	
	4	4	51,54	51,8	51,81	51,72	
	5	3	51,82	51,45	51,65	51,64	
	6	2	53,49	51,61	52,73	52,61	
	7	1	64,07	63,41	66,44	64,64	
9	1	8	59,84	60,98	60,31	60,38	**52,27**
	2	7	54,26	54,33	54,57	54,39	
	3	6	51,91	52,4	52,5	52,27	
	4	5	52,03	52,61	52,24	52,29	
	5	4	54,69	52,36	52,15	53,07	
	6	3	56,26	55,26	51,85	54,46	
	2	8	52,44	53,12	53,25	52,94	
	3	7	53,64	54,81	53,65	54,03	
	4	6	55,33	53,74	54,36	54,48	
	5	5	52,27	51,85	52,87	52,33	
	6	4	55,21	55,9	55,2	55,44	
	7	3	55,77	56,3	54,2	55,42	
	8	2	56,69	58,4	60,06	58,38	
12	1	11	67,14	66,71	67,55	67,13	**52,18**
	2	10	52,65	53,67	52,78	53,03	
	3	9	52,35	52,06	52,14	52,18	
	4	8	54,24	54,12	55,25	54,54	
	5	7	55,26	57,1	54,84	55,73	
	6	6	53,09	56,87	54,16	54,71	
	7	5	56,39	59,06	56,42	57,29	
	8	4	56,48	56,84	55,95	56,42	

Figura 2: Legenda

4 Conclusão

O nosso trabalho implementou uma tabela hash multi-threaded, que armazena e busca informações telefônicas de forma paralela e eficiente. Cada linha da tabela hash consiste em lista encadeada de chaves e um valores. Às chaves, ou o nome do contato, é aplicada uma função hash que a transforma no índice da tabela. O valor, ou número do telefone, é então armazenado na lista encadeada adequada da tabela.