Relatório
CI215 - Sistemas Operacionais
Agenda telefônica

Aluno A
Aluno B
Departamento de informática,
Universidade Federal do Paraná - UFPR

24 de novembro de 2017

1 Introdução

Neste relatório iremos relatar uma melhoria na implementação da agenda telefônica do professor Casanova, cuja melhoria se dará principalmente pelo paralelismo das operações através de multi-threads. A implementação original do professor, se dá através da inserção de nomes e telefones ($chave \leq valor$), e consultar telefones de maneira sequencial de cada operação. A melhoria será 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 após a inserção, respeitando a restrição de que se em uma busca G# (número sequencial), se houverem inserções (PUTs) com número de sequencia menor ou igual G#, estes tenham sido realizadas.

Para atingir o objetivo desta proposta, utilizamos a biblioteca pthreads, de modo a executar várias threads e permitir a execução paralela das operações. Além disso, desenvolvemos uma hashtable thread-safe, para que diversas operações simultâneas possam ser concluídas de forma concisa.

2 Arquitetura do Sistema

O sistema foi dividido em três grandes partes: o *put*, o *get* e a *hashtable*.

Para sincronizar as diversas threads será construído um pool de threads que quando há serviços os mesmos são atribuídos as threads no estado idle.

Como o pool é iniciado, exemplo de put:

```
int put_pronta = 0;
int *k;
sem_init(&sem_put_pronta, 0, 0);
for (i = 0; i < PUT_THREADS; i++){
  k = (int *) malloc (sizeof(int));
  *k = i;
  sem_init(&sem_put_servico[i], 0, 0);
  sem_init(&sem_wakeup_get, 0, 1);
  put_threads_list[i] = 0;
  put_threads_actual[i] = 0xffffffff;
  put_threads_buffer[i][PUT_MESSAGE_SIZE] ='\0';
  pthread_create(&put_threads[i], NULL,
   (void*) &store_mult, (void *) k);
 }
```

Encontrando alguma thread livre:

```
//espera alguma thread estar pronta
sem_wait(&sem_put_pronta);
//descobre qual esta pronta, seta put_pronta
for(i = 0; i< PUT_THREADS; i++){
  if(put_threads_list[i] == 1){
    put_pronta = i;
    //fprintf(stderr,"%d ", i);
    break;
  }
 }
```

Uma vez que obtido o que uma thread deve fazer, envia a quantidade de mensagens e sinaliza para thread que o conteúdo do buffer está pronto para ser consumido:

```
put_threads_mavail[put_pronta] = m_avail;

if(read_ret > 0) {
  put_threads_list[put_pronta] = 0;
  sem_post(&sem_put_servico[put_pronta]);
}
```

Quando não há mais mensagens é feito o join das threads, como o trecho de código abaixo:

```
//espera todas as threads terminarem
i = 0;
ac = 0;
sem_destroy(&sem_put_pronta);
while( 1 ) {
  if(ac == PUT_THREADS)
    break;
  if(put_threads_list[i] == 1){
    put_threads_mavail[i] = -1;
    sem_post(&sem_put_servico[i]);
    pthread_join(put_threads[i], NULL);
    sem_destroy(&sem_put_servico[i]);
    fprintf(stderr,"Thread %d: Acabou\n", i);
    ac++;
  }
  i++;
  if(i == PUT_THREADS)
    i = 0;
 }
```

Para tratar quando há dados no buffer é sinalizado para a thread de controle que há threads no estado idle, além de liberar a mémória quando recebe o sinal do controlador para fechar.

```
EVP_CIPHER_CTX *ctxe;
if(!(ctxe = EVP_CIPHER_CTX_new())) handleErrors();
int my_id = *((int *) id);
free(id);
int n;
put_threads_list[my_id] = 1;
sem_post(&sem_put_pronta);

while( 1 ) {
  sem_wait(&sem_put_servico[my_id]);

  if(put_threads_mavail[my_id] == -1)
    break;

  //consome
  for (n =0; n < put_threads_mavail[my_id]; n++) {
    store(put_threads_buffer[my_id]+ID_SIZE+
```

```
            n*PUT_MESSAGE_SIZE, ctxe);
    }
    char * last_insert = put_threads_buffer[my_id]+
      (n-1)*PUT_MESSAGE_SIZE;
    last_insert[ID_SIZE] = '\0';

    put_threads_actual[my_id] =
      strtol(last_insert, NULL, 16);
    //ACORDA THREADS DE GET
    //
    sem_wait(&sem_wakeup_get);
    for(int i =0; i < GET_THREADS; i++) {
      if(get_threads_waiting[i])
        sem_post(&sem_get_wating[i]);
    }
    sem_post(&sem_wakeup_get);

    put_threads_list[my_id] = 1;
    sem_post(&sem_put_pronta);

  }
put_threads_list[my_id] = -1;
EVP_CIPHER_CTX_free(ctxe);
```

2.1 Hashtable

A implementação da *hashtable* foi construída através de uma lista de *unsigned chars*, alocada com blocos de tamanho fixo.

As buscas na *hashtable* não necessita de nenhuma estratégia de sincronização de threads, visto que as inserções não altera o que já foi salvo. Para as inserções, utilizamos um semáforo para implementarmos exclusão mútuta, na descoberta da posição da inserção. A cópia dos dados não sofre de exclusão mútua, visto que garantimos regiões diferentes no *mutex* da descoberta. Esta lista é definida pelas seguintes tipos:

```
#define LIST_BLOCK_STORAGE 15
#define LIST_BLOCK_DATA_SIZE 32
#define LIST_BLOCK_SIZE 480

typedef struct List_block {
  // bloco de dados, onde chaves e
```

```
  // valores sao armazenados continuamente
  // na memoria.

  unsigned char data[LIST_BLOCK_SIZE];
  void *align1, *align2, *align3;
  struct List_block *next;
} list_block_t;

typedef struct {
  int size;
  list_block_t *head, *tail;
  sem_t mutex;    // mutex para insercoes paralelas
} list_t;

void list_init (list_t *);

void list_insert (list_t *, unsigned char *,
                  unsigned char *);

unsigned char *list_find (list_t *, unsigned char *);

void list_destroy (list_t *);
```

Além da lista, foi definido a estrutura de dados *hashtable*, de modo a mapear varias listas para o caso de colisão de chaves. Essa estrutura de dados é apresentado abaixo:

```
#define HT_MAX_BITS 18
#define HT_MAX_SIZE (1<<(HT_MAX_BITS))
#define ht_map_int(a) ((a)&( ( 1<<( HT_MAX_BITS ) ) -1))

#define HT_DESTROY_TH 20
#define HT_INIT_TH 1


typedef struct {
  // numero de itens na tabela
  unsigned int count;
  // array de lista (buckets)
  list_t *b;
} ht_hashtable_t;
```

```
typedef ht_hashtable_t* ht_hashtable;

// As seguintes interfaces sao oferecidas
// para operar com a hashtable

// retorna nova hashtable
ht_hashtable ht_init ();

// insere elemento na tabela
void ht_insert (unsigned char *, unsigned char *,
            ht_hashtable);

// busca e retorna elemento na tabela
// caso nao tenha, retorna NULL
unsigned char* ht_get (unsigned char *, ht_hashtable);

// funcao de hashing, para melhorar a distribuicao
void ht_destroy (ht_hashtable);
```

No trecho de código abaixo mostra onde foi implementado em nossa arquitetura para respeitar a restrição $G\# \leq P\#$, ou seja, enquanto houver sequencias gets menor ou igual que puts este deve ser efetivado primeiro antes de retornar o get.

```
int existe_menor;
char *ptr;
long long int actual_id;

while( 1 ) {
  sem_wait(&sem_get_servico[my_id]);

  if(get_threads_mavail[my_id] == -1)
    break;

  //consome
  for (n =0; n < get_threads_mavail[my_id]; n++){
    ptr = ((char*)&(get_threads_buffer[my_id]))+n*
      GET_MESSAGE_SIZE+ID_SIZE;
    actual_id = strtol(get_threads_buffer[my_id]+n*
                  GET_MESSAGE_SIZE,&ptr, 16);
    do{
      existe_menor = 0;
```

```
    for (i = 0; i < PUT_THREADS; i++){
      if (put_threads_actual[i] != -1 && actual_id >
      put_threads_actual[i]){
        get_threads_waiting[my_id] = 1;
        sem_wait(&sem_get_wating[my_id]);
        get_threads_waiting[my_id] = 0;
        existe_menor=1;
      }
    }
  }while(existe_menor==1);
  retrieve(get_threads_buffer[my_id]+ID_SIZE+n*
       GET_MESSAGE_SIZE, fp, ctxe, ctxde);
```

3 Experimentos

Todos os experimentos desta implementação foram executados nos hosts do Lab4 do departamento de informática da Universidade Federal do Paraná, cuja configuração é: Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz, 8 cores 8GB de RAM.

A versão *casanova-am* apresentou o seguinte resultado:

Tabela 1: Versão da agenda original.

TAM. CONJ.	**THREADS PUT/GET**	**TIME segundos**	**MEMORY MB**
20.10^6	1/1	64.14	4035

Na primeira implementação construímos uma versão da hashtable pouca otimizada que resultou na seguinte saída:

Tabela 2: Primeira versão da agenda de telefone

TAM. CONJ.	**THREADS PUT/GET**	**HTSIZE bits**	**BLKSIZE**	**TIME segundos**	**MEMORY MB**
20.10^6	2/2	4	15	*	1010
	2/2	4	15	*	640
	2/2	18	15	44.35	751

* Não terminou em 5 minutos.

Observe na tabela 2, na última linha, fizemos um tunning no número de bits da hashtable, no qual resultou em um melhor resultado, com um custo adicional de memória.

Tabela 3: Versão final da agenda de telefones

TAM. CONJ.	**THREADS PUT/GET**	**HTSIZE bits**	**BLKSIZE**	**TIME segundos**	**MEMORY MB**
20.10^6	2/4	18	15	44.16	750
	4/4	18	15	44.79	751
	4/8	18	15	45.79	751
	8/4	18	15	50.05	752
	4/2	18	15	46.99	751
	2/4	18	15	50.05	751
	2/6	18	15	43.85	747
	6/2	18	15	49.75	751
	16/8	18	15	55.07	750
	8/8	18	15	49.59	751
	2/16	18	15	43.80	751
	1/7	18	15	44.98	751

Observamos que na tabela 3 quando adicionamos mais threads put, não ganhamos desempenho, em contra partida aumentando as get ganhamos.

4 Conclusão

Os experimentos revelaram que existe uma forte relação de como é desenvolvido o algoritmo, e como este se relaciona com as estruturas de dados, nem sempre paralelizar terá um ganho de desempenho, como mostrado na tabela 3 na última linha em que put possui apenas uma thread e sete outras threads para processar os gets, o tempo ficou próximo dos demais melhores tempos com mais threads para put.

Na primeira implementação deste trabalho, percebemos que para ter um bom programa multithread é necessário construir estruturas de dados que possam suportar acesso paralelo de forma consistente. Então imaginamos inicialmente, que apenas aumentar o número de threads seria suficiente, com tudo, os experimentos empíricos mostrou ser diferente e tivemos que adaptar a hashtable para ter algum ganho de desempenho. Junto com o aumento do desempenho, tivemos uma drástica redução no consumo de memória.