UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE ENG. DE CONTROLE, AUTOMAÇÃO E COMPUTAÇÃO
CURSO DE GRADUAÇÃO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Bruno Bueno Bronzeri
Gabriel Mariano Gonçalves Santos
Leonardo dos Santos Schmitt

Trabalho 1: Multithreading para Ataque de Força Bruta em Lista de Hashes de Senhas

Blumenau
2024

Bruno Bueno Bronzeri
Gabriel Mariano Gonçalves Santos
Leonardo dos Santos Schmitt

Trabalho 1: Multithreading para Ataque de Força Bruta em Lista de Hashes de Senhas

Relatório referente ao Trabalho número 1 (um) da disciplina de Sistemas Computacionais para Controle e Automação, no curso de graduação de Engenharia de Controle e Automação na Universidade Federal de Santa Catarina (UFSC) - Campus Blumenau.
Orientador:
Prof. Carlos Roberto Moratelli, Dr.

Blumenau
2024

RESUMO

Este trabalho visa abordar um algoritmo que implementa um produtor/consumidor visando paralelizar um ataque de força bruta em uma lista de *hashes* de senhas de um sistema Linux. Esse ataque é feito através de uma técnica chamada de *Multithreading*. Dessa forma a partir de um arquivo de *hashes*, a *thread* produtora disponibiliza para n consumidoras que, a partir de um dicionário criptografam senhas em *hashes* e efetuam a comparação uma a uma até obter correspondência.

Palavras-chave: Algoritmo; força bruta, *hashes*, *Multithreading*.

ABSTRACT

This report aims to approach an algorithm that implements a producer/consumer with the aim of parallelize an Brute-force attack in a list of password hashes of a Linux system.This attack is done through a technique called Multithreading. Therefore, from a hash file, the producing thread available to n consumers who, using a dictionary, encrypt passwords in hashes and compare them one by one until a match is found.

Keywords: Algorithm; Brute-force, hashes, multithreading.

LISTA DE FIGURAS

Figura 1 – Gráfico de tempo por número de *threads* comparando desempenho. 14
Figura 2 – Gráfico de tempo por número de *threads* para Máquina Virtual. 15
Figura 3 – Gráfico de tempo por número de *threads* para Dual-Boot. 16
Figura 4 – Gráfico de tempo por número de *threads* para Servidor da UFSC. 16

LISTA DE CÓDIGOS

3.1 Criação de N *threads* consumidoras. 9
3.2 Aquisição de parâmetros. 10
3.3 Funções para contagem de número de linhas de arquivos. 11

SUMÁRIO

1	**IDENTIFICAÇÃO DO TRABALHO**	**7**
1.1	TÍTULO	7
1.2	TEMA PRINCIPAL	7
2	**INTRODUÇÃO**	**8**
3	**DESENVOLVIMENTO**	**9**
4	**RESULTADOS**	**13**
5	**CONCLUSÃO**	**17**
	REFERÊNCIAS	**18**

1 IDENTIFICAÇÃO DO TRABALHO

1.1 TÍTULO

Trabalho 1 - Ataque de Força Bruta via *Multithreading*.

1.2 TEMA PRINCIPAL

O trabalho em questão consiste na tentativa de realizar um ataque de força bruta em um conjunto de palavras *hash*, para descobrir as senhas utilizadas pelos usuários de um determinado sistema. Ou seja, através da lista de *hashes* que representam senhas criptografadas, é possível juntamente com um arquivo dicionário de senhas reais, checar a correspondência, na qual caso haja, considera-se a senha como descoberta. Esse processo é realizado com o uso de *threads*, a fim de tornar a execução paralelizada e otimizada em relação ao tempo.

2 INTRODUÇÃO

Neste trabalho, serão aplicados os conceitos estudados na disciplina de Sistemas Computacionais para Controle e Automação. Para isso, foi proposto um cenário de ataque cibernético. No caso, foram fornecidos dois dados a serem utilizados, o primeiro deles é o arquivo `rockyou.txt`, e o outro é o arquivo `hashes.txt`.

O primeiro trata-se de um dos dicionários de senhas mais famosos e amplamente utilizados. Ele contém uma lista enorme de senhas comuns e populares, sendo frequentemente utilizado em testes de segurança e auditorias de sistemas para verificar a força das senhas. O nome "*rockyou*"deriva de um ataque de hackers em 2009 contra o site de redes sociais *RockYou*, no qual milhões de senhas foram vazadas e muitas delas eram bastante simples e comuns, como aquelas encontradas no arquivo "`rockyou.txt`". Já o segundo arquivo, seria referente a um conjunto de *hashes*, que tratam-se justamente de senhas criptografadas de tipo MD5, que no cenário, seriam senhas vazadas como resultado de um primeiro ataque, para a posteriori extrair a informação de criptografia dessa palavra *hash* e criptografar senha a senha do dicionário para encontrar correspondências.

Vale ressaltar que a criptografia usada em todas as *hashes* são do tipo MD5, já que todas elas se iniciam com 1. E que o fragmento da *hash*, chamado de "*salt*"é uma sequência aleatória concatenada à senha real para gerar a criptografia, de modo com que não seja possível possuir um dicionário prévio de *hashes*.

3 DESENVOLVIMENTO

A implementação do algoritmo que realiza o ataque para descobrir senhas de usuários de um determinado sistema ou plataforma, foi feito através da utilização de *threads*. Uma produtora na função `main`, que disponibiliza as senhas no formato de *hashes*, ou seja, senhas criptografadas a serem descobertas, no arquivo `hashes.txt` tornando o *buffer* da *thread* produtora o próprio arquivo. Além disso, o algoritmo deve criar N *threads* consumidoras, conforme o número passado como parâmetro de entrada pelo usuário/hacker, as quais "consomem"uma *hash* da lista e através do *salt*, cada *thread* adquire o padrão de criptografia, acessando o arquivo `rockyou.txt`, senha por senha, criptografando-as para esse padrão e testando sua correspondência até descobrir todas as senhas criptografadas.

Após a criação das *threads* consumidoras (Código 3.1), e das variáveis globais, tais como número de *hashes*, número de senhas e contador de *hash* para as *threads*, e locais, tal como uma variável que identifica as *threads* em um vetor, devem ser obtidos os parâmetros de entrada do usuário, que são respectivamente o número de *threads* a serem criadas e os arquivos `rockyou.txt` e `hashes.txt` (Código 3.2). Por seguinte, durante a execução do algoritmo a *thread* produtora tem como *buffer* o próprio arquivo de *hashes* e a mesma é responsável também por obter o número de linhas dos dois arquivos através das funções `loadpasswd()` e `loadhash()`, como visto no Código 3.2. Já em relação às *threads* consumidoras, as mesmas possuem número de identificação, tanto para criação das mesmas, como também, para que a produtora espere as *threads* consumidoras em questão finalizarem suas tarefas através da função `pthread_join(cons[m], NULL)` dentro de um laço de repetição que percorre as 'm' *threads*, finalizando-as.

Listing 3.1 – Criação de N *threads* consumidoras.

```
1  // Creating threads
2  pthread_t *cons;
3  int *ids;
4
5  /*Cria array com os numeros de identificacao das
       threads. */
6  nc = atoi(argv[1]);
7  ids = (int *)malloc(nc * sizeof(int));
8  for(i=0; i<nc; i++){
9      ids[i] = i;
```

```
10      }
11
12       /* Cria diversas threads consumidoras */
13      cons = (pthread_t *)malloc(nc * sizeof(pthread_t));
14      for(k=0; k<nc; k++){
15          if(pthread_create(&cons[k], NULL, consumidor, (void
               *)&ids[k])) {
16              fprintf(stderr, "Error creating thread\n");
17              return 2;
18          }
19      }
```

Fonte: Desenvolvido pelos Autores.

Listing 3.2 – Aquisição de parâmetros.

```
1       npasswd = loadpasswd(argv[2]);
2       nhash = loadhash(argv[3]);
3       nc = atoi(argv[1]);
```

Fonte: Desenvolvido pelos Autores.

Por conseguinte para que as *threads* consumidoras sejam capazes de comparar as *hashes* e descobrir as senhas, o algoritmo foi implementado de modo com que cada *thread* criada pudesse adquirir uma palavra *hash*, extrair o *salt* e percorrer todo o dicionário, criptografando cada senha e efetuando a comparação. Dessa forma, é notável que cada *hash* a ser descoberta foi associada ao valor de sua linha, ou seja, cada *thread* consumidora pegaria uma *hash* de acordo com esse valor, operando o laço de repetição no dicionário e após finalizado, adquiriria a próxima *hash* ainda não analisada.

Contudo, foi identificado uma região crítica justamente no momento de adquirir uma *hash* e alterar o valor do contador de linhas, indicando que aquela linha de *hash* já teria sido utilizada por outra *thread* ou estar em execução. Então, como é possível visualizar no Código 3.3, a região crítica foi isolada por um *Mutex*, fazendo com que a *thread* que atingir primeiro essa região, possa fechar o *Mutex*, adquirir sua palavra *hash*, somar no contador - indicando que aquela linha já está sendo executada para não haver sobreposição - e abrir o *Mutex*, possibilitando que outra *thread* faça o memso processo sem causar condição de corrida ou fazer com que duas ou mais *threads* adquiram a mesma *hash* e façam trabalho dobrado.

Listing 3.3 – Funções para contagem de número de linhas de arquivos.

```
int compara(){

    // The password hash from the shadow file (user-
       provided example)
    char *shadow_hash;
    char salt[12];

    struct crypt_data crypt_data;

    pthread_mutex_lock(&espera_thread);

    shadow_hash = hash_list[j];
    // Extracting the salt from the shadow_hash, it
       includes "$1$" and ends before the second "$"
    strncpy(salt, shadow_hash, 11);
    salt[11] = '\0';  // Ensure null termination
    j++;

    pthread_mutex_unlock(&espera_thread);

    for (int i=0; i<npasswd; i++) {
        // em multithread use crypt_r(), pois crypt() nao e
            threadsafe.
        char *new_hash = crypt_r(password_list[i], salt, &
           crypt_data);
        if (strcmp(shadow_hash, new_hash) == 0) {
            printf("Password found: %s\n", password_list[i
               ]);
            i = npasswd;
            flag = 1;
        }
    }
    if(flag == 0){printf("Password not found!\n");}
}
```

Fonte: Desenvolvido pelos Autores.

Ademais, ainda no Código 3.3, é possível inferir que a função `crypt()` por não ser "*thread safe*", foi alterada para a função *crypt_r()*, a qual exige um parâmetro extra para garantir a "*thread safety*", no caso uma estrutura de dados, "`struct crypt_data`". Ao passar essa estrutura como um argumento para `crypt_r()`, é garantido que cada *thread* tenha sua própria área de armazenamento para esses dados temporários. Isso impede que threads concorrentes se sobreponham ou alterem inadvertidamente o estado interno da função, tornando-a segura para uso em ambientes *multithread* e dispensando a necessidade do uso variáveis globais ou estáticas para armazenar dados temporários.

E por fim, vale ressaltar o uso de uma *flag* que tem valor global zero, em que toda vez que uma *thread* descobre a senha, esse valor é alterado para 1. Isso possibilita que, caso permaneça em zero esse valor, o programa entenda que a senha não foi descoberta, imprimindo isso na tela ("`Password not found!`").

4 RESULTADOS

Diante do trabalho previamente apresentado, é imprescindível destacar os resultados obtidos. Para isso, comparam-se os tempos de execução do programa para o sistema *multithreading*, validando, dessa forma, o comportamento esperado de otimização do processamento. Para fins de comparação, realizou-se a simulação em três ambientes diferentes, sendo eles, um computador pessoal, uma máquina virtual e o servidor na nuvem da UFSC. Para apresentar os resultados de maneira condizente, é importante detalhar as especificações de cada ambiente utilizado, assim:

Máquina Virtual:

- Processador: I7 de 11ª geração;
- 4 Núcleos ;
- 8 *Threads* para cada núcleos.

Máquina com Dual-Boot:

- Processador: I5 de 10ª geração;
- 4 Núcleos;
- 8 *Threads* para cada núcleos.

Servidor

- 8 Núcleos;
- 16 *Threads* para cada núcleos.

Em vista das especificações anteriores, e após executar o algoritmo para cada ambiente, é possível obter diferentes resultados. Entretanto, percebe-se que ao aumentar a quantidade de *threads*, ocorre uma saturação em relação ao tempo de execução, ou seja não há uma melhora no desempenho, conforme apresentado na Figura 1. E com essas informações é possível inferir que o número de núcleos possui relação com a quantidade exata de *threads* para máximo proveito do processador. No caso do Dual-boot (4 núcleos) e da Máquina Virtual (3 núcleos), a partir de 8 *threads* não há melhora de desempenho. Já para o servidor, com 8 núcleos, a partir de 16 *threads* não há melhora de desempenho.

Abaixo na Tabela 1 é possível visualizar os valores de tempo obtidos para a variação de *threads* para as execuções feitas nos três computadores, assim como, na Figura 3 é possível visualizar graficamente o decaimento do tempo em relação a *threads*.

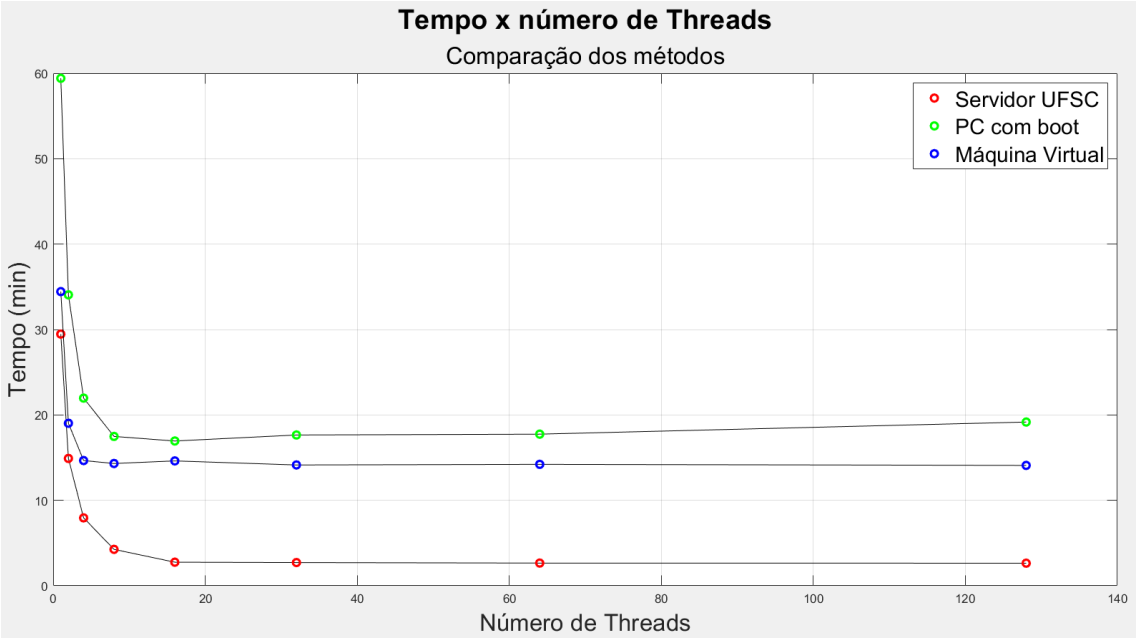
Em primeiro instante, analisando Figura 2 é possível visualizar que quando executado em uma máquina virtual, como mencionado anteriormente, 8 *threads*

Tabela 1 – Apresentação de parâmetros das funções de transferência para sistemas teóricos.

Relação computador e tempo [min] de execução por 'n' *threads*			
Máquina / nº *threads*	**Dual-boot**	**Máquina Virtual**	**Servidor da UFSC**
1 *thread*	59.414	34.4404	29.4631
2 *threads*	34.0618	19.0267	14.9092
4 *threads*	21.9708	14.6597	7.9469
8 *threads*	17.4858	14.3137	4.2600
16 *threads*	16.9472	14.6166	2.7581
32 *threads*	17.6500	14.1409	2.7178
64 *threads*	17.7500	14.2153	2.6548
128 *threads*	19.1667	14.0911	2.6420

Fonte: Elaborada pelos autores.

Figura 1 – Gráfico de tempo por número de *threads* comparando desempenho.

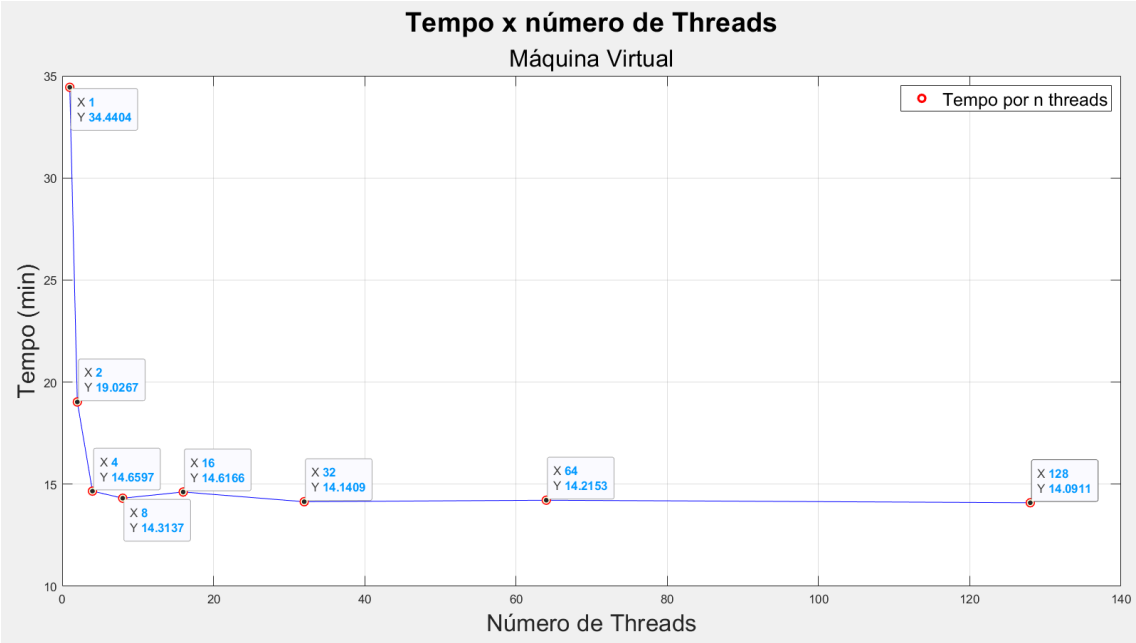


Fonte: Desenvolvido pelo Autor.

obtém a melhor otimização, já que mais que isso não apresenta melhora de tempo. Além disso vale inferir que de 1 para 8 *threads*, o ganho do tempo é exponencial.

Já quando executado em ambiente com *Dual Boot* o desempenho se torna inferior quando comparado com a máquina virtual, haja vista que o processador é inferior, o que pode gerar um atraso na simulação. No entanto, assim como na

Figura 2 – Gráfico de tempo por número de *threads* para Máquina Virtual.

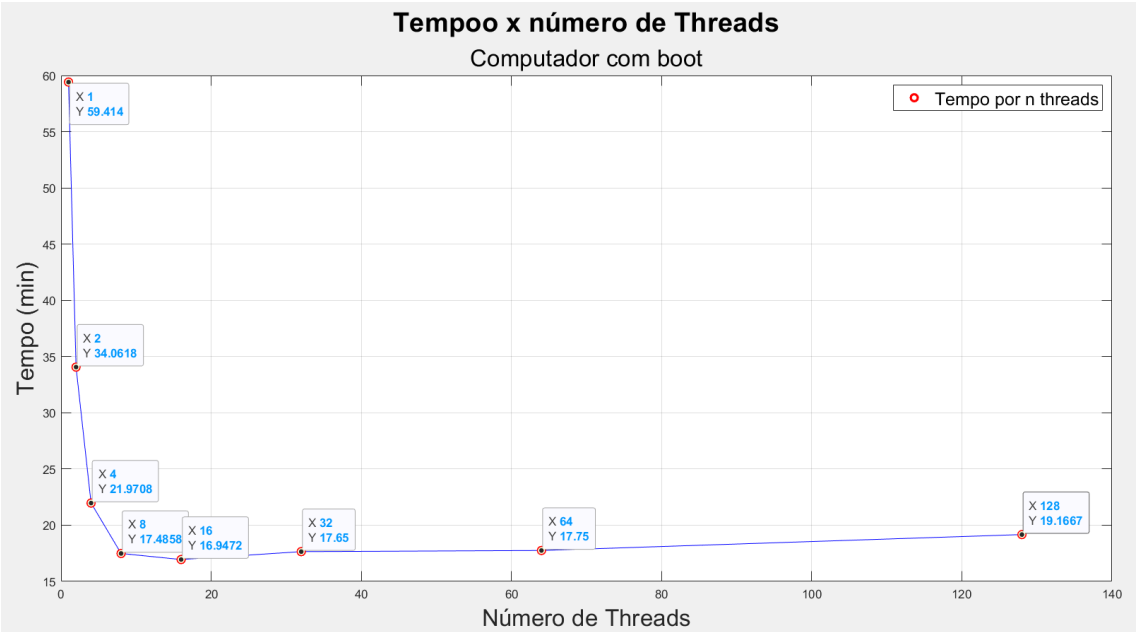


Fonte: Desenvolvido pelo Autor.

Máquina Virtual, de 1 para 8 *threads* o ganho de tempo é exponencial.

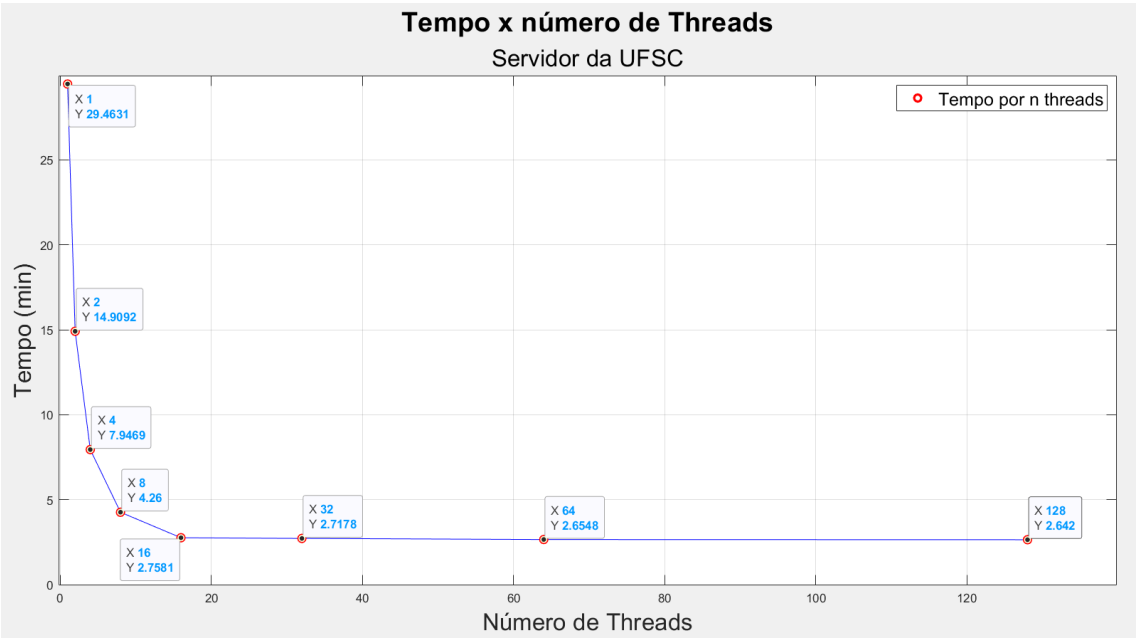
Por fim, quando analisados os resultados obtidos quando realizada a execução no servidor, percebe-se que o tempo de simulação tende a ser inferior comparadas com os ambientes anteriores, isso se da pelo fato das especificações de *hardware* comentados anteriormente. Como visto na Tabela 1 os valores de tempo são consideravelmente menores para o servidor. Já na Figura 4 é possível analisar graficamente o comportamento do tempo de execução no servidor ao passo que o número de *threads* aumenta.

Figura 3 – Gráfico de tempo por número de *threads* para Dual-Boot.



Fonte: Desenvolvido pelo Autor.

Figura 4 – Gráfico de tempo por número de *threads* para Servidor da UFSC.



Fonte: Desenvolvido pelo Autor.

5 CONCLUSÃO

Destarte esse trabalho teve como objetivo demonstrar na prática a teoria de *multithreading* lecionada em sala de aula. Com os testes em diferentes *hardwares* e diferentes quantidades de *threads*, foi possível analisar a relação de núcleos do processador com o número ideal de *threads* que extrai o melhor desempenho possível. Além disso, foi possível compreender solidamente o funcionamento das *threads*, suas regiões críticas e seu modo de otimizar o funcionamento de processos, já que paralelizando a execução, obtém-se um ganho exponencial de tempo, ainda tendo em vista o baixo custo que *threads* implicam no sistema operacional.

REFERÊNCIAS

RÔMULO SILVA DE OLIVEIRA, Et. al. **Sistemas Operacionais**. [*S.l.*]: Bookman, 2010.