PUC CAMPINAS
PONTIFÍCIA UNIVERSIDADE CATÓLICA

Redes de computadores A

Campinas, 2 de abril de 2019

4ª Atividade

NOME:	**RA:**
Ettore Biazon Baccan	16000465
Mateus Henrique Zorzi	16100661
Matheus Martins Pupo	16145559
Murilo Martos Mendonça	16063497
Victor Hugo do Nascimento	16100588

Introdução

Nessa atividade foi implementada uma aplicação cliente-sevidor TCP utilizando threads de maneira à atender vários clientes simultaneamente, a vantagem da utilização de threads é que elas compartilham, por natureza, a mesma região de memória, ou seja, uma variável alterada em uma das threads terá seu valor alterado em todas, o que facilita a sincronização. No entanto, as dificuldades são as chamadas *race conditions*, ou condições de corridas, que pode alterar o resultado da execução dependendo da ordem em que as threads são atendidas pelo processador. Para que isso não ocorra, deve-se tratar com calma e precisão a ordem do atendimento, nessa aplicação utilizamos semáforos, para garantir que a memória seja alterada e lida em uma thread a cada vez.

Passo a passo

Cliente

```
if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0)
{
    perror("Socket()");
    exit(3);
}

/* Estabelece conex�o com o servidor */
if (connect(s, (struct sockaddr *)&server, sizeof(server)) < 0)
{
    perror("Connect()");
    exit(4);
}
```

Criamos o socket do cliente e o conectamos à porta usada pelo servidor.

```c
case 1:
    // Cadastrar mensagem
    strcpy(operacao, "cad");
    if (send(s, operacao, strlen(operacao) + 1, 0) < 0)
    {
        perror("Send()");
        exit(5);
    } //informa ao servidor qual operacao sera feita

    __fpurge(stdin);
    printf("\nDigite o nome: ");
    fgets(nome, sizeof(nome), stdin);
    strtok(nome, "\n"); //tira o \n inserido pelo fgets

    __fpurge(stdin);
    printf("Digite a mensagem: ");
    fgets(mensagem, sizeof(mensagem), stdin);
    strtok(mensagem, "\n"); //tira o \n inserido pelo fgets
    printf("\nNome recebido do cliente: %s\n", nome);
    printf("\nMensagem recebida do cliente: %s\n", mensagem);

    strcpy(envio, nome);
    strcat(envio, "#");
    strcat(envio, mensagem);
    strcat(envio, "$$");
    printf("\nEnvio: %s\n", envio);

    if (send(s, envio, strlen(envio) + 1, 0) < 0)
    {
        perror("Send()");
        exit(5);
    } //envia o nome ao servidor

    if (recv(s, recvbuf, sizeof(recvbuf), 0) < 0)
    {
        perror("Recv()");
        exit(6);
    } //recebe a resposta do servidor
    printf("Servidor: %s\n", recvbuf);

    break;
```

Se o usuário optar por cadastrar uma nova mensagem, enviamos a mensagem “cad” ao servidor, em seguida recebemos o nome e a mensagem que serão cadastrados. Juntamos os dois em uma única string seguindo o seguinte padrão:

Nome#mensagem$$.

Então, enviamos ao servidor a string e esperamos a confirmação de que foi recebida.

```c
case 2:
    // Ler mensagens
    strcpy(operacao, "ler");

    if (send(s, operacao, strlen(operacao) + 1, 0) < 0)
    {
        perror("Send()");
        exit(5);
    } //informa ao servidor qual operacao sera feita

    //salva as mensagens e ja printa
    if (recv(s, recvbuf, sizeof(recvbuf), 0) < 0)
    {
        perror("Recv()");
        exit(6);
    } //recebe o numero de mensagens cadastradas
    //talvez tenhamos que converter quantidade para int (char)
    if (send(s, "OK", 3, 0) < 0)
    {
        perror("Send()");
        exit(5);
    }

    char indice_recebido[2];
    strcpy(indice_recebido, recvbuf);
    quantidade = atoi(indice_recebido);
    printf("\nMensagens cadastradas: %d\n", quantidade);

    for (int i = 0; i < quantidade; i++)
    {
        //recebe o nome e imprime

        if (recv(s, envio, sizeof(envio), 0) < 0)
        {
            perror("Nome()");
            exit(6);
        }
        if (send(s, "OK", 3, 0) < 0)
        {
            perror("Send()");
            exit(5);
        }

        strcpy(nome, strtok(envio, "#"));
        strcpy(mensagem, strtok('\0', "$$"));

        printf("Usuario: %s", nome);
        printf("\t\tMensagem: %s\n", mensagem);
    }
    break;
```

Caso opte por ler, enviamos o comando “ler” ao servidor.

Em seguida, recebemos o número de mensagens cadastradas, avisamos o servidor sobre seu recebimento e o usamos como parâmetro no for e exibimos uma a uma as mensagens na tela. A cada mensagem recebida, enviamos um OK ao servidor para garantir a sincronia.

```c
case 3:
    // Apagar mensagem
    strcpy(operacao, "apa");

    if (send(s, operacao, strlen(operacao) + 1, 0) < 0)
    {
        perror("Send()");
        exit(5);
    } //informa ao servidor qual operacao sera feita

    printf("Digite o nome do usuario que tera a mensagem apagada: ");

    __fpurge(stdin);
    fgets(nome, sizeof(nome), stdin);
    strtok(nome, "\n"); //tira o \n inserido pelo fgets

    if (send(s, nome, strlen(nome) + 1, 0) < 0)
    {
        perror("Send()");
        exit(5);
    } //envia o nome associado a mensagem que sera apagada

    if (recv(s, recvbuf, sizeof(recvbuf), 0) < 0)
    {
        perror("Recv()");
        exit(6);
    } //recebe a resposta do servidor
    char msg_apagadas[2];

    strcpy(msg_apagadas, recvbuf);

    printf("\nMensagens apagadas: %s\n", recvbuf);
    int z;
    z = atoi(msg_apagadas);
    for (int i = 0; i < z; i++)
    {
        //recebe o nome e imprime
        if (recv(s, envio, sizeof(envio), 0) < 0)
        {
            perror("Nome()");
            exit(6);
        }

        strcpy(nome, strtok(envio, "#"));
        strcpy(mensagem, strtok('\0', "$$"));

        printf("Usuario: %s", nome);
        printf("\t\tMensagem: %s\n", mensagem);
    }
    break;
```

Se o usuário optar por apagar, enviamos ao servidor o comando “apa”. Na sequência, enviamos o nome do usuário dono da(s) mensagem(ns).

Recebemos do servidor o número de mensagens apagadas.

Em seguida, utilizando desse valor, recebemos e exibimos na tela as mensagens apagadas.

```
    case 4: //sair
        printf("Obrigado por utilizar a aplicacao\n");

        strcpy(operacao, "out");

        if (send(s, operacao, strlen(operacao) + 1, 0) < 0)
        {
            perror("Send()");
            exit(5);
        } //informa ao servidor qual operacao sera feita

        break;

    default:
        printf("Opcao invalida!\n");
        break;
    }

} while (op != 4);

/* Fecha o socket */
close(s);

printf("Cliente terminou com sucesso.\n");
exit(0);
```

Saíndo do programa, enviamos ao servidor o comando “out”.

No cliente, fechamos o socket, que não será mais utilizado nesse programa.

Servidor

```
if ((s = socket(PF_INET, SOCK_STREAM, 0)) < 0)
{
    perror("Socket()");
    exit(2);
}
```

Criamos o socket do servidor.

```
if (bind(s, (struct sockaddr *)&server, sizeof(server)) < 0)
{
    perror("Bind()");
    exit(3);
}
```

O conectamos à porta recebida no ARGV.

```
if (listen(s, 1) != 0)
{
    perror("Listen()");
    exit(4);
}
printf("*** Servidor Iniciado! ***\n");
```

Esperamos por conexões.

```c
while (1)
{
    namelen = sizeof(client);
    if ((ns = accept(s, (struct sockaddr *)&client, &namelen)) == -1)
    {
        perror("Accept()");
        exit(5);
    }

    if (pthread_create(&thread_id, NULL, servidor, (void *)ns))
    {
        printf("ERRO: impossivel criar uma thread\n");
        exit(-1);
    }
    count_servers++;
    pthread_detach(thread_id);
}
```

Conectamos ao cliente e criamos uma thread que lidará com ele, a thread executará a função “servidor”.

Como não usaremos o comando listen e não há necessidade de armazenar o ID das threads, usamos a mesma variável “thread_id” para todas elas e usamos o comando “pthread_detach” logo que ela foi criada.

```c
void *servidor(int ns)
{
    char sendbuf[101];
    char recvbuf[101];
    char mensagembuf[80]; // mensagem que recebe na função
    char usuariobuf[20];  // usuario que recebe na função
    char mensagem_inteira[101];
    int id_this_thread = count_servers; //no momento que a função é chamada, recebe o count pra saber qual o "id" dela

    printf("\n[%d] Thread criada com sucesso\n", id_this_thread);
```

Armazenamos o número da thread para que pudéssemos controlar com mais facilidade a criação e o encerramento delas.

```c
while (1)
{
    /* Recebe uma mensagem do cliente atrav�s do novo socket conectado */
    memset(mensagembuf, 0, sizeof(mensagembuf));
    memset(usuariobuf, 0, sizeof(usuariobuf));
    memset(recvbuf, 0, sizeof(recvbuf));
    memset(sendbuf, 0, sizeof(sendbuf));

    int retorno;
    retorno = recv(ns, recvbuf, sizeof(recvbuf), 0);    //rececе a mensagem do cliente e verifica o valor de retorno
    if (retorno == -1)
    {
        perror("Recvbuf()");
        close(ns);
        pthread_exit(NULL);
    }
    else if (retorno == 0)
    {
        printf("thread encerrada pois o cliente foi fechado num momento inesperado\n");
        close(ns);
        pthread_exit(NULL);
    }
    retorno = 0;
    printf("\n[%d] Mensagem recebida do cliente: %s\n", id_this_thread, recvbuf);
```

Recebemos uma mensagem do cliente contendo a operação selecionada:

Cad – cadastrar uma nova mensagem.

Ler – Exibir todas as mensagens cadastradas

Apa – Apagar mensagem(ns).

Out – Sair da aplicação.

Fazemos duas verificações no comando recv, uma delas, com o valor –1, caso ocorra algum tipo de erro no envio. A outra, com o valor 0, caso o cliente seja encerrado antes de enviar a mensagem esperada. Além disso, caso ocorra, fechamos apenas a thread, e não o programa todo, como faria o comando “exit” usado nas atividades anteriores.

```c
if (strcmp(recvbuf, "cad") == 0)
{
    // Cadastrar mensagem
    if (indice == 10)
    {
        strcpy(sendbuf, "Numero maximo de mensagens atingido!");
        if (send(ns, sendbuf, strlen(sendbuf) + 1, 0) < 0)
        {
            perror("Send()");
            exit(7);
        }
    }
    else
    {
        retorno = recv(ns, mensagem_inteira, sizeof(recvbuf), 0);    //rececе a mensagem do cliente e verifica o valor de retorno
        printf("retorno: %d\n", retorno);
        if (retorno == -1)
        {
            perror("Recvbuf()");
            close(ns);
            pthread_exit(NULL);
        }
        else if (retorno == 0)
        {
            printf("thread encerrada pois o cliente foi fechado num momento inesperado\n");
            close(ns);
            pthread_exit(NULL);
        }
        else
        {
            printf("\n[%d] Mensagem inteira: %s\n", id_this_thread, mensagem_inteira);
            pthread_mutex_lock(&mutex);
            strcpy(usuarios[indice], strtok(mensagem_inteira, "#"));
            strcpy(mensagens[indice], strtok('\0', "$$"));

            printf("[%d] Usuario cadastrado: %s\n", id_this_thread, usuarios[indice]);
            //strcpy(mensagens[indice], mensagembuf);
            printf("[%d] Mensagem cadastrada: %s\n", id_this_thread, mensagens[indice]);
            printf("[%d] Indice: %d\n", id_this_thread, indice);
            indice++;
            pthread_mutex_unlock(&mutex);
            /* Envia uma mensagem ao cliente através do socket conectado */
            strcpy(sendbuf, "Mensagem cadastrada com sucesso!");
            if (send(ns, sendbuf, strlen(sendbuf) + 1, 0) < 0)
            {
                perror("Send()");
                exit(7);
            }
        }
    }
    printf("[%d] Mensagem enviada ao cliente: %s\n", id_this_thread, sendbuf);
}
```

No caso do cadastro de uma nova mensagem, inicialmente verificamos se o número máximo de mensagens (10) já foi atingido, caso não, recebemos as duas informações em uma única mensagem e a quebramos em duas. (novamente, fazemos a verificação na função “recv”).

Em seguida, armazenamos o nome e a mensagem e informamos o cliente que tudo ocorreu bem.

```
else if (strcmp(recvbuf, "ler") == 0)
{
    // Ler mensagem
    char qtd_msg[2];
    sprintf(qtd_msg, "%d", indice);
    strcpy(sendbuf, qtd_msg);
    printf("[%d] SENDBUF: %s\n", id_this_thread, sendbuf);

    pthread_mutex_lock(&mutex);
    if (send(ns, sendbuf, strlen(sendbuf) + 1, 0) < 0)
    {
        perror("Send()");
        exit(7);
    }
    retorno = recv(ns, mensagem_inteira, sizeof(mensagem_inteira), 0); //rececе a mensagem do cliente e verifica o valor de retorno
    if (retorno == -1)
    {
        perror("Recvbuf()");
        close(ns);
        pthread_exit(NULL);
    }
    else if (retorno == 0)
    {
        printf("thread encerrada pois o cliente foi fechado num momento inesperado\n");
        close(ns);
        pthread_exit(NULL);
    }
    pthread_mutex_unlock(&mutex);
```

No comando ler, enviamos ao cliente o número de mensagens cadastradas e recebemos a sua confirmação (“OK”).

```
pthread_mutex_lock(&mutex);
for (int i = 0; i < indice; i++)
{
    memset(sendbuf, 0, sizeof(sendbuf));
    strcpy(mensagem_inteira, usuarios[i]);
    strcat(mensagem_inteira, "#");
    strcat(mensagem_inteira, mensagens[i]);
    strcat(mensagem_inteira, "$$");
    strcpy(sendbuf, mensagem_inteira);

    if (send(ns, sendbuf, strlen(sendbuf) + 1, 0) < 0)
    {
        perror("Send()");
        exit(7);
    }
    retorno = recv(ns, mensagem_inteira, sizeof(mensagem_inteira), 0); //rececе a mensagem do cliente e verifica o valor de retorno
    if (retorno == -1)
    {
        perror("Recvbuf()");
        close(ns);
        pthread_exit(NULL);
    }
    else if (retorno == 0)
    {
        printf("thread encerrada pois o cliente foi fechado num momento inesperado\n");
        close(ns);
        pthread_exit(NULL);
    }
    //receber msg confirmação do cliente
}
pthread_mutex_unlock(&mutex);
```

Em seguida, enviamos as mensagens, uma a uma, seguindo o padrão já mencionado:

Nome#mensagem$$.

Finalmente, recebemos uma mensagem do cliente informando se ocorreu tudo bem.

```
else if (strcmp(recvbuf, "apa") == 0)
{
    // Apaga mensagem
    char nome[20];
    retorno = recv(ns, recvbuf, sizeof(recvbuf), 0);    //rececе a mensagem do cliente e verifica o valor de retorno
    if (retorno == -1)
    {
        perror("Recvbuf()");
        close(ns);
        pthread_exit(NULL);
    }
    else if (retorno == 0)
    {
        printf("thread encerrada pois o cliente foi fechado num momento inesperado\n");
        close(ns);
        pthread_exit(NULL);
    }
    strcpy(nome, recvbuf);
    strcpy(sendbuf, "Usuario nao encontrado!\n");

    // MOSTRAR MENSAGENS APAGADAS

    int msg_apagadas = 0;
    char k[2];

    pthread_mutex_lock(&mutex);
    for (int i = 0; i < indice; i++)
    {
        printf("[%d] Nome: %d\n", id_this_thread, i);
        if (strcmp(nome, usuarios[i]) == 0)
        {
            msg_apagadas++;
        }
    }
    pthread_mutex_unlock(&mutex);

    sprintf(k, "%d", msg_apagadas);
    strcpy(sendbuf, k);

    printf("[%d] SENDBUF: %s\n", id_this_thread, sendbuf);
    if (send(ns, sendbuf, strlen(sendbuf) + 1, 0) < 0)
    {
        perror("Send()");
        exit(7);
    }
```

Caso o usuário opte por apagar, recebemos do cliente o nome do dono da(s) mensagem(ns).

Procuramos o número de mensagens vinculadas a esse usuário e informamos o cliente.

```c
pthread_mutex_lock(&mutex);
for (int i = 0; i < indice; i++)
{
    if (strcmp(nome, usuarios[i]) == 0)
    {
        memset(sendbuf, 0, sizeof(sendbuf));
        strcpy(mensagem_inteira, usuarios[i]);
        strcat(mensagem_inteira, "#");
        strcat(mensagem_inteira, mensagens[i]);
        strcat(mensagem_inteira, "$$");
        strcpy(sendbuf, mensagem_inteira);

        if (send(ns, sendbuf, strlen(sendbuf) + 1, 0) < 0)
        {
            perror("Send()");
            exit(7);
        }
    }
    //receber msg confirmação do cliente
}
pthread_mutex_unlock(&mutex);

// FIM MOSTRAR MENSAGENS APAGADAS
```

Então, enviamos ao cliente as mensagens (no mesmo padrão já citado).

```c
// FIM MOSTRAR MENSAGENS APAGADAS

pthread_mutex_lock(&mutex);
for (int i = 0; i < indice; i++)
{
    printf("[%d] Nome: %d\n", id_this_thread, i);
    if (strcmp(nome, usuarios[i]) == 0)
    {
        printf("[%d] Nome %d localizado\n", id_this_thread, i);
        for (int j = i; j < indice; j++)
        {
            printf("[%d] Usuario %d recebe usuario %d\n", id_this_thread, j, j + 1);
            strcpy(usuarios[j], usuarios[j + 1]);
            strcpy(mensagens[j], mensagens[j + 1]);
        }
        indice--;
        printf("[%d] Indice: %d\nI: %d\n", id_this_thread, indice, i);
    }
}
pthread_mutex_unlock(&mutex);
```

As reposicionamos nos vetores, para que os espaços vazios sempre fiquem ao final.

```
else if (strcmp(recvbuf, "out") == 0)
{
    close(ns);
    pthread_exit(NULL);
}
```

Caso o cliente opte por sair da aplicação, fechamos o socket com o servidor e encerramos a thread, através do comando “pthread_exit”.

Testes

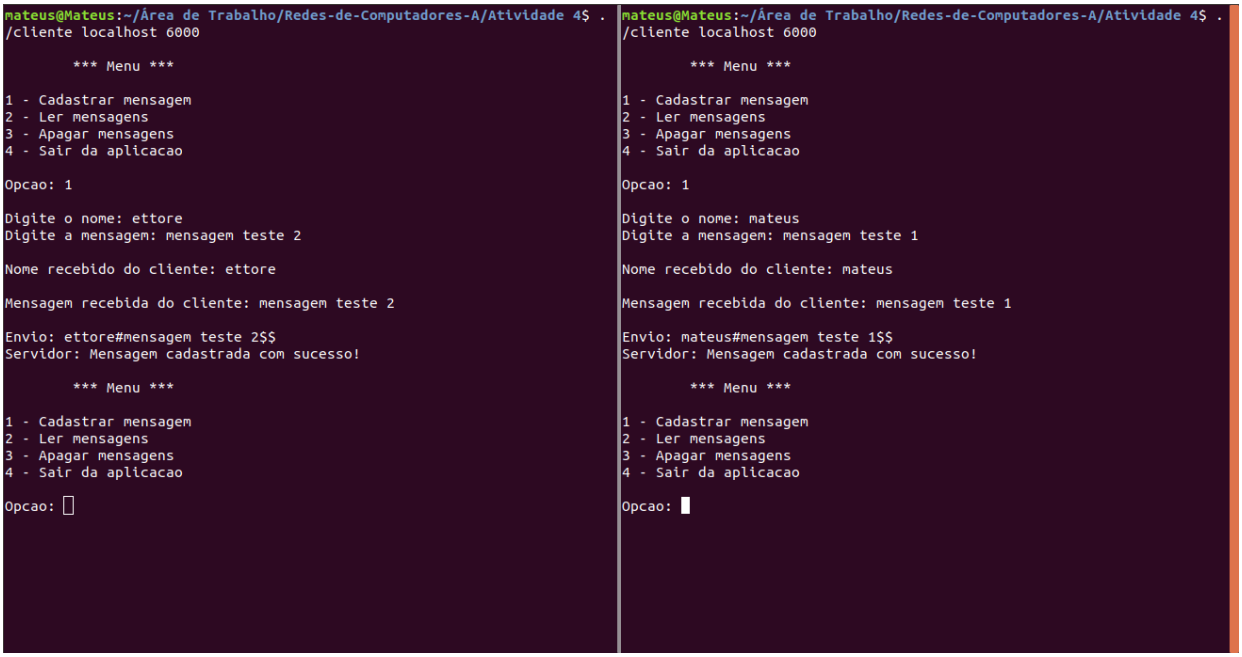

Imagem 1: Cadastramos dois usuários, um em cada servidor, para mostrar as threads sendo utilizadas.

```
mateus@Mateus:~/Área de Trabalho/Redes-de-Computadores-A/Atividade 4$ ./servidor
 6000
*** Servidor Iniciado! ***

[1] Thread criada com sucesso

[1] Mensagem recebida do cliente: cad

[1] Mensagem inteira: mateus#mensagem teste 1$$
[1] Usuario cadastrado: mateus
[1] Mensagem cadastrada: mensagem teste 1
[1] Indice: 0
[1] Mensagem enviada ao cliente: Mensagem cadastrada com sucesso!

[2] Thread criada com sucesso

[2] Mensagem recebida do cliente: cad

[2] Mensagem inteira: ettore#mensagem teste 2$$
[2] Usuario cadastrado: ettore
[2] Mensagem cadastrada: mensagem teste 2
[2] Indice: 1
[2] Mensagem enviada ao cliente: Mensagem cadastrada com sucesso!
```

Imagem 2: Servidor mostrando as mensagens para quando novas threads são criadas e sobre os usuários e mensagens cadastrados.

```
mateus@Mateus:~/Área de Trabalho/Redes-de-Computadores-A/Atividade 4$ ./cliente localhost 6000

        *** Menu ***

1 - Cadastrar mensagem
2 - Ler mensagens
3 - Apagar mensagens
4 - Sair da aplicacao

Opcao: 1

Digite o nome: ettore
Digite a mensagem: mensagem teste 2

Nome recebido do cliente: ettore

Mensagem recebida do cliente: mensagem teste 2

Envio: ettore#mensagem teste 2$$
Servidor: Mensagem cadastrada com sucesso!

        *** Menu ***

1 - Cadastrar mensagem
2 - Ler mensagens
3 - Apagar mensagens
4 - Sair da aplicacao

Opcao:
```

```
1 - Cadastrar mensagem
2 - Ler mensagens
3 - Apagar mensagens
4 - Sair da aplicacao

Opcao: 1

Digite o nome: mateus
Digite a mensagem: mensagem teste 1

Nome recebido do cliente: mateus

Mensagem recebida do cliente: mensagem teste 1

Envio: mateus#mensagem teste 1$$
Servidor: Mensagem cadastrada com sucesso!

        *** Menu ***

1 - Cadastrar mensagem
2 - Ler mensagens
3 - Apagar mensagens
4 - Sair da aplicacao

Opcao: 2

Mensagens cadastradas: 2
Usuario: mateus         Mensagem: mensagem teste 1
Usuario: ettore         Mensagem: mensagem teste 2

        *** Menu ***

1 - Cadastrar mensagem
2 - Ler mensagens
3 - Apagar mensagens
4 - Sair da aplicacao

Opcao:
```

Imagem 3: Após cadastrar os dois usuários em clientes diferentes, utilizamos um deles para ler, mostrando o funcionamento das threads.

```
[1] Mensagem recebida do cliente: ler
[1] SENDBUF: 2
```

Imagem 4: Servidor mostrando qual das threads leu as mensagens, o número à esquerda é o “id” da thread.

```
        *** Menu ***

1 - Cadastrar mensagem
2 - Ler mensagens
3 - Apagar mensagens
4 - Sair da aplicacao

Opcao: 2

Mensagens cadastradas: 5
Usuario: mateus         Mensagem: mensagem teste 1
Usuario: ettore         Mensagem: mensagem teste 2
Usuario: victor         Mensagem: eu amo o nilson
Usuario: matheus                Mensagem: ragnarok
Usuario: murilo         Mensagem: sion

        *** Menu ***

1 - Cadastrar mensagem
2 - Ler mensagens
3 - Apagar mensagens
4 - Sair da aplicacao
```

Imagem 5: Antes de realizar a operação apagar, cadastramos mais usuários para deixar o teste mais realista.

```
Opcao: 2

Mensagens cadastradas: 5
Usuario: mateus         Mensagem: mensagem teste 1
Usuario: ettore         Mensagem: mensagem teste 2
Usuario: victor         Mensagem: eu amo o nilson
Usuario: matheus                Mensagem: ragnarok
Usuario: murilo         Mensagem: sion

        *** Menu ***

1 - Cadastrar mensagem
2 - Ler mensagens
3 - Apagar mensagens
4 - Sair da aplicacao

Opcao: 2

Mensagens cadastradas: 4
Usuario: mateus         Mensagem: mensagem teste 1
Usuario: ettore         Mensagem: mensagem teste 2
Usuario: victor         Mensagem: eu amo o nilson
Usuario: murilo         Mensagem: sion

        *** Menu ***
```

```
Arquivo  Editar  Ver  Pesquisar  Terminal  Ajuda
Usuario: mateus         Mensagem: mensagem teste 1
Usuario: ettore         Mensagem: mensagem teste 2

        *** Menu ***

1 - Cadastrar mensagem
2 - Ler mensagens
3 - Apagar mensagens
4 - Sair da aplicacao

Opcao: 3
Digite o nome do usuario que tera a mensagem apagada: matheus

Mensagens apagadas: 1
Usuario: matheus                Mensagem: ragnarok

        *** Menu ***

1 - Cadastrar mensagem
2 - Ler mensagens
3 - Apagar mensagens
4 - Sair da aplicacao

Opcao:
```

Imagem 6: No cliente à esquerda, lemos os usuários e mensagens primeiro sem realizar a operação apagar, após isso, no cliente à direita realizamos a operação e relemos no cliente à esquerda mostrando que o usuario foi apagado de fato.

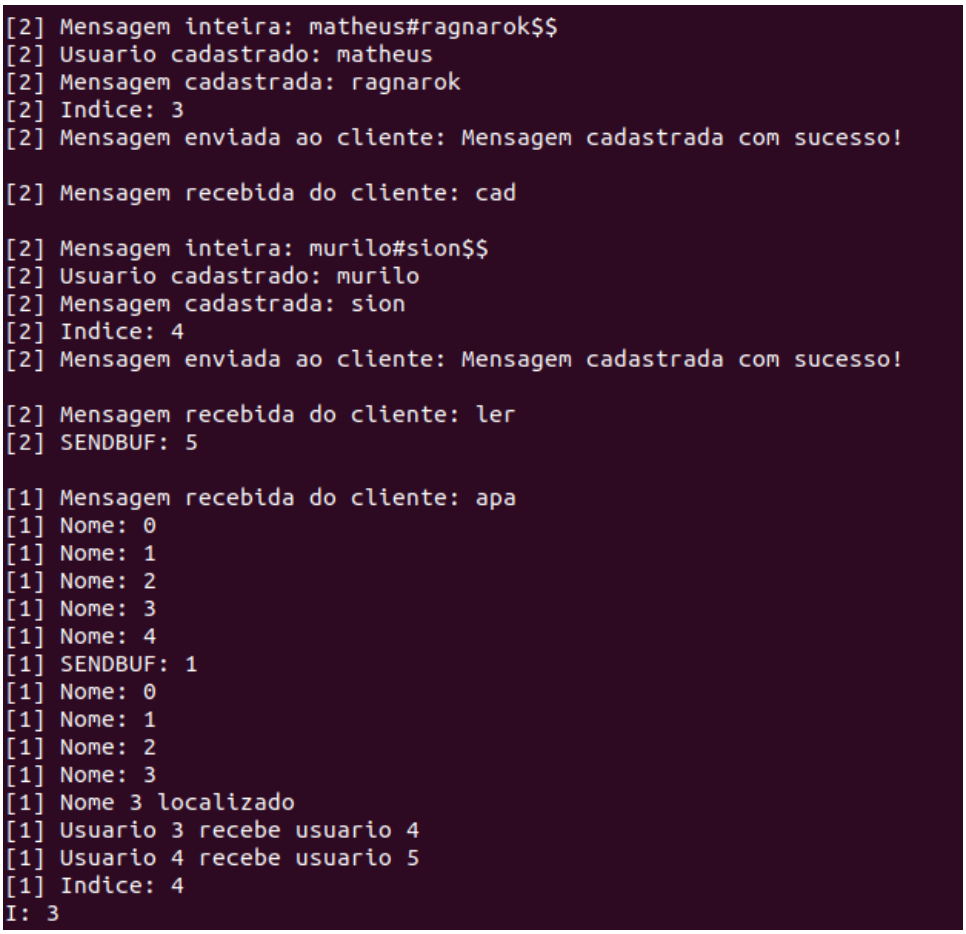

Imagem 7: Mensagem do servidor durante o processo de teste da operação apagar.

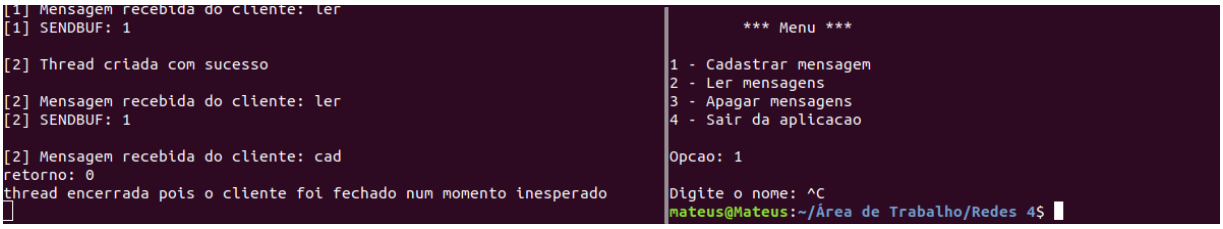

Imagem 8: Quando realizamos uma parada inesperada no cliente, aquela thread é apenas fechada, o socket desconectado e o servidor continua normalmente.

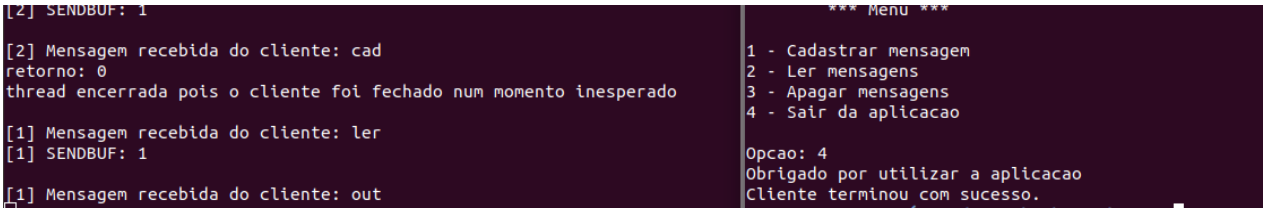

Imagem 9: Ao utilizar a opção 4, o cliente em questão é fechado e o servidor continua operando.

Conclusão

Com a realização da atividade, concluímos que o uso de threads para atender vários clientes simultâneamente, assim como todos os métodos, tem vantagens e desvantagens. Durante a implementação do código, nos deparamos com vários problemas de sincronia, principalmente com valores de variáveis, por isso o uso de semáforos é tão importante, já que as threads compartilham a região de memória.

Além disso, pudemos mais uma vez perceber as diferenças de uso de processos filhos para threads, suas vantagens e desvantagens.

Obtivemos problemas na sincronia, em caso do cliente se desconectar antes do seu término, ou se o terminal for fechado sem o envio do comando de saída da aplicação. Para contornar essa situação, verificamos o retorno da função recv quando isso acontecia (valor 0) e tratamos esse caso, fechamos o socket para evitar desperdício dos recursos do servidor e encerramos a thread que atendia o cliente, já que ela não seria mais usada.

Outra maneira de garantir a sincronia, foi utilizando “confirmações” de envio e recebimento das mensagens, para evitar que duas ou mais mensagens se “aglomerem” em um único recv, por exemplo, garantimos que apenas uma seria recebida por vez ao enviar uma mensagem de confirmação, similar a ideia do three-way handshake.