

**Cauã Ribas, Nilson Andrade**

Universidade do Vale do Itajaí - Univali

Escola Politécnica

Ciência da Computação

(cauaribas, nilson.neto) @edu.univali.br

**Sistemas Operacionais**

M1 - Avaliação 01 – Processos, Threads, Concorrência e Paralelismo

**Felipe Viel**

17/09/2024

## **Resumo:**

Este trabalho aborda a implementação de um servidor que utiliza um pool de threads para processar múltiplas requisições de clientes de forma eficiente. A comunicação é realizada por meio de sockets Unix, onde dois tipos de tarefas são processados: manipulação de strings e operações numéricas. Um sistema de enfileiramento de tarefas distribui as conexões entre as threads, maximizando o uso de recursos de hardware. As threads são pré-criadas, evitando a sobrecarga de criar e destruir novas threads constantemente. Os resultados das simulações mostraram que, embora o aumento no número de threads reduza o tempo de execução, esse ganho não é linear devido ao overhead de gerenciamento de threads. O equilíbrio ideal de desempenho foi atingido com 4 a 8 threads, considerando as limitações do ambiente de teste.

## **Abstract:**

This paper presents the implementation of a server that utilizes a thread pool to efficiently process multiple client requests. Communication is handled through Unix sockets, where two types of tasks are processed: string manipulation and numerical operations. A task queuing system distributes connections among the threads, optimizing the hardware resource usage. Pre-created threads are used to avoid the overhead of continuously creating and destroying new threads. Simulation results showed that increasing the number of threads reduces execution time, though the gain is not linear due to the overhead of thread management. The optimal performance balance was achieved with 4 to 8 threads, considering the limitations of the test environment.

## **1. Introdução:**

Este trabalho tem como objetivo a implementação de um sistema servidor que utiliza um pool de threads para processar requisições de clientes. A abordagem de pool de threads visa otimizar o desempenho do servidor, evitando a sobrecarga de criação e destruição de threads a cada nova conexão. Em vez disso, threads pré-criadas ficam em estado de espera e são ativadas conforme as solicitações chegam, resultando em um sistema mais eficiente e responsivo.

A comunicação entre os clientes e o servidor é realizada através de sockets Unix, e diferentes tipos de threads são responsáveis por processar dois tipos de requisições: operações envolvendo números e manipulação de strings. O uso de pipes nomeados (named pipes) foi adotado para viabilizar a comunicação inter processual (IPC). Além disso, implementou-se um mecanismo de enfileiramento de tarefas, onde as conexões dos clientes são distribuídas entre as threads disponíveis no pool.

O sistema foi projetado para rodar continuamente, sempre pronto para aceitar novas conexões, e um componente fundamental é a utilização de mecanismos de sincronização, como mutexes e variáveis de condição, para garantir o acesso seguro à fila de tarefas compartilhadas entre as threads.

Este relatório detalha a estrutura do sistema, os desafios encontrados durante o desenvolvimento, os resultados obtidos e uma análise de desempenho.

## 2. Enunciado dos Projetos:

O projeto envolve a criação de um servidor local que utiliza comunicação inter processual (IPC) com clientes por meio de sockets Unix, utilizando um pool de threads para processar simultaneamente múltiplas requisições de clientes. O servidor é responsável por escutar dois tipos de solicitações: strings e números. As conexões de string realizam a conversão de texto para letras maiúsculas, enquanto as conexões numéricas processam números inteiros, multiplicando-os por dois.

Para gerenciar essas requisições, o servidor mantém um pool de threads onde cada thread, é responsável por atender as requisições de um cliente. Esse modelo de execução visa reduzir a sobrecarga de criação e destruição de threads, otimizando a alocação de recursos do sistema.

Os clientes se conectam ao servidor utilizando sockets Unix, enviando uma string ou número para processamento, e recebem o resultado de volta.

Além disso, o projeto permite a comunicação simultânea de múltiplos clientes com o servidor, explorando o conceito de concorrência e paralelismo para aumentar a eficiência do sistema.

## 3. Explicação e Contexto da Aplicação para Compreensão do Problema Tratado pela Solução:

Sistemas que lidam com múltiplas conexões simultâneas, como servidores web ou serviços de banco de dados, enfrentam o desafio de gerenciar requisições de clientes de maneira eficiente e escalável. Um dos principais problemas nesses sistemas é a necessidade de processar várias requisições ao mesmo tempo, sem sacrificar o desempenho ou sobrecarregar o sistema com a criação contínua de novos processos ou threads para cada nova solicitação.

O uso de um **pool de threads** surge como uma solução eficiente para esse problema. Com o pool de threads, threads pré-criadas ficam em estado de espera para processar requisições conforme chegam. Isso evita o custo computacional de criar e destruir threads repetidamente, além de reduzir a latência no atendimento das conexões. Ao utilizar threads, o sistema pode aproveitar o paralelismo do hardware, permitindo que várias requisições sejam atendidas simultaneamente.

Neste projeto, o servidor recebe e processa dois tipos de dados: strings e números. O primeiro grupo de threads lida com a manipulação de strings, enquanto outro grupo cuida das operações numéricas. Cada cliente, ao se conectar ao servidor por meio de sockets Unix, é tratado de forma concorrente, com sua solicitação sendo enfileirada para uma thread do pool.

Um detalhe importante é que foram utilizadas **4 threads como padrão** no pool de threads. Esse número foi escolhido devido às **limitações do ambiente Codespace do GitHub**, onde o projeto foi desenvolvido. O Codespace oferece uma infraestrutura virtualizada que impõe restrições de uso de recursos de hardware, como número de núcleos e alocação de memória, o que impediu a implementação de um número maior de threads para

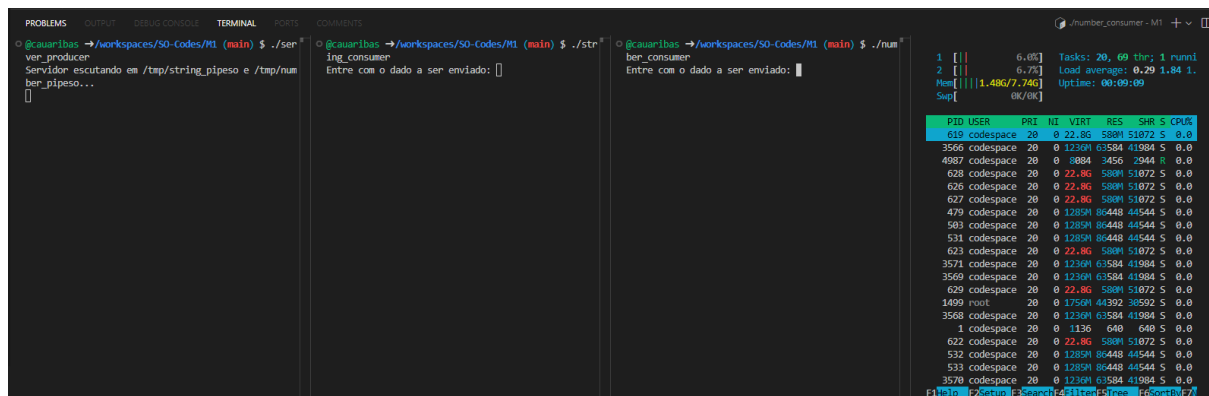
teste. Ainda assim, o uso de 4 threads foi suficiente para garantir a execução paralela das tarefas e demonstrar a funcionalidade proposta.

Essa abordagem resolve problemas de sobrecarga e otimiza o uso dos recursos do sistema, permitindo que múltiplos clientes possam interagir com o servidor de forma eficiente. Além disso, o uso de IPC através de sockets Unix possibilita a comunicação entre processos dentro do mesmo sistema, simulando um ambiente de comunicação em rede.

## 4. Resultados obtidos com as simulações:

### 4.1 Servidor ouvindo os dois sockets e esperando dados com dois clientes conectados:

O servidor está ativo e ouvindo em ambos os sockets (um para strings e outro para números), com dois clientes já conectados.



```
o @cauaribas →/workspaces/SO-Codes/M1 (main) $ ./server_producer
Servidor escutando em /tmp/string_pipeso e /tmp/number_pipeso...

o @cauaribas →/workspaces/SO-Codes/M1 (main) $ ./string_consumer
Entre com o dado a ser enviado: []

o @cauaribas →/workspaces/SO-Codes/M1 (main) $ ./number_consumer
Entre com o dado a ser enviado: []

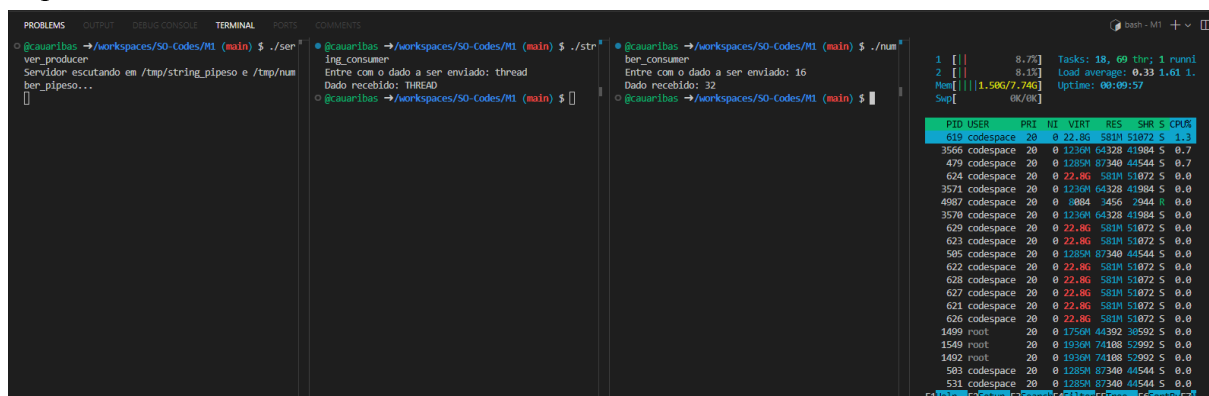
1 6.0% Tasks: 20, 69 thr; 1 runni
2 6.7% Load average: 0.29 1.84 1.
Mem[1.486/7.746] Uptime: 00:09:09
Swp[0/0]

PID USER PRI NI VIRT RES SHR S CPU%
619 codespace 20 0 22.8G 580M 51072 S 0.0
3560 codespace 20 0 1230M 63584 41984 S 0.0
4987 codespace 20 0 8084 3456 2944 S 0.0
628 codespace 20 0 22.8G 580M 51072 S 0.0
626 codespace 20 0 22.8G 580M 51072 S 0.0
627 codespace 20 0 22.8G 580M 51072 S 0.0
479 codespace 20 0 1285M 86448 44544 S 0.0
583 codespace 20 0 1285M 86448 44544 S 0.0
531 codespace 20 0 1285M 86448 44544 S 0.0
623 codespace 20 0 22.8G 580M 51072 S 0.0
3571 codespace 20 0 1230M 63584 41984 S 0.0
3569 codespace 20 0 1230M 63584 41984 S 0.0
629 codespace 20 0 22.8G 580M 51072 S 0.0
1499 root 20 0 1750M 44392 36592 S 0.0
3568 codespace 20 0 1230M 63584 41984 S 0.0
1 codespace 20 0 1136 640 640 S 0.0
622 codespace 20 0 22.8G 580M 51072 S 0.0
532 codespace 20 0 1285M 86448 44544 S 0.0
533 codespace 20 0 1285M 86448 44544 S 0.0
3570 codespace 20 0 1230M 63584 41984 S 0.0
F1=In F2=Setup F3=Search F4=Filter F5=Tree F6=SortB F7=
```

Imagem 1: Servidor ouvindo dois sockets (para strings e números) com dois clientes conectados aguardando para enviar dados.

### 4.2 Dados processados e resposta enviada aos clientes:

O servidor processa os dados recebidos dos dois clientes conectados e envia a resposta de volta.



```
o @cauaribas →/workspaces/SO-Codes/M1 (main) $ ./server_producer
Servidor escutando em /tmp/string_pipeso e /tmp/number_pipeso...

o @cauaribas →/workspaces/SO-Codes/M1 (main) $ ./string_consumer
Entre com o dado a ser enviado: thread
Dado recebido: THREAD

o @cauaribas →/workspaces/SO-Codes/M1 (main) $

o @cauaribas →/workspaces/SO-Codes/M1 (main) $ ./number_consumer
Entre com o dado a ser enviado: 16
Dado recebido: 32

o @cauaribas →/workspaces/SO-Codes/M1 (main) $

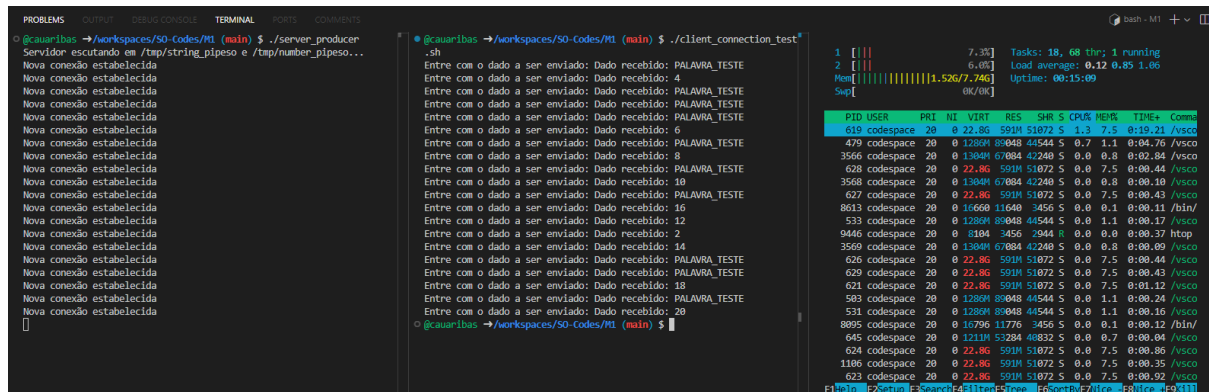
1 8.7% Tasks: 18, 69 thr; 1 runni
2 8.1% Load average: 0.33 1.61 1.
Mem[1.506/7.746] Uptime: 00:09:57
Swp[0/0]

PID USER PRI NI VIRT RES SHR S CPU%
619 codespace 20 0 22.8G 581M 51072 S 1.3
3566 codespace 20 0 1230M 64328 41984 S 0.7
479 codespace 20 0 1285M 87340 44544 S 0.7
624 codespace 20 0 22.8G 581M 51072 S 0.0
3571 codespace 20 0 1230M 64328 41984 S 0.0
4987 codespace 20 0 8084 3456 2944 R 0.0
628 codespace 20 0 22.8G 581M 51072 S 0.0
629 codespace 20 0 22.8G 581M 51072 S 0.0
623 codespace 20 0 22.8G 581M 51072 S 0.0
585 codespace 20 0 1285M 87340 44544 S 0.0
622 codespace 20 0 22.8G 581M 51072 S 0.0
627 codespace 20 0 22.8G 581M 51072 S 0.0
621 codespace 20 0 22.8G 581M 51072 S 0.0
626 codespace 20 0 22.8G 581M 51072 S 0.0
1499 root 20 0 1750M 44392 36592 S 0.0
1549 root 20 0 1930M 74188 52992 S 0.0
1492 root 20 0 1836M 74188 52992 S 0.0
583 codespace 20 0 1285M 87340 44544 S 0.0
531 codespace 20 0 1285M 87340 44544 S 0.0
F1=In F2=Setup F3=Search F4=Filter F5=Tree F6=SortB F7=
```

Imagem 2: Servidor ouvindo dois sockets (para strings e números) com dois clientes conectados aguardando para enviar dados.

## 4.3 Servidor recebendo múltiplas conexões ao mesmo tempo e processando dados:

O servidor recebe múltiplas conexões simultâneas (de mais de dois clientes), processa os dados e os envia de volta.



The screenshot shows a terminal window with three panes. The left pane shows a server process running `./server_producer` which prints "Servidor escutando em /tmp/string\_pipeso e /tmp/number\_pipeso..." and "Nova conexão estabelecida" multiple times. The middle pane shows a client connection test running `./client_connection_test` which prints "Entre com o dado a ser enviado: Dado recebido: PALAVRA\_TESTE" and "Dado recebido: 4" multiple times. The right pane shows a system monitor window with a table of running processes.

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Comm
610	codespace	20	0	22.8G	591M	51872	S	1.3	7.5	0:10.21	/vsco
479	codespace	20	0	128M	89048	44544	S	0.7	1.1	0:04.76	/vsco
3566	codespace	20	0	130M	67884	42240	S	0.0	0.8	0:02.84	/vsco
628	codespace	20	0	22.8G	591M	51872	S	0.0	7.5	0:00.44	/vsco
3568	codespace	20	0	130M	67884	42240	S	0.0	0.8	0:00.10	/vsco
627	codespace	20	0	22.8G	591M	51872	S	0.0	7.5	0:00.43	/vsco
8613	codespace	20	0	11668	11640	3456	S	0.0	0.1	0:00.11	/bin/
533	codespace	20	0	128M	89048	44544	S	0.0	1.1	0:00.17	/vsco
9446	codespace	20	0	8104	3456	2944	R	0.0	0.0	0:00.37	htop
3569	codespace	20	0	130M	67884	42240	S	0.0	0.8	0:00.09	/vsco
626	codespace	20	0	22.8G	591M	51872	S	0.0	7.5	0:00.44	/vsco
629	codespace	20	0	22.8G	591M	51872	S	0.0	7.5	0:00.43	/vsco
621	codespace	20	0	22.8G	591M	51872	S	0.0	7.5	0:01.12	/vsco
583	codespace	20	0	128M	89048	44544	S	0.0	1.1	0:00.24	/vsco
531	codespace	20	0	128M	89048	44544	S	0.0	1.1	0:00.16	/vsco
8895	codespace	20	0	10796	11776	3456	S	0.0	0.1	0:00.12	/bin/
645	codespace	20	0	1211M	53284	46832	S	0.0	0.7	0:00.04	/vsco
624	codespace	20	0	22.8G	591M	51872	S	0.0	7.5	0:00.86	/vsco
1186	codespace	20	0	22.8G	591M	51872	S	0.0	7.5	0:00.35	/vsco
623	codespace	20	0	22.8G	591M	51872	S	0.0	7.5	0:00.92	/vsco

Imagem 3: Servidor ouvindo dois sockets (para strings e números) com dois clientes conectados aguardando para enviar dados.

## 5. Códigos Importantes da Implementação:

### 5.1 Função executeTask:

Esta função executa a função associada a uma tarefa, passando o descritor de socket do cliente como argumento. Dependendo da função de tarefa, ela pode manipular strings ou números.



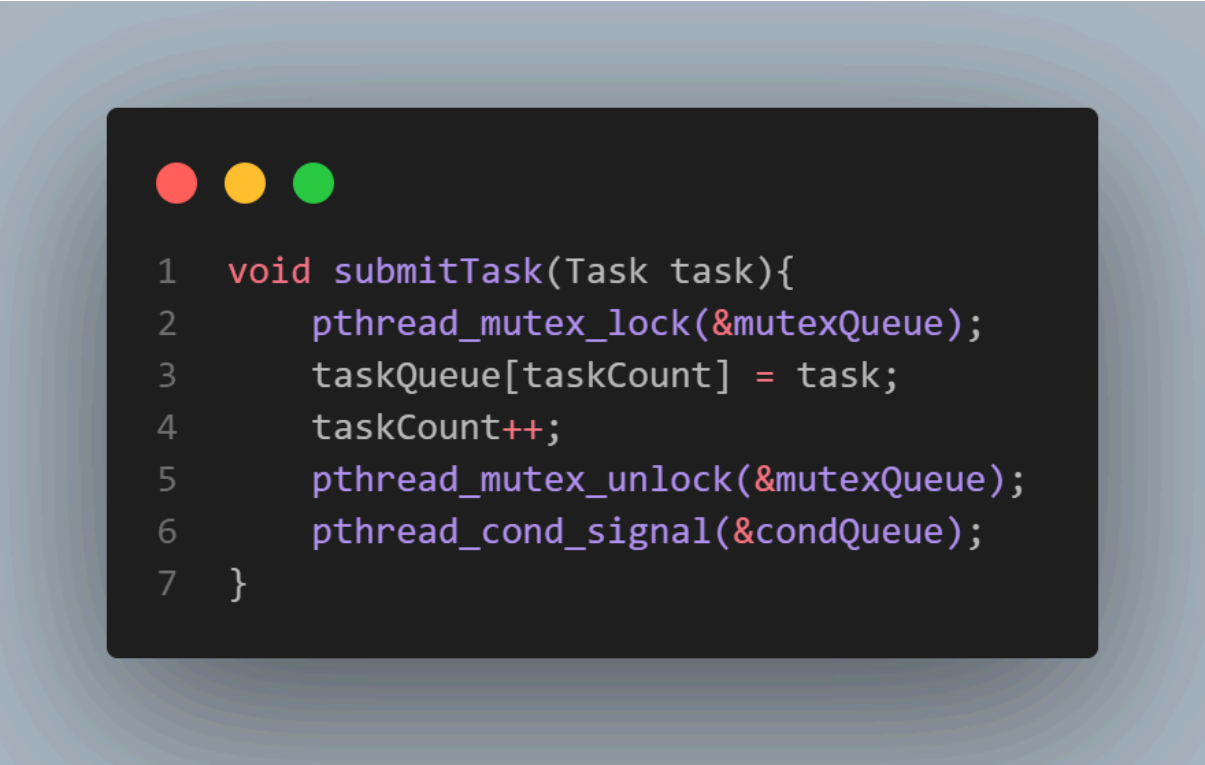
The screenshot shows a code editor with a dark background. It contains a C function named `executeTask` that takes a `Task*` pointer as an argument and calls `task->taskFunction(task->client_socket);` to execute the task function associated with the client socket.

```
1 void executeTask(Task* task){
2     task->taskFunction(task->client_socket); // Executa a função de tarefa associada
3 }
```

Imagem 4: Código referente a execução da função associada a uma tarefa.

## 5.2 Função submitTask:

Enfileira uma nova tarefa na fila de tarefas compartilhada entre as threads. A função usa um mutex para garantir que a fila seja acessada de forma segura por múltiplas threads, e depois sinaliza para as threads que há uma nova tarefa disponível.




```
1 void submitTask(Task task){
2     pthread_mutex_lock(&mutexQueue);
3     taskQueue[taskCount] = task;
4     taskCount++;
5     pthread_mutex_unlock(&mutexQueue);
6     pthread_cond_signal(&condQueue);
7 }
```

Imagem 5: Código referente ao processo de enfileirar uma nova tarefa na fila de tarefas.

### 5.3 Função startThread:

Esta função define o comportamento de cada thread no pool. As threads ficam esperando novas tarefas e, quando uma tarefa é adicionada à fila, a thread a remove e a executa. A função utiliza um mutex para garantir acesso exclusivo à fila de tarefas e uma variável de condição para esperar até que novas tarefas estejam disponíveis.

A screenshot of a code editor with a dark background and light-colored text. The editor has three colored window control buttons (red, yellow, green) in the top-left corner. The code is a C++ function named startThread, which is a void\* returning function that takes a void\* argument. It contains a while(1) loop that runs indefinitely. Inside the loop, it declares a Task pointer. It then locks a mutex (mutexQueue) and enters another while loop that waits until taskCount is not equal to 0. Once the condition is met, it unlocks the mutex. It then removes the first task from the taskQueue, shifts the remaining tasks one position to the left, and decrements taskCount. Finally, it calls executeTask on the removed task.

```
1 void* startThread(void* args){
2     while(1){
3         Task task;
4
5         pthread_mutex_lock(&mutexQueue);
6         while(taskCount == 0){ // Se não há tarefas, espera até ser sinalizado
7             pthread_cond_wait(&condQueue, &mutexQueue);
8         }
9
10        // Retira a primeira tarefa da fila
11        task = taskQueue[0];
12        for(int i = 0; i < taskCount - 1; i++){
13            taskQueue[i] = taskQueue[i + 1];
14        }
15        taskCount--;
16        pthread_mutex_unlock(&mutexQueue);
17
18        executeTask(&task); // Executa a tarefa removida da fila
19    }
20 }
```

Imagem 6: Código referente ao comportamento das tarefas dentro da pool de threads.

## 5.4 Função startServer:

Esta função cria e inicializa um socket Unix para comunicação inter processual (IPC). Ele vincula o socket a um caminho específico no sistema de arquivos e coloca o socket em modo de escuta, pronto para aceitar conexões.

A screenshot of a code editor with a dark background and light-colored text. The code is a C function named startServer. It takes a const char\* sockpath as an argument. The function creates a Unix socket, binds it to the specified path, and puts it in listen mode. It includes error handling with perror and return values. The code is numbered from 1 to 34.

```
1  int startServer(const char* sockpath){
2      int server_socket;
3      struct sockaddr_un local;
4
5      // Criação do socket UNIX
6      server_socket = socket(AF_UNIX, SOCK_STREAM, 0);
7      if (server_socket < 0) {
8          perror("Falha ao criar socket");
9          return 1;
10     }
11
12     // Bind socket no endereço local
13     memset(&local, 0, sizeof(local));
14     local.sun_family = AF_UNIX;
15     strncpy(local.sun_path, sockpath, sizeof(local.sun_path) - 1); // Define o caminho do socket
16     unlink(local.sun_path); // Remove qualquer socket anterior com o mesmo caminho
17     int len = strlen(local.sun_path) + sizeof(local.sun_family);
18
19     // Vincula o socket ao caminho especificado
20     if (bind(server_socket, (struct sockaddr*)&local, len) < 0) {
21         perror("Falha ao fazer bind no socket");
22         close(server_socket);
23         return 1;
24     }
25
26     // Coloca o socket em modo de escuta por conexões
27     if (listen(server_socket, 5) < 0) {
28         perror("Erro ao escutar socket");
29         close(server_socket);
30         return 1;
31     }
32
33     return server_socket;
34 }
```


Imagem 7: Código referente a criação e inicialização do socket Unix para o IPC.



## 5.5 Função `handleStringConnection` e `handleNumberConnection`:

### 5.5.1 `handleStringConnection`:

Esta função lida com conexões de strings. Ela lê uma string do cliente, converte para maiúsculas e envia de volta ao cliente antes de fechar o socket.




```
1 void handleStringConnection(int client_socket){
2     char buffer[1024];
3
4     // Ler dados do cliente
5     if (read(client_socket, buffer, sizeof(buffer)) < 0) {
6         perror("Erro ao ler do socket");
7         close(client_socket);
8         return;
9     }
10
11    // Processa a string recebida e converte para maiúsculas
12    for (int i = 0; i < strlen(buffer); i++){
13        buffer[i] = toupper(buffer[i]);
14    }
15
16    // Enviar dados de volta ao cliente
17    write(client_socket, buffer, strlen(buffer) + 1);
18    close(client_socket);
19 }
```

Imagem 8: Código referente ao tratamento de conexões de strings.

### 5.5.2 handleNumberConnection:

Esta função lida com conexões de números. Ela lê um número enviado pelo cliente, o multiplica por 2 e envia o resultado de volta ao cliente.




```
1 void handleNumberConnection(int client_socket){
2     char buffer[1024];
3
4     // Ler dados do cliente
5     if (read(client_socket, buffer, sizeof(buffer)) < 0) {
6         perror("Erro ao ler do socket");
7         close(client_socket);
8         return;
9     }
10
11     // Processa o numero recebido e multiplica por 2
12     int number = atoi(buffer);
13     number *= 2;
14
15     snprintf(buffer, sizeof(buffer), "%d", number);
16
17     // Enviar dados de volta ao cliente
18     write(client_socket, buffer, strlen(buffer) + 1);
19     close(client_socket);
20 }
```

Imagem 9: Código referente ao tratamento de conexões de números.

## 5.6 Função acceptConnection:

Esta função aceita conexões de clientes. Após aceitar a conexão, ela cria uma nova tarefa e a submete ao pool de threads. A tarefa utiliza a função apropriada (string ou número) para processar a conexão.



```
1 void acceptConnection(int socket, void (*handler)(int)){
2     struct sockaddr_un remote;
3     socklen_t len = sizeof(remote);
4     int client_socket = accept(socket, (struct sockaddr*)&remote, &len);
5     if (client_socket >= 0) {
6         // printf("Nova conexão recebida\n");
7
8         // Cria uma nova tarefa e a submete para o pool de threads
9         Task t = {
10             .taskFunction = handler,
11             .client_socket = client_socket
12         };
13
14         submitTask(t);
15     }
16 }
```

Imagem 10: Código responsável por aceitar conexões de clientes e delegar o processamento ao pool de threads.

## 5.7 Função main:

O **main** configura o pool de threads, cria os sockets para lidar com strings e números e entra em um loop infinito, utilizando **select** para monitorar os sockets. Dependendo de qual socket recebe uma conexão, ele chama a função apropriada (**handleStringConnection** ou **handleNumberConnection**).

```
1 // Define variáveis para o select e determina o maior descritor de arquivo
2 fd_set read_fds;
3 int max_fd = (string_socket > number_socket) ? string_socket : number_socket;
4
5 while(1){
6     FD_ZERO(&read_fds); // Limpa o conjunto de descritores
7     FD_SET(string_socket, &read_fds); // Adiciona string_socket ao conjunto
8     FD_SET(number_socket, &read_fds); // Adiciona number_socket ao conjunto
9
10    // Espera por atividade nos sockets com select
11    int activity = select(max_fd + 1, &read_fds, NULL, NULL, NULL);
12    if (activity < 0) {
13        perror("Erro no select");
14        continue;
15    }
16    // Verifica se há conexão no socket de strings
17    if(FD_ISSET(string_socket, &read_fds)){
18        acceptConnection(string_socket, handleStringConnection);
19    }
20    // Verifica se há conexão no socket de números
21    if(FD_ISSET(number_socket, &read_fds)){
22        acceptConnection(number_socket, handleNumberConnection);
23    }
24 }
```

Imagem 11: Código referente ao main, que configura o pool de threads e os sockets para strings e números. O loop infinito monitora as conexões usando select.

## 6. Resultados obtidos com a implementação (tabelas, gráficos e etc):

Resultados (em segundos)	2 Threads	4 Threads	8 Threads	16 Threads
1000 Tasks	2.1915s	2.1711s	2.1379s	2.1186s
5000 Tasks	10.8673s	10.6909s	10.6614s	10.6497s
1000 Tasks	21.5474s	21.5001s	21.3483s	21.1439s

## 6.1 Teste com 1000 tarefas:

No teste com 1000 tarefas, podemos observar uma pequena redução no tempo de execução conforme o número de threads aumenta. A diferença entre o tempo para 2 threads e 16 threads é de apenas 0.0729 segundos. O fato de o tempo não diminuir de forma significativa com o aumento do número de threads pode ser explicado pela natureza do overhead de gerenciamento de threads e pelo limite de tarefas simultâneas que o hardware é capaz de processar. Além disso, para um número relativamente pequeno de tarefas (1000), o ganho de paralelismo já é explorado eficientemente mesmo com poucas threads.

Para um conjunto pequeno de 1000 tarefas, o aumento do número de threads proporciona um ganho mínimo de desempenho, pois a sobrecarga de gerenciamento de threads já começa a impactar o tempo total de execução.

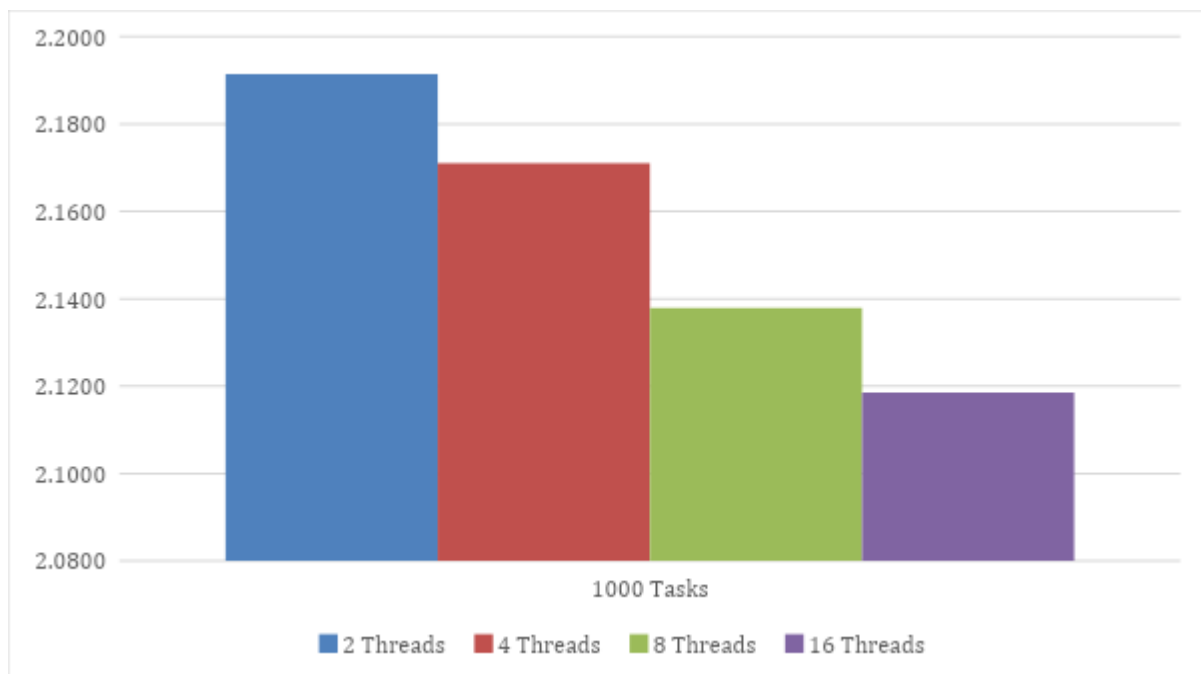


Imagem 12: Gráfico mostrando o desempenho com 1000 tarefas.

## 6.2 Teste com 5000 tarefas:

Com 5000 tarefas, a tendência de redução do tempo com o aumento das threads é menos perceptível, com apenas uma diminuição de cerca de 0.2176 segundos de 2 threads para 16 threads. Esse comportamento sugere que o sistema está começando a encontrar um equilíbrio entre a carga de trabalho e o paralelismo disponível no ambiente de execução. Novamente, o overhead de gerenciamento de muitas threads começa a afetar os ganhos de desempenho, tornando a diferença marginal.

Com 5000 tarefas, há uma ligeira vantagem em aumentar o número de threads, porém o ganho de desempenho não é tão significativo quanto esperado. A sobrecarga adicional no gerenciamento de threads provavelmente está afetando a escalabilidade.

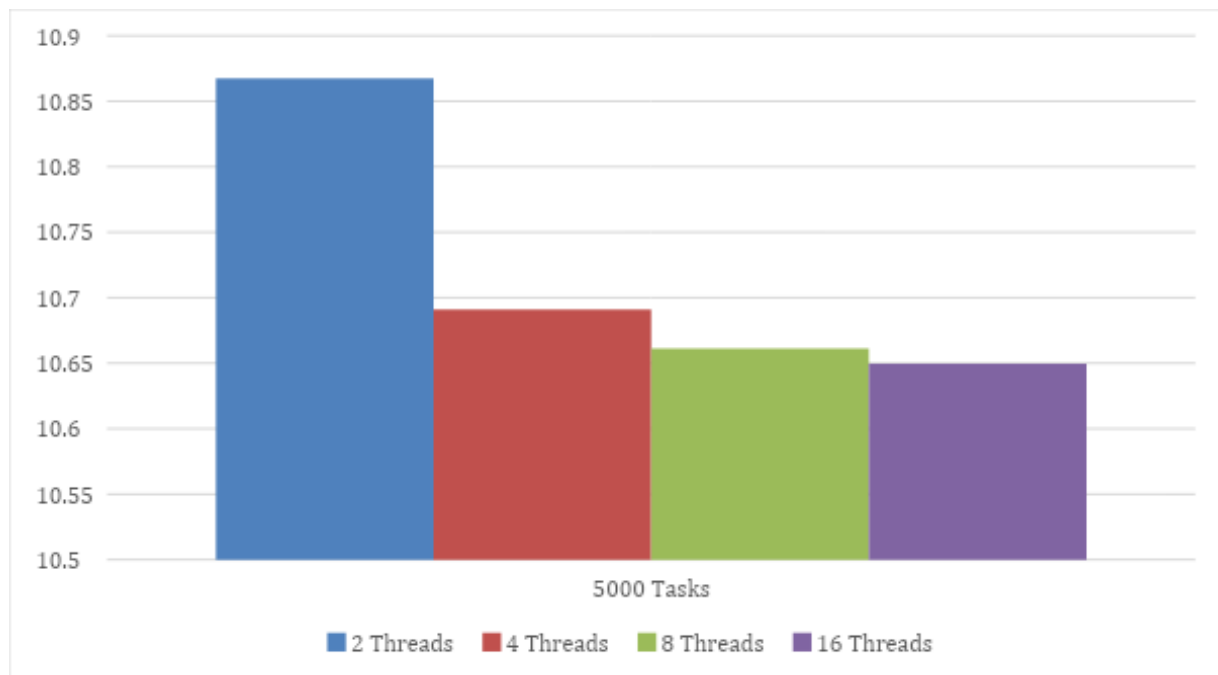


Imagem 13: Gráfico mostrando o desempenho com 5000 tarefas.

### 6.3 Teste com 10000 tarefas:

No teste com 10.000 tarefas, o ganho de desempenho é mais notável, com uma diferença de cerca de 0.4035 segundos entre 2 threads e 16 threads. Mesmo com um número maior de tarefas, o tempo de execução total diminui à medida que o número de threads aumenta, mas de forma cada vez mais limitada. Isso sugere que o ambiente de hardware (com um número limitado de núcleos) está atingindo seu limite de capacidade de paralelismo.

Para 10.000 tarefas, o aumento do número de threads ainda resulta em uma melhoria no tempo de execução, mas com um benefício cada vez menor, devido à sobrecarga no gerenciamento das threads e ao limite físico de paralelismo que o hardware pode sustentar.

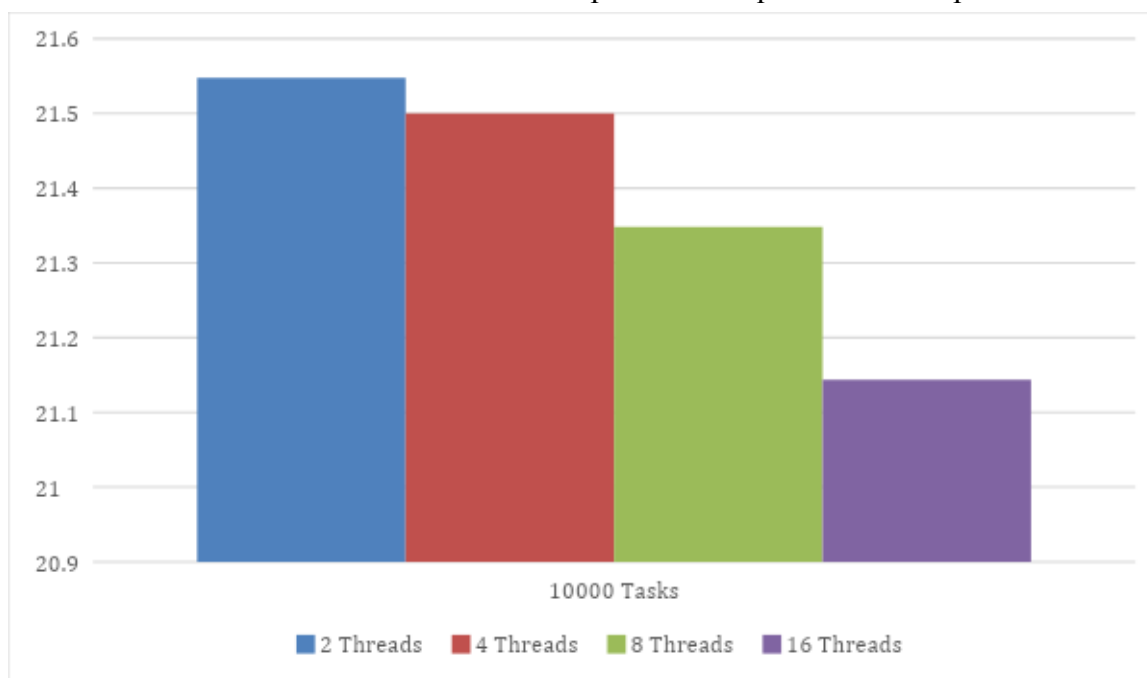


Imagem 14: Gráfico mostrando o desempenho com 10000 tarefas.

## 7. Análise e discussão sobre os resultados finais:

Durante a simulação da aplicação. Os resultados mostraram que, de maneira geral, o tempo de execução real diminui à medida que o número de threads aumenta. No entanto, o comportamento não segue uma redução proporcional linear, o que é esperado em sistemas que utilizam concorrência e paralelismo. Mas por que tempos maiores com menos threads?

Quando utilizamos um número menor de threads (como 2 ou 4 threads), as tarefas são processadas de forma mais sequencial, o que significa que o servidor consegue processar apenas uma quantidade limitada de tarefas ao mesmo tempo. Isso resulta em tempos de execução maiores porque o paralelismo oferecido pelo sistema não está sendo totalmente aproveitado. Com menos threads, há uma maior quantidade de tarefas em espera, aumentando o tempo total de resposta.

No entanto, o aumento no número de threads, até certo ponto, reduz o tempo de execução real. Quando o número de threads é ampliado para 8 ou 16, observamos uma leve diminuição no tempo, mas esses ganhos não são proporcionais ao número de threads

adicionadas. Isso se deve à sobrecarga associada ao gerenciamento de um número elevado de threads. Quanto mais threads são criadas além da capacidade do hardware, mais o sistema precisa gastar tempo na sincronização entre threads e na alternância de contexto entre elas, o que pode aumentar o tempo total.

Portanto, utilizar poucas threads resulta em tempos maiores devido à falta de paralelismo suficiente, enquanto o uso de um número excessivo de threads também gera maior sobrecarga, afetando a eficiência. Com base nos dados obtidos, o equilíbrio ideal para esse tipo de tarefa, nesse ambiente específico, parece estar em torno de 4 a 8 threads., que aproveitam bem o paralelismo sem gerar uma sobrecarga significativa no ambiente de execução limitado.

## 8. Referências:

CODEVAULT. (2021) “Thread Pools in C (using the PTHREAD API)”. Disponível em:

▶ Thread Pools in C (using the PTHREAD API) , acesso em: 16 set. 2024.

CODEVAULT. (2021) “Thread Pools with function pointers in C”. Disponível em:

▶ Thread Pools with function pointers in C , acesso em: 16 set. 2024.

JACOB SORBER. (2019) “How to write a multithreaded server in C (threads, sockets)”.

Disponível em: ▶ How to write a multithreaded server in C (threads, sockets) , acesso em: 16 set. 2024.

JACOB SORBER. (2019) “Multithreaded Server Part 2: Thread Pools”. Disponível em:

▶ Multithreaded Server Part 2: Thread Pools , acesso em: 16 set. 2024.

JACOB SORBER. (2019) “How to write a multithreaded webserver using condition variables (Part 3)”. Disponível em:

▶ How to write a multithreaded webserver using condition variables (Part 3) , acesso em: 16 set. 2024.

JACOB SORBER. (2019) “How one thread listens to many sockets with select in C”.

Disponível em: ▶ How one thread listens to many sockets with select in C. , acesso em: 16 set. 2024.

PANDEY, DURGESH. Inter Process Communication (IPC). Disponível em: [Inter Process Communication \(IPC\) - GeeksforGeeks](#), acesso em: 16 set. 2024.

OPENGROUP. Threads. Disponível em: [Threads \(opengroup.org\)](#), acesso em: 16 set. 2024.