Processos, Threads, Concorrência e Paralelismo

Eduarda Immianowsky[1]
Filipi da Costa[2]

Resumo: este artigo apresenta a resolução da primeira avaliação da disciplina de Sistemas Operacionais que tem por objetivo consolidar o aprendizado sobre conceitos de IPC, threads, concorrência e paralelismo.
Palavras Chaves: Threads; Servidor; Cliente; Processo; Sistema.

1. Introdução

É necessário que os sistemas computacionais apresentem a capacidade de atender múltiplas requisições simultaneamente para garantia de um bom desempenho e escalabilidade. O uso de threads é o que permite uma execução paralela de tarefas dentro de um mesmo processo. Porém, criar uma nova thread para cada requisição se torna custoso em termos de recursos e desempenho, principalmente com um número alto de requisições. Para solucionar esse problema, a utilização de um pool de threads se torna mais eficaz. Isto é, um conjunto de threads pré-criadas que podem ser reutilizadas para processar requisições, tornando menos provável o overhead de criação e destruição de threads. Isso resulta em um sistema mais funcional e eficiente.

O objetivo deste projeto é a implementação de um servidor que utiliza dois sockets ou pipes nomeados para comunicação com clientes. Ele opera com dois tipos de conexões: uma para strings e outra para números. O servidor utiliza um pool de threads para gerenciar múltiplos clientes de maneira coerente.

2. Inicialização

Primeiramente, são criados dois sockets, um para strings e outro para números, e um pool de threads com duas threads. O servidor então utiliza a função select para monitorar conexões nos dois sockets simultaneamente, aguardando até que um deles tenha uma conexão

[1] UNIVALI – Universidade do Vale do Itajaí
eduardaimmianowsky@edu.univali.br

[2] UNIVALI – Universidade do Vale do Itajaí
filipi_costa@edu.univali.br

pendente. Quando uma conexão é detectada, o servidor a aceita, estabelecendo uma conexão com o cliente em um dos pipes.

No processamento de dados, o servidor trata as solicitações de forma diferente dependendo do tipo de conexão. Se o cliente está conectado ao pipe de strings, o servidor converte os dados recebidos para maiúsculas. Se o cliente está conectado ao pipe numérico, o servidor adiciona 10 aos dados recebidos. Após o processamento, o servidor envia os dados de volta ao cliente.

A utilização de threads separadas permite que o servidor atenda múltiplos clientes ao mesmo tempo, aproveitando a concorrência para gerenciar várias conexões simultaneamente. Este código utiliza concorrência, sockets e comunicação entre processos em um ambiente Linux, permitindo ao servidor gerenciar múltiplos clientes de maneira eficiente.

3. Criação dos sockets

Os sockets locais são criados com as funções socket e bind. O servidor utiliza dois caminhos diferentes:

- /tmp/pipestr para strings;
- /tmp/pipenum para números.

Esses sockets usam o tipo AF_UNIX para se comunicar com processos locais no mesmo sistema operacional.

O servidor também usa a função select para aguardar ambos os sockets ao mesmo tempo. Ela identifica qual pipe está pronto para aceitar uma nova conexão.

Basicamente o servidor chama select e aguarda até que um dos sockets tenha uma conexão pendente. Se um cliente tentar se conectar ao socket de strings (/tmp/pipestr), o servidor o aceita e processa como uma conexão de string. Por outro lado, se um cliente tentar se conectar ao socket numérico (/tmp/pipenum), o servidor o aceitará e processará como uma conexão numérica.

Em questão de aceitação de novas conexões, quando a função select detecta uma conexão, a função accept retorna um novo socket que representa a conexão específica com o cliente.

4. Struct cliente_data_t

Essa struct armazena duas informações:

- newsockfd: o socket do cliente;

- pipe_type: indica qual tipo de pipe o cliente está usando (string ou numérica).

5. Processamento de dados do cliente

Após o servidor aceitar a conexão, é necessário processar o que foi enviado pelo cliente. Isso é feito pela função process_client, que é adicionada ao pool de threads para ser executada usando concorrência.

Dentro da função process_client, o servidor lê os dados usando read. O processamento dos dados varia conforme o tipo de pipe: se o cliente estiver conectado ao de strings, o buffer é transformado em maiúsculas, mas se o cliente estiver conectado ao numérico, o buffer é incrementado com uma constante (10). Por fim, o servidor escreve os resultados usando write.

6. Pool de threads

O servidor utiliza uma biblioteca de pool de threads chamada thpool, que é responsável por gerenciar as threads para executar as funções usando concorrência.

O servidor inicializa o pool de threads com duas threads usando a função thpool_init(2). Quando um novo cliente se conecta, a função process_client é adicionada ao pool, sendo executada em uma thread separada das outras. Isso permite que o servidor aceite novas conexões enquanto processa as conexões já existentes.

7. Conexão do client ao server

O código estabelece uma conexão ente cliente e servidor utilizando sockets Unix. Primeiro, prepara a estrutura sockaddr_un para definir o endereço do servidor, configurando o tipo de socket e o caminho com base no tipo de pipe (string ou numérica). A conexão é então feita usando a função connect.

Se a conexão for bem-sucedida, o cliente solicita um dado ao usuário, envia-o para o servidor com a função write, e após isso lê a resposta do servidor com a função read.

8. Encerramento

Ao encerrar, a conexão com o cliente é fechada (representada por newsockfd) e a memória alocada para o cliente (struct cliente_data_t) é liberada.

Se o servidor foi encerrado, os pipes (sockfd_str e sockfd_num) são fechados e o pool de threads é destruído utilizando a função thpool_destroy.

9. Resultados obtidos

Seguindo as orientações de execução apresentadas no repositório, a biblioteca que executa o pool de threads não está pré-compilada e deve ser compilada junto com o projeto, como é demonstrado na Figura 1:

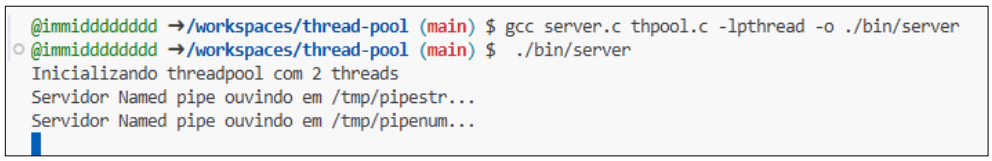

Figura 1: Compilação e execução do server.

Depois, em outro terminal, é compilado e executado o client, como é visto na Figura 2:

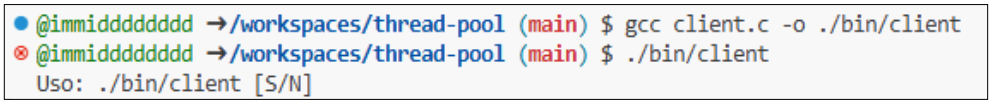

Figura 2: Compilação e execução do client.

Recebemos a mensagem "./bin/client [S/N]", que solicita qual pipe desejamos executar, S de string e N de numérico. Após a seleção é solicitado o dado a ser enviado ao servidor.

Na Figura 3, é possível observar a seleção de pipe S (string), e é enviado o dado "testeum". Como informado anteriormente, o tipo de pipe string converte o dado para maiúsculas.

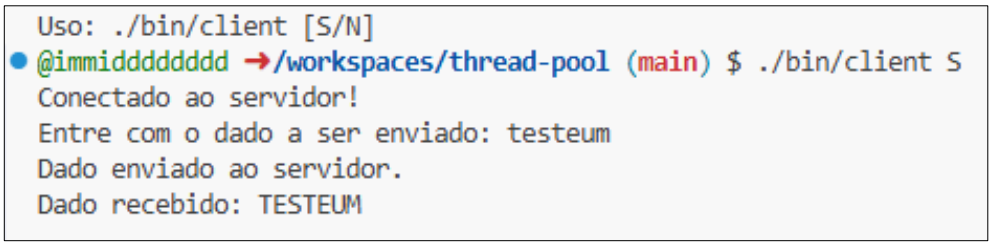

Figura 3: Realizada a execução do tipo de pipe string.

Outros testes do tipo de pipe string podem ser observados na Figura 4 abaixo:

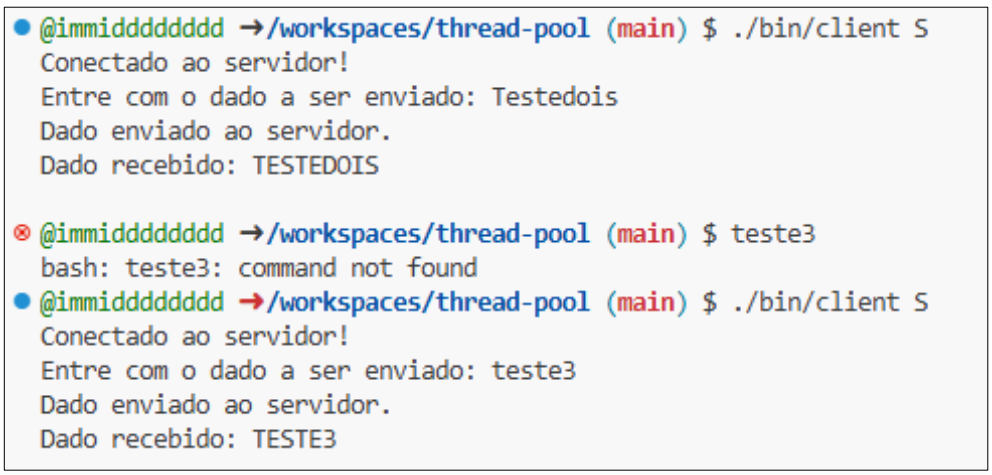

Figura 4: Testes do tipo de pipe string.

No tipo de pipe numérico, após selecionarmos o tipo N (numérico), recebemos a mensagem que estamos conectados ao servidor. Podemos então entrar com o dado a ser enviado.

Na Figura 5 abaixo é demonstrado alguns testes realizados:

```
@immidddddddd →/workspaces/thread-pool (main) $ ./bin/client N
Conectado ao servidor!
Entre com o dado a ser enviado: 10
Dado enviado ao servidor.
Dado recebido: 20
@immidddddddd →/workspaces/thread-pool (main) $ ./bin/client N
Conectado ao servidor!
Entre com o dado a ser enviado: 20349358
Dado enviado ao servidor.
Dado recebido: 20349368
@immidddddddd →/workspaces/thread-pool (main) $ ./bin/client N
Conectado ao servidor!
Entre com o dado a ser enviado: teste
Dado enviado ao servidor.
Dado recebido: 10
```

Figura 5: Testes realizados no tipo de pipe numérico.

10. Análise de desempenho

Utilizando o comando htop em um novo terminal, podemos realizar um monitoramento do sistema e visualizar em tempo real os processos em execução e também o uso dos recursos do sistema.

Na Figura 6 é apresentado o desempenho após algumas chamadas de sistema de um só cliente. É possível observar que o impacto no uso da CPU não foi muito significativo.

```
  1  [||||                                                6.9%]   Tasks: 24, 75 thr; 1 running
  2  [||||                                                5.6%]   Load average: 0.13 0.14 0.18
Mem[||||||||||||||||||||||||||||||||||||||||||1.42G/7.74G]   Uptime: 00:59:51
Swp[                                                    0K/0K]

    PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
   1706 codespace  20   0  2616  1536  1536 S  0.0  0.0  0:00.00 /bin/sh
   1739 root       20   0  2616  1408  1408 S  0.0  0.0  0:00.00 /bin/sh
   2144 root       20   0 1936M 79988 52864 S  0.0  1.0  0:00.02 dockerd
   2145 root       20   0 1936M 79988 52864 S  0.0  1.0  0:00.06 dockerd
   2146 root       20   0 1936M 79988 52864 S  0.0  1.0  0:00.00 dockerd
   2147 root       20   0 1936M 79988 52864 S  0.0  1.0  0:00.00 dockerd
   2148 root       20   0 1936M 79988 52864 S  0.0  1.0  0:00.00 dockerd
   2149 root       20   0 1936M 79988 52864 S  0.0  1.0  0:00.00 dockerd
   2179 root       20   0 1936M 79988 52864 S  0.0  1.0  0:00.00 dockerd
   2153 root       20   0 1756M 48888 31232 S  0.0  0.6  0:00.10 containerd
   2155 root       20   0 1756M 48888 31232 S  0.0  0.6  0:00.00 containerd
   2156 root       20   0 1756M 48888 31232 S  0.0  0.6  0:00.00 containerd
  19853 codespace  20   0  146M  1536  1408 S  0.0  0.0  0:00.00 thpool-0
  19854 codespace  20   0  146M  1536  1408 S  0.0  0.0  0:00.00 thpool-1
  19852 codespace  20   0  146M  1536  1408 S  0.0  0.0  0:00.00 ./bin/server
  25454 codespace  20   0 16792 11640  3328 S  0.0  0.1  0:00.14 /bin/bash --init-file /vscode/bin/linux-x64/38c31bc77e0dd
  31716 codespace  20   0 16792 11648  3456 S  0.0  0.1  0:00.10 /bin/bash --init-file /vscode/bin/linux-x64/38c31bc77e0dd
  39837 codespace  20   0  2620  1536  1536 S  0.0  0.0  0:00.00 /bin/sh -c "/vscode/bin/linux-x64/38c31bc77e0dd6ae88a4e9c
  39838 codespace  20   0  8628  3200  2944 S  0.0  0.0  0:00.00 /bin/bash /vscode/bin/linux-x64/38c31bc77e0dd6ae88a4e9cc9
  39842 codespace  20   0  7240  1920  1920 S  0.0  0.0  0:00.00 sleep 1
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Figura 6: Visualização apresentada pelo comando htop.

A coluna CPU% mostra um uso de processador muito baixo (0.0% para quase todos os processos e threads).

Isso sugere que as chamadas de sistema no client não estão gerando uma carga alta o suficiente para aumentar significativamente o uso de CPU. Isso pode indicar que o sistema está ocioso ou que as operações são rápidas, portanto não consomem muito tempo de CPU.

O uso da memória aparece normal, com 1.42 GB de 7.74 GB utilizados, e o swap não está sendo utilizado, o que indica que a memória física disponível é suficiente para a carga atual.

Em conclusão, o sistema está em um estado de baixa utilização de recursos, mesmo após as chamadas de sistema no client, o que sugere que são rápidas ou que a carga gerada por elas é muito pequena para afetar significativamente o desempenho.

Em outra demonstração, onde foi utilizado dois clientes conectados ao servidor e realizando chamadas de sistema, o desempenho é levemente alterado, como é visto na Figura 7:

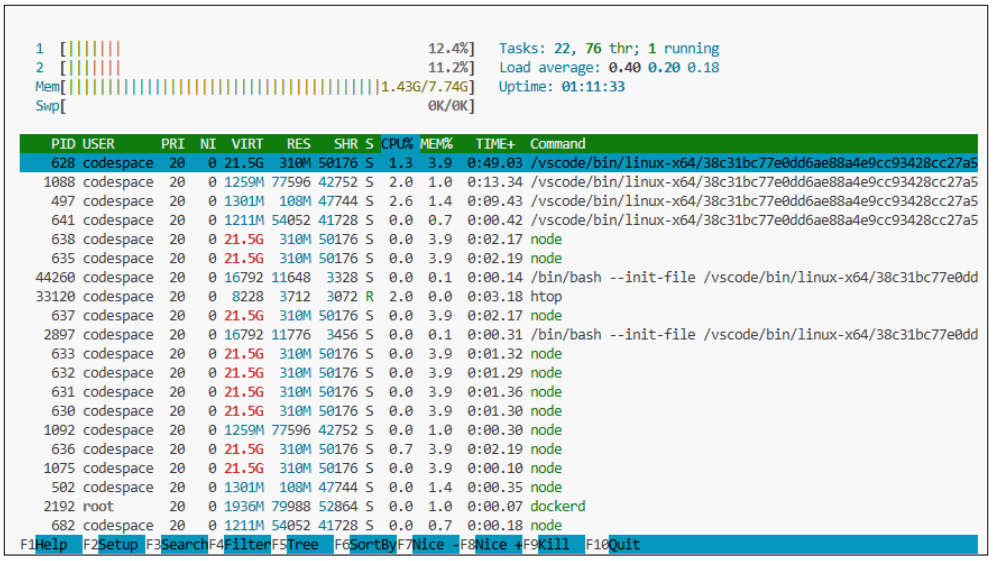

Figura 7: Segunda visualização apresentada pelo comando htop.

Neste segundo caso, o uso do processador parece estar bem distribuído entre as threads, com algumas utilizando até 2.6% da CPU. A maioria está em estado S (sleeping), o que é normal para threads que estão aguardando operações.

O load average apresenta resultados baixos (0.40, 0.20, 0.18: Isso significa que, em média, houve 0.40 processos em execução ou esperando nos últimos 1 minuto, 0.20 nos últimos 5 minutos e 0.18 nos últimos 15 minutos), o que sugere que o sistema não está sobrecarregado,

sendo um bom sinal de que as threads estão gerenciando bem as tarefas. Por fim, o uso de memória também parece estar sob controle.

Em conclusão, os resultados não apresentaram uma carga significante por serem chamadas rápidas e simples, além da utilização de poucos clientes no servidor. No entanto, as threads apresentaram um bom desempenho no uso da CPU, e o código funcionou da maneira esperada.