Relatório do Trabalho de Cliente-Servidor

Alunos	RA
Rafael Rodrigues dos Santos	94075
Ricardo Henrique Brunetto	94182

1 Introdução

Este documento explica brevemente como os programas foram implementados, tanto em aspectos de linguagem, como de decisões de projeto e teoria envolvida.

Serão usados os termos servidor e cliente para se referir ao código-fonte do programa que faz o papel de servidor e o que faz papel de cliente, respectivamente.

2 Visão geral

Os programas foram implementados com a linguagem C, rodando em ambiente Linux, por linha de comando. Cada um consiste de um único arquivo de extensão .c, que deve ser compilado e executado por meio do utilitário make, isto é, estando na respectiva pasta, é só digitar o comando make no terminal para compilar e executar.

Basicamente, ocorre o seguinte fluxo de execução:

1. No cliente, o usuário escolhe entre consultar um registro, inserir um novo registro ou sair
2. Feita a escolha, o usuário insere os dados solicitados
3. Uma requisição é feita ao servidor
4. O servidor (que já deve ter sido iniciado anteriormente), aceita a requisição, lê os dados, faz o processamento necessário e devolve uma resposta ao cliente

Como mencionado, o servidor deve estar em execução antes que o cliente faça uma requisição, caso contrário um erro será exibido e os dados inseridos terão de ser digitados novamente.

O servidor é capaz de atender a 5 requisições simultaneamente, isto é, 5 programas cliente executados ao mesmo tempo, enviando solicitações ao mesmo tempo. Caso ocorra uma sexta requisição simultânea, ela é descartada, caracterizando o problema do barbeiro dorminhoco.

Uma vez que a conexão é feita em máquina local, o tempo de envio e recebimento de dados é mínimo. O ideal seria ter várias máquinas rodando uma instância do cliente, acessando uma máquina rodando o servidor.

Os registros manipulados são constituídos de 3 campos: ID, Nome e Curso, todos no formato de string e concatenados com uma barra vertical (”) no final. Não há verificação de registros duplicados, ou validação de campos.

3 O cliente

É uma aplicação bem simples, que exibe um menu e demais dados necessários. Os dados são colocados em um buffer e enviados para o servidor. No caso de uma inserção, os três campos são concatenados e separados por uma barra vertical.

Antes de começar a interação com o usuário, a estrutura para a conexão com o servidor é montada, coletando o endereço da máquina em que o servidor estará rodando (no caso, a máquina local), e a porta, que pode ser qualquer número de 16 bits (foi fixado 5700).

Com a estrutura montada, após cada inserção de dados pelo usuário, é criado o socket para enviar a requisição e os dados. Para tal, utiliza-se a função socket, que retorna um descritor de arquivo para o socket.

Os dados então são enviados escrevendo nesse arquivo por meio da função write. No caso de uma consulta, é enviado o ID a ser buscado e, no caso de uma inserção, o buffer com os três campos concatenados. Em ambos os casos é enviado o tipo da requisisção (1 para consulta e 2 para inserção).

Após enviar os dados, o cliente fica esperando uma resposta, isto é, tenta ler uma sequência de bytes enviados pelo servidor. Essa leitura, feita pela função read só será feita após o servidor escrever algo e o cliente fica parado nesse ponto esperando essa resposta. Em caso de requisição de consulta, se o registro existir, ele será devolvido da mesma forma que foi enviado (campos concatenados e separados por uma barra vertical) e por isso precisa ser reformatado.

4 O servidor

É a parte mais elaborada e que utiliza mais recursos da linguagem, como threads, semáforo, memória compartilhada, pipe nomeado e criação de processo filho.

Ao iniciar o programa, é preparada a estrutura para a conexão (incluindo o socket) e a estrutura de memória compartilhada, a qual consiste de um descritor para o arquivo de registros e um semáforo. O uso dessa memória compartilhada será explicado posteriormente. Então ele fica esperando uma requisição chegar, o que é feito pela função accept. Isto é, tudo que vem depois dessa função só é executado após o recebimento de uma requisição do cliente.

Como o servidor deve ficar rodando ininterruptamente, toda essa parte é colocada em um loop while(1), em que cada iteração corresponde a uma requisição aceita.

4.1 Aceitando uma requisição

Se a função accept retornar sem erro, a conexão com o cliente foi estabelecida e, a partir desse ponto, ambos podem trocar dados. Nesse ponto o cliente também já sabe do sucesso da conexão e já enviou os dados para o servidor, os quais são lidos do socket pela função write. Esses dados são o tipo da requisição (consulta ou inserção) e os dados envolvidos.

Além de ler os dados, o servidor incrementa um contador de requisições e cria um pipe nomeado, também conhecido como fifo. Esse fifo é um arquivo que permite trocar informações entre processos e seu uso será explicado mais adiante.

4.2 Criando um processo filho

Tendo recebido a requisição, o servidor cria um processo filho por meio da função fork. Esse processo filho é uma cópia exata do processo pai, incluindo todo o espaço de enderçamento, isto é, todas as variáveis e os respectivos valores armazenados.

A partir daí, o fluxo de execução é diferenciado. O processo pai faz a leitura dos dados da requisição, escreve os dados no fifo, atualiza o log de requisições e encerra a conexão, ficando à espera de outra requisição. O filho lê os dados escritos no fifo e trata a requisição, enviando a resposta para o cliente.

É importante destacar que, apesar de pai e filho serem dois processos paralelos cuja execução depende do escalonador de processos, o filho não consegue prosseguir antes do pai escrever os dados no fifo. A chamada read fica travada enquanto não houver dados para serem lidos. Além disso, o filho mantém a conexão com o cliente aberta até que o tratamento da requisição seja concluído, mesmo que o pai tenha fechado a conexão. Isso é possível porque após a criação do processo filho, são dois processos totalmente distintos.

Outro ponto a se considerar é que cada requisição é tratada por um processo filho diferente, então para cada requisição tem um novo processo, um novo fifo, um novo socket e novos dados. Apenas o pai é o mesmo, mantendo-se no laço que espera uma requisição, cria o filho, lê os dados, passa os dados para o filho e atualiza o log.

4.3 Atualizando o log de requisições

Cada requisição que chega (seja de inserção ou de consulta) é registrada em um arquivo de log, o qual grava o id da requisição, o tipo e a data em que ocorreu. Como essa operação é independente do tratamento da requisição, ela é feita por meio de uma thread.

Threads são processos paralelos que compartilham o espaço de endereçamento do processo que as criou. Assim, ao mesmo tempo que as operações para manipular o log ocorrem (abrir arquivo, escrever e fechar arquivo), o processo pai continua seu fluxo, podendo inclusive atender uma nova requisição enquanto a primeira está sendo registrada.

Como a conexão está sendo feita em máquina local, tudo ocorre rápido demais, e se outra requisição for atendida enquanto uma primeira está sendo registrada, é preciso proteger o arquivo de log para que apenas uma thread por vez o manipule. A biblioteca que oferece suporte a threads implementa o mutex e fornece várias funções para manipulá-lo.

Antes de abrir o arquivo de log, a thread em execução tenta "pegar"o mutex para si, o que é feito pela função pthread_mutex_lock. Se nenhuma outra thread estiver com o mutex, a thread pega ele para si e o bloqueia, de modo que, se outra thread tentar escrever no arquivo de log, ela ficará bloqueada até que o mutex seja liberado pela função pthread_mutex_unlock.

Por questões de implementação da biblioteca, para passar o id e o tipo da requisição para a thread, foi criada uma estrutura que tem esses campos.

4.4 Tratando a requisição

Após ter lido os dados no fifo, o processo filho realiza as operações necessárias para tratar a requisição. Se for uma requisição de consulta, lê o arquivo de registros até encontrar um registro com o ID enviado pelo cliente ou até chegar no fim do arquivo (se isto acontecer, o registro não existe). Em caso de inserção, os campos enviados são escritos no arquivo de registro no mesmo formato em que foram enviados.

É aqui que entra a memória compartilhada e o semáforo. As requisições manipulam o mesmo arquivo de registro, o qual é aberto pelo mesmo descritor de arquivo, então é preciso proteger esse arquivo para que apenas um processo por vez tenha acesso a ele, caso contrário os dados podem não ser lidos e escritos corretamente.

Como já mencionado, um processo filho criado com a função fork começa com as mesmas variáveis e mesmo espaço de endereçamento, tendo apenas um fluxo de execução diferente. No entanto, o descritor do arquivo de registros e o semáforo foram criados como memória compartilhada, então na verdade todos os filhos acessam o mesmo arquivo, mas um de cada vez.

A proteção do arquivo se dá pelo semáforo, que basicamente é um contador que bloqueia processos quando atinge um dado valor. Como o arquivo de registros pode ser acessado por um processo de cada vez, o semáforo foi inicializado com 1 e, quando os registros são acessados, o semáforo é decrementado e incrementado (no caso, alternando entre 0 e 1).

Quando um filho, que está tratando uma requisição, tenta acessar os registros, ocorre a chamada da função sem_wait, que bloqueia o processo até que o valor do semáforo seja 1. Então o valor é decrementado (agora está em 0), o arquivo é acessado e, quando o processo não precisar mais do arquivo, ele chama a função sem_post, que incrementa o valor do semáforo (agora em 1), permitindo que outro processo acesse o arquivo de registros.

4.5 Respondendo o cliente

Terminadas as operações, o filho retorna alguma informação para o cliente, escrevendo no socket por meio da função write e enquanto a escrita não ocorrer o cliente fica travado esperando.

As possíveis respostas são:

- uma string com o registro cujo ID corresponde ao informado - em caso de consulta de um registro que foi encontrado

- a string NULL - em caso de consulta de um registro não encontrado
- uma string informando que o registro foi gravado com sucesso - em caso de inserção
- a string ERROR - caso algum erro ocorra e as operações não possam ser completadas

5 Arquivos auxiliares

O servidor cria dois arquivos temporários em /tmp, um para a memória compartilhada e outro para o fifo (um fifo para cada requisição).

Antes da execução do servidor, o makefile cria os arquivos de log e de registro, na mesma pasta do servidor, caso eles não existam. Ter os arquivos criados desde o início evita algumas verificações ao longo do código.

Referências

Foram consultados vários links para a realização do trabalho:

- https://stackoverflow.com/questions/16400820/c-how-to-use-posix-semaphores-on-forked-proce
- http://www.csc.villanova.edu/~mdamian/threads/posixsem.html
- http://www.cs.rpi.edu/~moorthy/Courses/os98/Pgms/socket.html
- https://computing.llnl.gov/tutorials/pthreads/
- http://menehune.opt.wfu.edu/Kokua/More_SGI/007-2478-008/sgi_html/ch03.html
- https://stackoverflow.com/questions/2784500/how-to-send-a-simple-string-between-two-progra