Universidade Federal do Rio Grande do Sul
Instituto de Informática - INF01151

Amanda Bandeira (180409)
Gabriella Barbieri (228992)
Eduardo Pereira (228333)
Lucas Biff (228871)

TRABALHO PRÁTICO PARTE 1: THREADS E SINCRONIZAÇÃO

Este trabalho tem como finalidade realizar a implementação de um serviço semelhante ao Dropbox, para a disciplina “Sistemas Operacionais II N - INF01151, turma A, e está dividido em duas partes. Sendo esta a primeira parte, ela compreende tópicos aprendidos durante a primeira metade da disciplina, tais como: threads, processos e sincronização.

O trabalho consiste em criar um servidor (Dropbox) responsável por gerenciar arquivos de diversos usuários (clientes). Para tal, o programa foi desenvolvido na linguagem C, utilizando a API de sockets Unix. O tipo de conexão usada foi a conexão TCP (Transmission Control Protocol), com um servidor concorrente, ou seja, capaz de tratar simultaneamente requisições de vários clientes.

Para o desenvolvimento do trabalho e a realização dos testes, utilizamos o macOS Sierra, versão 10.12.5. A máquina possui as seguintes características:

- Processador: 2,7 GHz, Intel Core i5;
- Memória: 16GB 1867 MhHz DDR3;
- Disco de inicialização: Macintosh HD;
- Gráficos: Intel Iris Graphics 6100 1536 MB.

O compilador usado possui as características abaixo:

- Apple LLVM version 8.1.0 (clang-802.0.42);
- Target: x86_64-apple-darwin16.6.0;
- Thread model: Posix.

Concorrência no Servidor

Para a concorrência no servidor, optamos por utilizar múltiplas threads. Uma thread é responsável por gerenciar o Servidor. Quando um cliente envia uma tentativa de conexão (Connect()), uma nova thread é criada. O servidor então, verifica se o cliente que fez a solicitação já possui o número máximo de duas conexões e, caso já possua, a conexão é fechada, e a thread é deletada. Caso o cliente ainda não tenha estabelecido o número máximo de conexões, o servidor aceita essa requisição (Accept()), e a thread criada se torna responsável por gerenciar essa conexão.

Sincronização no Acesso a Dados

Para sincronização no acesso a dados, optamos por utilizar uma variável do tipo mutex. Ela foi usada para garantir que apenas uma thread está lidando com o registro de sockets de cada sessão de usuário, ou seja, ele foi colocado para controlar a variável *devices*, da estrutura do cliente, se uma tentativa de conexão for feita, o mutex irá garantir acesso exclusivo à essa variável, para verificar se a conexão poderá ser estabelecida ou não.

Estruturas e Funções Adicionais

Para auxiliar no programa, foram utilizados os arquivos dropboxUtil.c e fila2.c, contendo as seguintes funções:

dropboxUtil.c

- int file_size(FILE *fp) - retorna o tamanho de um arquivo
- CLIENT* createClient(char* name) - cria um novo cliente
- FILEINFO* createFile(char* name,int size) - cria um novo arquivo
- char* getCurrentTime() - retorna a hora atual
- FILEINFO* findFile(CLIENT* user, char* name) - encontra arquivo dentro da lista de files do usuário
- CLIENT* findClient(PFILA2 clientsList, char* name) - encontra cliente dentre os cadastrados no servidor
- char* parseFilename - retorna nome do arquivo a partir de seu path
- int getFileFromStream(char* file, int socket) - pega um arquivo byte a byte de um socket
- char* getPath(CLIENT* user, char*file) - retorna o path do diretório do usuário onde o arquivo será salvo
- void parseNameExt(char* src, char* name, char* ext) - separa o nome de arquivo de sua extensão

fila2.c - usado para gerenciar lista de usuários no servidor e lista de arquivos de cada usuário

- int CreateFila2(PFILA2 pFila) - aloca uma fila vazia
- int FirstFila2(PFILA2 pFila) - seta um iterador para o primeiro elemento da fila
- int LastFila2(PFILA2 pFila) - seta um iterador para o último elemento da fila
- int NextFila2(PFILA2 pFila) - seta iterador para o próximo elemento da fila
- void *GetAtIteratorFila2(PFILA2 pFila) - retorna elemento da fila apontado pelo iterador

- int AppendFila2(PFILA2 pFila, void *content) - insere elemento ao fim da lista
- int InsertAfterIteratorFila2(PFILA2 pFila, void *content) - insere elemento após o iterador de fila
- int DeleteAtIteratorFila2(PFILA2 pFila) - deleta elemento apontado pelo iterador

Quanto às estruturas utilizadas, a estrutura *file_info* foi mantida em seu formato original:

```
struct  file_info {
        char name[MAXNAME];
        char extension[MAXNAME];
        char last_modified[MAXNAME];
        int size;
}.
```

A estrutura *client* também foi mantida no seu formato original:

```
struct  client  {
        int devices[2];
        char userid[MAXNAME];
        struct  file_info[MAXFILES];
        int logged_in;
}.
```

Para as funções de manipulação de filas, usamos as seguintes estruturas:

```
struct  sFilaNode2 {
        void    *node;              // Ponteiro para a estrutura de dados do NODO
        struct  sFilaNode2 *ant;    // Ponteiro para o nodo anterior
        struct  sFilaNode2 *next;   // Ponteiro para o nodo posterior
};

struct sFila2 {
        struct  sFilaNode2 *it;     // Iterador para varrer a lista
        struct  sFilaNode2 *first;  // Primeiro elemento da lista
        struct  sFilaNode2 *last;   // Último elemento da lista
};
```

Primitivas de Comunicação

Para realizar a comunicação entre o cliente e servidor, utilizamos as primitivas de comunicação *read()* e *write(),* e duas funções criadas: *receive_file()* e *send_file().* As duas funções criadas não são as primitivas *send()* e *receive()* propriamente ditas, mas foram criadas com base nelas, e, por isso, achamos válido explicar aqui seu funcionamento.

No lado do cliente, o *send_file(char *file, int socket)* foi usado quando o cliente seleciona a opção "*upload*", realizando o envio do arquivo ao repositório correspondente no servidor. O *receive_file(char* file, int socket)* foi usado quando o cliente seleciona a opção "*download"*, realizando o download do arquivo do repositório correspondente no servidor.

No lado do servidor, elas são usadas de forma inversa, ou seja, o *send_file(char*file, int socket, CLIENT* user)* foi usado quando o cliente seleciona a opção "*download*", realizando o envio do arquivo do repositório correspondente no servidor para o cliente que fez a solicitação. O *receive_file(char* file, int socket, CLIENT* user)* foi usado quando o cliente seleciona a opção "*upload"*, realizando o recebimento do arquivo no repositório correspondente ao cliente que fez a solicitação no servidor.

Ambas primitivas, *read()* e *write()* foram utilizadas em diversos trechos do código, que necessitavam de escrita e leitura no buffer para a comunicação dos sockets.

No servidor a primitiva *read()* foi usada em dois momentos: no loop de leitura do buffer do socket para o controle de seleção das possíveis ações requisitadas pelo cliente, e no *send_file()*, para a leitura da resposta do cliente, após o envio do arquivo solicitado. A primitiva *write()* foi usada nos seguintes lugares: loop de leitura do buffer do socket para a resposta enviada ao buffer no momento do login, no *send_file()*, para a escrita no buffer do tamanho do arquivo, e após, caso possua, a escrita byte a byte do mesmo no buffer, no *receive_file()*, para a escrita de mensagem no buffer em caso de update de arquivo (sucesso ou falha) e, na função *list_files()*, para realizar a escrita da lista de arquivos do cliente no diretório remoto, ou para escrever mensagem em caso de diretório vazio.

No cliente, a primitiva *read()* foi usada nos seguintes trechos: para a leitura da resposta de tentativa de conexão com o servidor, na função *print_file_list()*, lendo do buffer a lista de arquivos presentes no diretório remoto e, na função *send_file()*, para ler do buffer a resposta do servidor após o envio do arquivo. A primitiva *write()* foi usada nas seguintes configurações: requisição de tentativa de conexão com o servidor, no loop de leitura do buffer do socket do cliente, para o realizar a escrita no buffer correspondente a cada ação, na função *send_file(),* para escrever uma mensagem ou para realizar a escrita do tamanho do arquivo no buffer, e após, a escrita byte a byte do arquivo, na função *receive_file()*, para escrever no buffer uma mensagem confirmando o recebimento do arquivo e, na função *close_connection()*, para escrever no buffer que o cliente deseja encerrar a conexão.

Problemas

Ao realizarmos o levantamento dos requisitos do trabalho, e as funcionalidades por ele pedidas, nos deparamos com algumas questões, que

geraram interessantes pontos de discussão entre o grupo. Abaixo listamos quais foram esses pontos e questões:

1) **Como passar o arquivo pelo TCP se é uma stream?** Para resolver essa questão, optamos por passar o tamanho do arquivo junto com ele, pois sabendo qual será exatamente o tamanho recebido, a troca de informações pode ser feita byte a byte.
2) **Como limitar o número de devices?** Primeiramente, pensamos em controlar o número de devices de cada cliente utilizando um semáforo inicializado em 2, e, no momento em que uma conexão fosse estabelecida, P(s) seria feito decrementando o valor de s ou bloqueando a tentativa de conexão (s==0), quando a conexão fosse fechada, V(s) seria feito, incrementando o valor de s ou desbloqueando uma conexão, caso houvesse alguma bloqueada. Após analisarmos novamente a questão, percebemos que o comportamento do semáforo ao bloquear uma tentativa de conexão caso não houvesse nenhum disponível não era adequada à especificação pedida, visto que, caso já tivesse 2 conexões estabelecidas, o cliente deveria receber uma notificação e a tentativa ser rejeitada. Acabamos optando por resolver essa questão usando o campo *devices[2]*, da estrutura do cliente. Cada device será inicializado com -1 (valor indicando que o slot de socket está vazio), e se uma tentativa de conexão for feita, e não houver nenhum device com o valor ” -1”, manda uma mensagem para o cliente e fecha a tentativa de conexão. Para garantir acesso exclusivo à essa variável, usamos um mutex no controle do seu acesso.
3) **Como sincronizar os arquivos do servidor com o cliente?** Em um primeiro momento, pensamos em passar a fila de FILEINFO do servidor para o cliente, podendo este processar os arquivos em seu diretório local e comparar se tem algum arquivo faltante no servidor, ou se houve alguma modificação, poder fazer o upload do mesmo. O grande problema dessa abordagem seria o parser do pacote, várias perguntas foram levantadas, por exemplo “Como passar uma fila de estruturas por TCP?”, e não houve uma implementação que resolvesse essa questão. Outra sugestão foi passar cada informação de um arquivo (nome, extensão e última modificação) através de 3 escritas e leituras na rede. Acreditamos que por conta da implementação do TCP, muitas vezes o buffer chegava com mais informações que deveria, e mesmo dando três writes no servidor, o cliente muitas vezes ficava esperando em um read pois a informação deste read já havia chegado no anterior. Por conta desses problemas a parte de sincronização não pôde ser concluída a tempo.