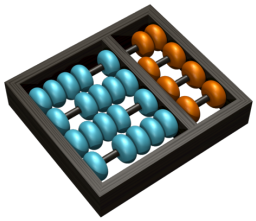

Exercício 4

Aluno: Felipe S. P. Carvalho e Arthur Maia Mendes

RA: 146040 e 135013

Instituto de Computação

Universidade Estadual de Campinas

Campinas, 14 de Outubro de 2018.

Sumário

1 Questão 1

Não. O servidor coloca os pedidos numa fila e trata eles sequencialmente, isto é, de forma síncrona. Isso ocorre porque apenas um processo está em execução no servidor a cada requisição, já que existe uma única thread em execução.

2 Questão 2

Nesta questão desenvolvemos dois programas em linguagem C, um cliente e um servidor, para comunicação entre sockets TCP. Os arquivos desta questão estão na pasta **questao-2/** e são dados por: `cliente.c` (programa cliente),`servidor.c` (programa servidor), Makefile (para compilar os programas), e README.md (guia para execução dos programas).

O servidor deve receber como argumento na linha de comando a porta na qual irá escutar, como por exemplo: `# ./servidor 8800`. Já o cliente deve receber como argumento na linha de comando o endereço IP do servidor e a porta na qual irá conectar, como por exemplo: `# ./cliente 127.0.0.1 8800`. Após executar o programa servidor (em vermelho), temos o seguinte output:

```
personal@kiss-mbp-3 ~/L/U/2/m/g/exercicio-4> ./servidor 8800
Iniciando servidor...
Socket criado
Bind completo
IP local da conexão: 127.0.0.1
Porta local da conexão: 8800
Aguardando conexoes...
```

Conforme especificado no enunciado, o servidor espera a entrada continuamente de uma cadeia de caracteres via teclado, que será enviado ao cliente. A forma como os programas foram implementados, sem utilizar o compartilhamento de `files descriptors` - que possibilitam compartilhar descritores de socket entre diferentes processos -, faz com que tenha-

mos que inserir o comando desejado mais de uma vez para que este seja propagado para todos os clientes. Essa é a única limitação existente no programa servidor. Além disso, o servidor exibe na saída padrão os dados de IP e PORTA utilizados na conexão. Vamos executar dois clientes (em azul e verde) em diferentes terminais para analisarmos os resultados:

```
personal@kiss-mbp-3 ~/L/U/2/m/g/exercicio-4> ./cliente 127.0.0.1 8800
Socket criado
Conectado em
IP: 127.0.0.1 Porta: 8800
```

Cliente A

```
personal@kiss-mbp-3 ~/L/U/2/m/g/exercicio-4> ./servidor 8800
Iniciando servidor...
Socket criado
Bind falhou: Address already in use
personal@kiss-mbp-3 ~/L/U/2/m/g/exercicio-4> ./cliente 127.0.0.1 8800
Socket criado
Conectado em
IP: 127.0.0.1 Porta: 8800
```

Cliente B

```
personal@kiss-mbp-3 ~/L/U/2/m/g/exercicio-4> ./servidor 8800
Iniciando servidor...
Socket criado
Bind completo
IP local da conexão: 127.0.0.1
Porta local da conexão: 8800
Aguardando conexoes...
Novo cliente conectado
IP: 127.0.0.1, Porta: 52075
Digite uma mensagem para ser enviada ao cliente:
Novo cliente conectado
IP: 127.0.0.1, Porta: 52088
Digite uma mensagem para ser enviada ao cliente:
```

Servidor

Assim que os programas clientes se conectam, é exibido na saída padrão de cada um o IP e PORTA do host servidor. Já no programa servidor, é exibido na saída padrão os dados dos hosts clientes ao qual o servidor está se conectando (IP e PORTA). Note que a função `getpeername()` não é utilizada no servidor para se descobrir o endereço IP e a PORTA dos clientes:

```
95
96      printf("Novo cliente conectado\nIP: %s, Porta: %u\n", ip, ntohs(client.sin_port));
97
```

Trecho de código no programa `servidor.c`

Vamos executar um comando no servidor para validarmos o exercício proposto. Temos a seguir os resultados:

```
personal@kiss-mbp-3 ~/L/U/2/m/g/exercicio-4> ./cliente 127.0.0.1 8800
Socket criado
Conectado em
IP: 127.0.0.1 Porta: 8800
/Users/personal/Local Documents/Unicamp/2018s2/mc833/git-exercicios/exercicio-4
/Users/personal/Local Documents/Unicamp/2018s2/mc833/git-exercicios/exercicio-4
```

Cliente A (2)

```
personal@kiss-mbp-3 ~/L/U/2/m/g/exercicio-4> ./servidor 8800
Iniciando servidor...
Socket criado
Bind falhou: Address already in use
personal@kiss-mbp-3 ~/L/U/2/m/g/exercicio-4> ./cliente 127.0.0.1 8800
Socket criado
Conectado em
IP: 127.0.0.1 Porta: 8800
/Users/personal/Local Documents/Unicamp/2018s2/mc833/git-exercicios/exercicio-4
```

Cliente B (2)

```
Enviando comando Unix [pwd] para 2 clientes...
Enviando para socket 4...
Enviando para socket 5...
Digite uma mensagem para ser enviada ao cliente:
Nova mensagem recebida de
IP: 127.0.0.1, Porta: 52075
Mensagem: pwd
Nova mensagem recebida de
IP: 127.0.0.1, Porta: 52088
Mensagem: pwd
```

Servidor (2)

O comando `# pwd` foi enviado pelo servidor para os dois clientes, conforme podemos ver na imagem Servidor (2). O servidor também exibe, para cada mensagem enviada, a linha de comando devolvida pelo cliente (que é idêntica à enviada), além das informações da conexão com o cliente. Já nos programas clientes, após receber a cadeia de caracteres do servidor, executa esta cadeia via comando `system` e imprime na saída padrão o resultado.

Caso o comando enviado pelo servidor não seja válido, os clientes retornarão uma mensagem "Comando Inválido". A seguir, temos um exemplo com um cliente apenas:

```
personal@kiss-mbp-3 ~/L/U/2/m/g/exercicio-4> ./cliente 127.0.0.1 8800
Socket criado
Conectado em
IP: 127.0.0.1 Porta: 8800
sh: hahaha: command not found
```

Cliente A (3)

Servidor (3)

Note que o comando inexistente hahaha foi enviado ao cliente, que retornou a mensagem esperada ao servidor. Por fim, funções envelopadoras foram utilizadas no desenvolvimento dos programas, conforme descrito no enunciado.

3 Questão 3

Nesta questão desenvolvemos dois programas em linguagem C, um cliente e um servidor, para comunicação entre sockets TCP. Os arquivos desta questão estão na pasta **questao-3/** e são dados por: `cliente.c` (programa cliente), `servidor.c` (programa servidor), Makefile (para compilar os programas), e README.md (guia para execução dos programas). O servidor deve receber como argumento na linha de comando a porta na qual irá escutar, como por exemplo: `# ./servidor 8800`. Já o cliente deve receber como argumento na linha de comando o endereço IP do servidor e a porta na qual irá conectar, como por exemplo: `# ./cliente 127.0.0.1 8800`. As modificações aqui realizadas, em relação ao exercício 2, é a implementação do registro das atividades de conexão e desconexão em um log, junto com os comandos enviados pelo servidor aos clientes. Além disso, também foram implementados os comandos `exitc` e `exits`, que encerram respectivamente as conexões dos clientes ou do servidor, e alterações no output do programa cliente e do programa servidor.

A forma como os programas foram implementados, sem utilizar o compartilhamento de `files descriptors` - que possibilitam compartilhar des-

critores de socket entre diferentes processos -, faz com que tenhamos que inserir o comando desejado mais de uma vez para que este seja propagado para todos os clientes (como o exercício 2). Essa é a única limitação existente no programa servidor.

Inicialmente, temos um exemplo de log de execução. Depois de inicializado, o servidor realiza a conexão com dois clientes. Em seguida, envia um comando `ls` para os clientes. Logo após, envia um comando `exitc` para os clientes o que desencadeia a desconexão dos clientes. Finalmente, envia um comando `exits` que desencadeia a desconexão do servidor. Como pedido na questão, os IPs e portas dos clientes são registrados a cada conexão/desconexão e cada registro é acompanhado do respectivo horário em que foi realizado.

```
CONECTADO IP 127.0.0.1 PORT 52600 TIME Sun Oct 14 17:44:53 2018
CONECTADO IP 127.0.0.1 PORT 52601 TIME Sun Oct 14 17:45:15 2018
COMANDO ENVIADO ls TIME Sun Oct 14 17:45:23 2018
COMANDO ENVIADO exitc TIME Sun Oct 14 17:45:43 2018
DESCONECTADO IP 127.0.0.1 PORT 52601 TIME Sun Oct 14 17:45:43 2018
DESCONECTADO IP 127.0.0.1 PORT 52601 TIME Sun Oct 14 17:45:43 2018
COMANDO ENVIADO exits TIME Sun Oct 14 17:45:51 2018
DESCONECTADO SERVIDOR TIME Sun Oct 14 17:45:51 2018
```

Log

A seguir, vamos analisar os outputs para os programas cliente e servidor após o envido de um comando válido para um cliente. Iremos analisar o envio do comando `ls` e `pwd`:

```
personal@kiss-mbp-3 ~/L/U/2/m/g/t/questao-3> ./cliente 127.0.0.1 8000
Socket criado
Conectado em
IP: 127.0.0.1 Porta: 8000
Data e Hora de recebimento: 2018-10-14 16:41:53
README.md       cliente         cliente.c       makefile        servidor        servidor.c
Data e Hora de recebimento: 2018-10-14 16:41:56
/Users/personal/Local Documents/Unicamp/2018s2/mc833/github-repo/trabalho-4/questao-3
```

Cliente A

```
personal@kiss-mbp-3 ~/L/U/2/m/g/t/questao-3> ./servidor 8000
Iniciando servidor...
Socket criado
Bind completo
IP local da conexão: 127.0.0.1
Porta local da conexão: 8000
Aguardando conexoes...
Novo cliente conectado
IP: 127.0.0.1, Porta: 50858
Digite uma mensagem para ser enviada ao cliente:
ls
Enviando comando Unix [ls] para 1 clientes...
Enviando para socket 4...
Digite uma mensagem para ser enviada ao cliente:
Nova mensagem recebida de
IP: 127.0.0.1, Porta: 50858
Mensagem: README.md
cliente
cliente.c
makefile
servidor
servidor.c

pwd
Enviando comando Unix [pwd] para 1 clientes...
Enviando para socket 4...
Digite uma mensagem para ser enviada ao cliente:
Nova mensagem recebida de
IP: 127.0.0.1, Porta: 50858
Mensagem: /Users/personal/Local Documents/Unicamp/2018s2/mc833/github-repo/trabalho-4/questao-3
```

Servidor

O cliente exibiu na saída padrão somente o comando enviado pelo servidor, além da data e hora de cada recebimento. Já o programa servidor, exibiu os dados de IP e PORTA seguido da mensagem enviada pelo cliente, que é o retorno dos comandos `ls` e `pwd`. Em relação aos novos comandos implementados para encerrar a conexão, temos o seguinte retorno para `exitc`:

```
personal@kiss-mbp-3 ~/L/U/2/m/g/t/questao-3> ./cliente 127.0.0.1 8000
Socket criado
Conectado em
IP: 127.0.0.1 Porta: 8000
Data e Hora de recebimento: 2018-10-14 16:41:53
README.md       cliente         cliente.c       makefile        servidor        servidor.c
Data e Hora de recebimento: 2018-10-14 16:41:56
/Users/personal/Local Documents/Unicamp/2018s2/mc833/github-repo/trabalho-4/questao-3
personal@kiss-mbp-3 ~/L/U/2/m/g/t/questao-3>
```

Cliente A

```
exitc
Enviando comando Unix [exitc] para 1 clientes...
Enviando para socket 4...
Data e Hora de encerramento de conexão com o cliente IP 127.0.0.1, Porta 50858: 2018-10-14 16:47:13
Digite uma mensagem para ser enviada ao cliente:
```

Servidor

Vemos que o servidor encerrou a conexão com o cliente, e o IP e PORTA dos clientes foram encerrados. Ao se executar o comando `exits`, o programa servidor irá executar o comando `exitc` para todos os processos e, em seguida, encerrar sua conexão local:

```
exits
Enviando comando Unix [exits] para 1 clientes...
Enviando para socket 4...
Data e Hora de encerramento de conexão com o cliente IP 127.0.0.1, Porta 50858: 2018-10-14 16:48:2
fish: './servidor 8000' terminated by signal SIGTERM (Polite quit request)
personal@kiss-mbp-3 ~/L/U/2/m/g/t/questao-3>
```

Servidor

4 Questão 4

O servidor continua escutando e os clientes continuam com suas conexões estabelecidas pois o socket de escuta TCP descrito pelo descritor `lfd` é usado para aguardar conexões de entrada TCP em uma porta específica e, após a chamada de `accept`, um novo descritor de socket é criado. No exemplo, ocorre uma chamada `fork()` que comapra o PID com 0, no caso o processo filho. Sendo assim, o processo filho é o processo que executa os comandos dentro do IF, isto é, `Close(listenfd)` e `Close(connfd)`, enquanto que o processo pai apenas executa `Close(connfd)`. Como o trecho está em um `FOR LOOP`, a próxima chamada de `accept` irá manter a conexão com o cliente estabelecida, e o servidor continuará escutando pois o processo pai não encerrou o socket `listenfd`.

O uso do `Close(connfd)` é para encerrar a conexão socket estabelecida com o cliente, já o `Close(listenfd)` é para encerrar o socket de escuta do servidor.

5 Questão 5

Quando um processo realiza um `fork`, os descritores de arquivo são duplicados no processo filho. No entanto, esses descritores de arquivo são distintos um do outro. Fechar um descritor de arquivo no filho não afeta o descritor de arquivo correspondente no pai, ou vice-versa. No caso descrito no exercício, se o filho executar as duas chamadas de `close` antes do processo pai receber o retorno do `fork`, nada ocorrerá no processo pai. O processo filho irá encerrar os sockets relativos apenas ao seu contexto.

6 Questão 6

Ao retirarmos a chamada do `bind` no arquivo servidor, porém mantendo a chamada de `listen`, obtivemos o seguinte erro ao se executar o programa cliente:

```
personal@kiss-mbp-3 ~/L/U/2/m/g/e/e/questao-2> ./cliente 127.0.0.1 8800
Socket criado
Falha no estabelecimento da conexao: Connection refused
```

Cliente

A chamada `bind` é responsável por associar o socket com o endereço local do servidor, possibilitando aos clientes utilizarem esse endereço para se conectarem ao servidor. A chamada de `listen` é a que passa a escutar as requisições dos clientes. Entretanto, sem a chamada do `bind` para associar o socket com o endereço, o cliente é rejeitado ao se tentar conectar ao servidor, visto que o endereço requerido não foi corretamente mapeado e exposto por parte do servidor.

7 Questão 7

Nesta questão, utilizamos os códigos de cliente e servidor do exercício 3, pois possuem os comandos que encerram as conexões. Primeiramente, executamos o servidor e o cliente em terminais distintos. Após a conexão estar estabelecida, inserimos o comando `exitc` no servidor para encerrarmos as conexões com os clientes. Em seguida, executamos o comando `# netstat -f inet -an` no terminal cliente para se obter informações das conexões TCP realizadas:

```
tcp4       0      0  127.0.0.1.1023         *.*                    LISTEN
tcp4       0      0  127.0.0.1.57207        127.0.0.1.8000         TIME_WAIT
udp4       0      0  *.51461                *.*
```

Após observar a saída do comando acima, nota-se que obtivemos uma conexão em estado `TIME_WAIT` na porta local 57207. Devido ao fato da maneira como os protocolos TCP/IP funcionam, as conexões não são fechadas de forma imediata. O estado de `TIME_WAIT` indica que o endpoint local, isto é, o cliente, encerrou a conexão. Após o cliente receber o comando `exitc` do servidor, o seguinte trecho de código do programa cliente encerra a conexão:

```
int ProcessConnection(int s) {

  ...

 //Encerra a conexao se o comando for "exitc" ou "exits"
 if (strcmp(buf, "exitc") == 0 || strcmp(buf, "exits") == 0) {
   return 0;
 }

  ...
```

```
  //Envia a mensagem
  if (send(s, output, strlen(output), 0) < 0) {
    puts("Envio falhou");
    return 0;
  }

  return 1;
}
```

O programa servidor também realiza uma chamada de `close` para o socket do cliente, após enviar a mensagem referente ao encerramento de conexão. Entretanto, não obtivemos uma conexão em estado `TIME_WAIT` com porta de origem 8000 (isto é, na perspectiva do servidor), pois o socket de comunicação ainda está aberto na perspectiva do servidor. Como a comunicação é apenas uma, o comando `# netstat -f inet -an` nos retorna que o responsável por encerrar a conexão é o cliente.