MC833 A - Programação de redes de computadores
Relatório - Tarefa 05

093125 - Tiago Martinho de Barros - *tiago.ec09@gmail.com*
093175 - Victor Fernando Pompêo Barbosa - *victorfpb@gmail.com*

12 de maio de 2016

Prof. Paulo Lício de Geus
IC – UNICAMP

Sumário

1 Introdução

Nesta tarefa estudaremos e melhoraremos um servidor TCP concorrente que usa a função `select`.

2 Questão 1

O código do programa *echo_server_select_tcp* foi estudado e abaixo se encontram explicações sobre algumas funções e macros que são usadas no programa.

Todas as funções e macros estão definidas em ⟨*sys/select.h*⟩.

- select

  Assinatura:

```
int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval
    *timeout);
```

  Esta função bloqueia o processo que a invoca até que haja atividade em qualquer um dos conjuntos de descritores de arquivos especificados ou até que um certo tempo passe.

  nfds é o descritor de arquivos com o maior número dentre os descritores dos conjuntos passados à função (somado com 1). O conjunto de descritores de arquivos readfds são verificados quanto à disponibilidade de leitura; o conjunto de descritores de arquivos writefds são verificados quanto à disponibilidade de escrita; e o conjunto de descritores de arquivos exceptfds são verificados quanto a condições excepcionais. Quando a função é executada, esses conjuntos são modificados para indicar quais descritores de arquivos mudaram de condição. E timeout especifica o tempo máximo que a função deve esperar para que descritores de arquivos estejam prontos.

  O valor de retorno é o número de descritores de arquivos contidos nos três conjuntos de descritores de arquivos (lembrando que a função modifica esses conjuntos). Em caso de erro, -1 é retornado.

- FD_ZERO

  Assinatura:

```
void FD_ZERO(fd_set *set);
```

  Esta macro inicializa o conjunto de descritores de arquivos set como um conjunto vazio.

- FD_SET

  Assinatura:

```
void FD_SET(int fd, fd_set *set);
```

  Esta macro adiciona o descritor de arquivos fd ao conjunto de descritores de arquivos set.

- FD_ISSET

  Assinatura:

```
int FD_ISSET(int fd, fd_set *set);
```

  Esta macro verifica se o descritor de arquivos fd faz parte do conjunto de descritores de arquivos set. Se sim, retorna um valor não nulo (true); caso contrário, retorna 0 (false).

- FD_CLR

  Assinatura:

```
void FD_CLR(int fd, fd_set *set);
```

  Esta macro remove o descritor de arquivos fd do conjunto de descritores de arquivos set.

3 Questão 2

Testamos o servidor *echo_server_select_tcp* com nossos clientes das tarefas 3 e 4 (alterando o valor da porta de conexão), com o telnet e com o netcat. Este servidor não produz nenhuma saída no terminal que o executa, apenas ecoa para o cliente o texto que lhe foi enviado. Abaixo temos as saídas dos clientes:

- Cliente da tarefa 3

```
bash-4.3$ ./client-t3 localhost
Cliente T3
Cliente T3
```

- Cliente da tarefa 4

```
bash-4.3$ ./client-t4 localhost
-------------------
IP local: 127.0.0.1
Porta local: 52809
-------------------

Cliente T4
Cliente T4
```

- Cliente telnet

```
bash-4.3$ telnet localhost 56789
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Cliente telnet
Cliente telnet
```

- Cliente netcat

```
bash-4.3$ nc localhost 56789
Cliente netcat
Cliente netcat
```

Todos esses clientes foram executados "simultaneamente", como mostra o `netstat` (omitindo as linhas de conexões irrelevantes para o nosso caso):

```
bash-4.3$ netstat -tu
Active Internet connections (w/o servers)
Proto Recv-Q Send-Q Local Address       Foreign Address       State
tcp       0      0 localhost.localdo:56789 localhost.localdo:52748 ESTABLISHED
tcp       0      0 localhost.localdo:52809 localhost.localdo:56789 ESTABLISHED
tcp       0      0 localhost.localdo:56789 localhost.localdo:52809 ESTABLISHED
tcp       0      0 localhost.localdo:56789 localhost.localdo:52817 ESTABLISHED
tcp       0      0 localhost.localdo:52748 localhost.localdo:56789 ESTABLISHED
tcp       0      0 localhost.localdo:52915 localhost.localdo:56789 ESTABLISHED
tcp       0      0 localhost.localdo:56789 localhost.localdo:52915 ESTABLISHED
tcp       0      0 localhost.localdo:52817 localhost.localdo:56789 ESTABLISHED
```

Notamos os pares de conexões {localhost:56789 ↔ localhost:52748, localhost:56789 ↔ localhost:52809, localhost:56789 ↔ localhost:52817 e localhost:56789 ↔ localhost:52915}.

4 Questão 3

O servidor funciona bloqueando na função `select` esperando que, pelo menos, um dos descritores de arquivos do conjunto *rset* tenha conteúdo para ser lido. Neste conjunto estão o descritor *listenfd* (que escuta novas conexões) e um *connfd* para cada conexão ativa (que escuta cada cliente).

Assim, quando uma nova requisição de conexão chegar (nova atividade no *listenfd*), o servidor vai aceitar, armazenar o recém-gerado descritor de arquivos do socket aceito e incluir esse descritor de arquivos no conjunto de todos os descritores (*allset*), cuja cópia (*rset*) será usada na chamada da função `select`. E depois chama a função `select`, que irá verificar nova atividade nos descritores de arquivos. Se houver, age de acordo; se não houver, espera até que haja atividade.

Quando um cliente com uma conexão com o servidor já estabelecida envia texto para o servidor (nova atividade em um dos *connfd*), a mesma chamada à função `select` vai detectar isso e o código verifica se foi o *listenfd* que recebeu essa atividade. Como nesse caso, não é, o servidor procura quais dos descritores de arquivos (quais clientes) receberam atividade (usando FD_ISSET), lê o texto enviado por cada cliente e os envia de volta para o respectivo cliente. Depois, chama a função `select` para processar a próxima atividade (nova conexão ou texto enviado por cliente) se houver, ou esperar uma nova atividade em um dos descritores de arquivos.

Este servidor não é paralelo realmente, mas como ele não bloqueia na função *read* (pois só a chama quando houver texto a ser lido), pode atender a vários clientes "simultaneamente", pois as requisições de conexões dos clientes são enfileiradas e tratadas uma a uma; e os textos enviados por cada cliente estão associados ao seu respectivo *socket* (descritor de arquivos) e também são tratados um a um. Desta forma, o servidor não executa nada em paralelo, porém como seu processamento é rápido, ele consegue atender mais de um usuário "simultaneamente".

5 Questão 4

No código fornecido do programa *echo_server_select_tcp*, algumas funções de uso chave para o funcionamento do servidor não tinham seus valores de retorno checados, possibilitando que o programa assumisse comportamento inesperado em caso de erro. São elas: `select, accept, read` e `send`. Dessa maneira, cada uma delas passou a ter seu valor de retorno checado e, caso o valor retornado indique erro, o servidor imprime na tela uma mensagem especificando a função problemática. No caso das funções `select` e `accept`, o programa é terminado.

Como exemplo, utilizaremos a função `select`. A chamada exibida a seguir estava presente no programa original.

```
nready = select(maxfd+1, &rset, NULL, NULL, NULL);
```

Com a adição da checagem dos valores de retorno, o trecho de código a seguir substituiu o anterior.

```
if( (nready = select(maxfd+1, &rset, NULL, NULL, NULL)) == -1 ){
   perror("simplex-talk: select");
   exit(1);
}
```

As outras funções sofreram modificações semelhantes.

6 Questão 5

O uso de `fork` é uma forma de lidar com múltiplas conexões, adotando como saída a utilização de múltiplos processos. As principais desvantagens dessa abordagem são duas:

1. Caso o servidor tenha que lidar com um grande número de conexões simultâneas, como cada uma delas utiliza o `fork`, o servidor pode exaurir a memória de aplicação, alcançando o número máximo de processos;

2. Como cada chamada de `fork` duplica variáveis, descritores de arquivos e todo o contexto do programa, o servidor pode ficar sem memória no caso de cada conexão necessitar de qualquer tipo de processamento significante.

Por outro lado, o uso de `select` gerencia múltiplas conexões utilizando apenas um processo. Dessa maneira, os problemas referentes ao overhead de duplicar processos não existem. Essa saída é mais custosa para o programador, por ser mais complexa, mas lida bem com os problemas inerentes ao uso de `fork`. Contudo, o uso do `select` tem a limitação do **FD_SETSIZE**, que define o tamanho do conjunto de descritores de arquivos e, na máquina utilizada, é 1024. Ou seja, a aplicação consegue atender a 1024 clientes simultaneamente, enquanto que a solução com `fork` está limitada pelo número de processos que um usuário pode ter executando que, na máquina utilizada, é 31161 (obtido via `ulimit -u`).

O uso de `select` também enfrenta problemas de escalabilidade advindos do funcionamento da função. Quando há um número grande de clientes conectados simultaneamente, o custo de criar, manter e checar o conjunto de descritores de arquivos é bastante grande.

No entanto, a chamada à função `select` cria uma situação de concorrência *aparente*, com a habilidade de lidar com múltiplas conexões, mas sem o uso de diferentes threads ou processos. Na prática, isso significa que o servidor gerencia as diferentes conexões *sem troca de contexto*. Dessa maneira, é possível compartilhar dados entre as diferentes sessões e clientes. Caso isso seja desejável, isso é uma vantagem do uso de `select`; no entanto, exige um cuidado adicional do programador para que dados não sejam erroneamente compartilhados entre clientes distintos, causando problemas de segurança.