

# **MC833 Relatório 5**

## **Backlog e Processos Zumbis**

**Aluno:** Fábio Camargo Ricci

**RA:** 170781

Instituto de Computação  
Universidade Estadual de Campinas

Campinas, 09 de Novembro de 2021.

# Sumário

1	Questões . . . . .	2
2	Respostas . . . . .	3

# 1 Questões

1. Explique o que é multiplexação de E/S e faça um resumo das diferenças entre os 5 tipos de E/S.
2. Modifique o programa cliente da atividade 3.2 para que este receba como entrada e envie ao servidor linhas de um arquivo texto qualquer. Explique no relatório as mudanças que considera mais importantes para sua solução.
  - (a) O arquivo de texto de entrada será passado utilizando o caracter de redirecionamento '<'
  - (b) O cliente continuará recebendo o eco enviado pelo servidor, que deverá ser escrito em um arquivo. Esse arquivo será criado utilizando o caracter de redirecionamento '>'.
  - (c) Seu programa deverá necessariamente utilizar ou a função select ou a função poll.
  - (d) Cada linha deve ser enviada separadamente para o servidor. O servidor irá enviá-las de volta para o cliente, então cuidado com os '\n' e '\t'.
  - (e) O cliente deve finalizar sua execução assim que tiver recebido todo o arquivo ecoado pelo servidor. Na saída padrão do servidor deve imprimir informações do cliente quando o cliente finalizar sua execução.
3. Segundo o observado na solução do ponto 2, explique qual a limitação de usar a função close para fechar a conexão e qual a vantagem de usar a função shutdown no seu lugar.

## 2 Respostas

1. Se trata da capacidade de avisar o kernel que o processo deseja ser notificado quando condições de E/S (entrada e saída) estejam válidas (ex: dados para leitura estão disponíveis). Os 5 tipos de E/S são:
  - (a) E/S bloqueante: Quando um processo faz a chamada de sistema, o mesmo é bloqueado e retorna a execução somente quando os recursos requisitados estiverem disponíveis.
  - (b) E/S não bloqueante: Quando um processo faz a chamada de sistema, o kernel retorna o erro `EWOULDBLOCK` caso os recursos não estejam disponíveis ou `OK` caso estejam.
  - (c) Multiplexação de E/S: Bloqueia o processo quando uma ou mais condições de E/S estão prontas para utilização, por ex:  
Primeiramente é chamado um `select()` que bloqueia o processo até os recursos estiverem disponíveis e retorna "readable". Após isso, o mesmo faz a chamada de sistema `recvfrom()` que o bloqueia novamente até a cópia dos dados do kernel para o usuário ser finalizada.
  - (d) E/S orientada a sinal: O processo solicita ao kernel que notifique quando um evento ocorrer através do sinal `SIGIO`. Não é uma chamada bloqueante.
  - (e) E/S assíncrona: É uma chamada de sistema assíncrona não bloqueante, informando ao kernel que realize a operação e copie os dados do mesmo para o usuário. Notifica o processo quando toda a operação for finalizada.
2. O cliente foi modificado para ler o input da entrada padrão, enviar

os comandos para o servidor e imprimir os resultados na saída padrão.

Como o cliente trabalha com I/O na leitura do arquivo de entrada e na troca de mensagens com o servidor, utilizou-se a função **select()** para a multiplexação de I/O. Ao chamar a função **FD\_SET(fd)**, o processo indica ao kernel do SO que deseja ser notificado quando houver operações de I/O no file descriptor (fd) em questão. Dessa forma, sempre que houver uma operação de I/O basta utilizar a função **FD\_ISSET()** para verificar sobre qual file descriptor aquela operação se trata.

O código que trata essa lógica é o seguinte:

```

FD_ZERO(&rset);

for (;;)
{
    FD_SET(fileno(stdin), &rset); // watch stdin
    FD_SET(sockfd, &rset);        // watch socket

    int max_sd = (fileno(stdin) > sockfd) ? fileno(stdin) : sockfd;

    select(max_sd + 1, &rset, NULL, NULL, NULL);

    if (FD_ISSET(sockfd, &rset)) // atividade no socket
    {
        // le retorno do servidor
        if (Read(sockfd, buffer, BUFFER_SIZE) == 0) // 0 significa d
        {
            break;
        }
        strcat(recvStr, buffer); // append received

        show_info_received(server_addr, buffer); // save server info

        if (strcmp(sentStr, recvStr) == 0) // if sent and received a
        {
            break;
        }
    }


    if (FD_ISSET(fileno(stdin), &rset)) // atividade na entrada pad
    {
        // le da entrada padrão
        if (fgets(buffer, BUFFER_SIZE, stdin) != NULL)
        {
            Write(sockfd, buffer); // write to server
            strcat(sentStr, buffer); // append sent
        }
    }
}

```

Com isso, redirecionando a entrada e saída padrão do cliente para os

arquivos in.txt e out.txt respectivamente, compilou-se e executou-se o servidor e o cliente ():

in.txt:

```
Relatório 5 >  in.txt
1      Lab 5
2      Teste
3      Fim
```

Servidor:


```
fabio@Fabios-MacBook-Pro Relatório 5 % gcc -o servidor servidor.c -Wall
fabio@Fabios-MacBook-Pro Relatório 5 % ./servidor
Listening on 0.0.0.0 port 57096

Client with IP: 127.0.0.1 Port: 57099 connected time: Mon Nov 22 11:23:56 2021
Client with IP: 127.0.0.1 Port: 57099 disconnected on: Mon Nov 22 11:23:56 2021
█
```

Cliente:

```
[fabio@Fabios-MacBook-Pro Relatório 5 % gcc -o cliente cliente.c -Wall
fabio@Fabios-MacBook-Pro Relatório 5 % ./cliente 0.0.0.0 57096 < in.txt > out.txt
fabio@Fabios-MacBook-Pro Relatório 5 % █
```

out.txt:

```
Relatório 5 >  out.txt
1      Received: 'Lab 5
2      Teste
3      Fim' from server on IP: 0.0.0.0 Port: 57096
4      █
```

Para compilar os programas, utilizou-se os comandos:

```
gcc -o servidor servidor.c -Wall
```

```
gcc -o cliente cliente.c -Wall
```

Para executar os programas, utilizou-se os comandos:

```
./servidor
```

```
./cliente IP_SERVIDOR PORTA_SERVIDOR < in.txt > out.txt
```

Vale notar que colocou-se uma limitação do tamanho do arquivo de entrada de 40960 bytes ( $10 * \text{BUFFER\_SIZE}$ ).

3. As funções **close()** e **shutdown()** funcionam de maneiras diferentes:

- (a) **close()**: Bloqueia a comunicação com o socket o o destrói (termina suas conexões e destrói o file descriptor que habilita troca de mensagens). Toda mensagem trocada com esse socket após isso resultará em uma exceção ao processo.
- (b) **shutdown()**: Bloqueia a comunicação com um socket sem que o mesmo seja destruído e recebe um parâmetro **how** que determina seu funcionamento:
  - i. **SHUT\_RD**: Bloqueia o socket de receber mensagens. Apesar disso, processos que leem do mesmo ainda podem ler dados no buffer (após isso, apenas receberão mensagens vazias).
  - ii. **SHUT\_WR**: Bloqueia o socket de enviar mensagens. O mesmo também informa aos clientes conectados que o envio de mensagens está suspenso.
  - iii. **SHUT\_RDWR**: Bloqueia o socket de receber e enviar mensagens. Possui o mesmo funcionamento que chamar **shut-**



**down(sockfd, SHUT\_RD)** e **shutdown(sockfd, SHUT\_WR)**  
em seguida

Dessa forma, ao usar **shutdown**, o cliente e servidor podem indicar um ao outro que estão finalizando comunicações para que a contraparte possa tomar as ações apropriadas (vale notar que essa função não destrói o socket). A função **close** irá destruir o socket e finalizar a conexão no momento que é chamada, de modo que a contraparte irá receber um erro ao tentar se comunicar via o socket destruído novamente, não sendo possível, em alguns casos, fazer o encerramento harmonioso da conexão entre as duas partes.