Exercício 3: Servidor TCP (HTTP) concorrente

***** Data de entrega: 30 de Setembro de 2025.*

**** Este trabalho deve ser realizado utilizando a linguagem C. Trabalhos que utilizarem bibliotecas externas poderão receber nota zero na implementação, pois todos os trabalhos serão compilados utilizando o padrão destas linguagens. Isto vale para quem utiliza Windows, certifique-se de compilar e executar em máquinas com Linux/Unix antes de entregar a tarefa.*

**** Lembre-se de justificar e comprovar suas respostas no relatório.*

**** Relatório arquivo deverá ser em formato PDF*

**** Detalhes da implementação:*

- *Trabalho em dupla*
- *Enviar um `.zip`/`.rar`/`.tar.gz` para classroom.*
- *O pacote deve conter:*
    1. *Um único `.pdf` com o relatório, saídas de execução, nomes e endereços das máquinas utilizadas, e respostas às questões.*
    2. *Os códigos `server_http.c` e `client_http.c` devidamente comentados com wrappers e **sem warnings**.*
- *Compile com `-Wall` e garanta **0 warnings**. Regra para atribuição de nota: Relatório 50% e Código 50%.*

Objetivo: O exercício é focado em um **servidor HTTP mínimo** que responde a GET / HTTP/1.0 com concorrência por processo.

1. "Segurar" a primeira conexão: Modifique o servidor para inserir um `sleep()` antes de fechar o `connfd`, de modo que a primeira conexão fique "segurada" por alguns segundos. Em seguida: Conecte dois clientes quase ao mesmo tempo. O servidor aceita as duas conexões de forma concorrente? Explique.
   Obs.: este passo é só para teste, o código do teste não precisa ser enviado.

2. Implemente um servidor TCP concorrente que:
    - Recebe a porta na linha de comando.
    - Aceita conexões e cria um processo filho por conexão.

    Lê a primeira linha da requisição e, se for `GET / HTTP/1.0` ou `GET / HTTP/1.1`, responde com:

    ```
    HTTP/1.0 200 OK\r\n
    Content-Type: text/html\r\n
    Content-Length: 91\r\n
    Connection: close\r\n
    \r\n
    <html><head><title>MC833</title</head><body><h1>MC833 TCP
    Concorrente </h1></body></html>
    ```

    - Para qualquer outro caminho ou método, responda `404 Not Found` ou `400 Bad Request` de forma simples.

- Imprime no stdout do servidor, no momento da conexão, o IP e a PORTA do cliente e a linha de requisição recebida.
- Deve atender a vários clientes de forma concorrente usando `fork()`.

Obs.:

- Não use `getpeername()` no servidor. Utilize o endereço retornado por `accept()` para saber IP e porta do cliente.
- Escreva e utilize funções envelopadoras para as chamadas de sockets seguindo a convenção do livro, com o mesmo nome iniciando em maiúscula.
- Não implemente, por enquanto, tratamento de backlog customizado nem coleta de zumbis. Esses pontos serão tratados nos próximos exercícios.

3. Implemente um cliente simples (browser) que:

   - Recebe IP do servidor e porta na linha de comando.
   - Abre a conexão, imprime seu IP e PORTA locais e os dados do servidor remoto.

     Enviar exatamente:

     ```
     GET / HTTP/1.0\r\n
     Host: teste\r\n
     \r\n
     ```

   - Lê a resposta e a imprime integralmente no stdout.
     Obs.: Pode usar `getsockname()` no cliente e, se quiser, `getpeername()` no cliente para imprimir os dados do servidor.

Wrappers a serem criados

Deve-se implementar funções envelopadoras (wrappers) para todas as chamadas da API de sockets e funções de sistema usadas no servidor e no cliente.

Por que criar Wrappers?

- Padronizar tratamento de erros: todas as chamadas de sistema (`socket()`, `bind()`, `accept()`, `fork()`, `read()`, `write()`, etc.) podem falhar. Se não tratarmos os erros, o programa pode se comportar de forma incorreta ou difícil de depurar.
- Evitar repetição de código: sem wrappers, cada chamada teria que repetir `if (...) { perror(...); exit(1); }`. Isso polui o código principal.
- Deixar o código mais legível, em vez de:

```
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
perror("socket");
exit(1);}
```

podemos simplesmente escrever:

```
int sockfd = Socket(AF_INET, SOCK_STREAM, 0);
```

Isso deixa o programa mais próximo de uma “linguagem de alto nível”.

- Facilitar manutenção: se no futuro quisermos que um erro seja tratado de outra forma (exemplo: logar em arquivo em vez de `perror`), basta alterar o wrapper, e não todas as ocorrências no código.

Convenção que deve ser utilizada:

- Nome da função original com a primeira letra maiúscula (`Socket`, `Bind`, `Listen`, `Accept`, `Fork`, `Close`, `Read`, `Write`).
- Em caso de erro, a função chama `perror()`, imprime a causa e encerra o processo com `exit(1)` (exceto em `Read`/`Write`, que retornam o valor para o programa decidir).

Exemplo de Wrapper implementado:

```
int Socket(int family, int type, int protocol) {
    int sockfd;
    if ((sockfd = socket(family, type, protocol)) < 0) {
        perror("socket");
        exit(1);
    }
    return sockfd;
}
```

Exemplo de uso no código

```
int listenfd = Socket(AF_INET, SOCK_STREAM, 0);
Bind(listenfd, (struct sockaddr*)&servaddr, sizeof(servaddr));
Listen(listenfd, BACKLOG);
```

Dessa forma, o código principal do servidor fica limpo, direto, e cada etapa da API de sockets é fácil de entender.

Em base à implementação, responda:

1. No servidor, por que inserir `sleep()` no filho antes de fechar `connfd` permite observar concorrência quando dois clientes se conectam quase ao mesmo tempo?
2. No trecho abaixo, explique por que o servidor segue escutando e por que os clientes permanecem conectados mesmo após os `Close`. Diga o propósito de cada `Close` e se algum está sobrando.

```
 for (;;) {
 connfd = Accept(listenfd, ...);
 if ((pid = Fork()) == 0) {
    Close(listenfd);
    doit(connfd);
    Close(connfd);
    exit(0);
}
Close(connfd);}
```

3. Ainda com base no trecho acima, é correto afirmar que os clientes “nunca” receberão FIN porque o servidor fica em LISTEN? Justifique.

4. Comprove com ferramentas do sistema operacional que os processos que manipulam as conexões são filhos do processo servidor original.
5. Após o encerramento de uma conexão, qual lado fica em `TIME_WAIT` e isso condiz com a sua implementação? Justifique.

Como compilar e testar rapidamente

```
gcc -Wall -Wextra -O2 -o server_http server_http.c
gcc -Wall -Wextra -O2 -o client_http client_http.c

# Terminal 1
./server_http 8080 10    # segura 10s cada conexão para evidenciar
concorrência

# Terminal 2
./client_http 127.0.0.1 8080

# Terminal 3
./client_http 127.0.0.1 8080

# Alternativamente, usando curl
curl -v http://127.0.0.1:8080/
```

Dicas úteis

- Para observar os processos filhos: `ps f | grep server_http` ou `pstree -p | grep server_http`.
- Para estados de conexão: `ss -tanp | grep :8080` ou `netstat -tanp | grep :8080`.
- `strace -f -e trace=network ./server_http 8080` ajuda a ver `accept`, `read`, `write`, `close`.