ISEL
Departamento de Engenharia Eletrónica e Telecomunicações e de Computadores

Licenciatura em Engenharia Informática, Redes e Telecomunicações

Sistemas Operativos

(*2ºTrabalho prático*)

Trabalho realizado por:

Nome: Diogo Cardoso	Nº 51474
Nome: Daniel Santos	Nº 51701
Nome: Paulo Magalhães	Nº 51702

Docente: Nuno António Afonso Cunha Oliveira

Sistemas Operativos
2024 / 2025 verão

12 de maio de 2025

Exercício 1

copy.c:

Nos testes realizados na máquina virtual, o copy com buffer de 1 e 64 teve de ser interrompido após 4 minutos de execução, sendo o tempo considerado indefinido. À medida que se aumenta o tamanho do buffer, o número de chamadas ao sistema operativo diminui, o que reduz significativamente o tempo de execução. Desta forma, a performance do copy aproxima-se da do fcopy, o que é o comportamento desejado. Por exemplo, com um buffer de 4096 bytes, o tempo de execução do copy foi de 1.638 segundos, enquanto o fcopy apresentou tempos entre 0.8 e 1.2 segundos.

```
Source     : filesrc          real    0m52,114s
Destination: filedst          user    0m0,290s
Buffer     : 128              sys     0m27,582s
Read calls 81921

Source     : filesrc          real    0m26,051s
Destination: filedst          user    0m0,141s
Buffer     : 256              sys     0m11,587s
Read calls 40961

Source     : filesrc          real    0m10,930s
Destination: filedst          user    0m0,055s
Buffer     : 512              sys     0m5,329s
Read calls 20481

Source     : filesrc          real    0m5,697s
Destination: filedst          user    0m0,025s
Buffer     : 1024             sys     0m2,414s
Read calls 10241

Source     : filesrc          real    0m1,638s
Destination: filedst          user    0m0,009s
Buffer     : 4096             sys     0m0,475s
Read calls 2561
```

fcopy.c:

Ao analisar os tempos de execução e os traces, verifica-se que a duração do fcopy é relativamente constante, independentemente do tamanho do buffer, com pequenas variações entre 0.8 e 1.2 segundos.

Além disso, é importante notar que o número total de chamadas read e write permanece constante, também independente do buffer, totalizando sempre 2603 chamadas no conjunto.

```
Source file     : filesrc          real  0m1,239s
Destination file: filedst          user  0m0,125s
Buffer size     : 1                sys   0m0,427s

Source file     : filesrc          real  0m0,983s
Destination file: filedst          user  0m0,014s
Buffer size     : 64               sys   0m0,314s

Source file     : filesrc          real  0m1,102s
Destination file: filedst          user  0m0,025s
Buffer size     : 128              sys   0m0,363s

Source file     : filesrc          real  0m0,800s
Destination file: filedst          user  0m0,007s
Buffer size     : 256              sys   0m0,244s

Source file     : filesrc          real  0m1,132s
Destination file: filedst          user  0m0,011s
Buffer size     : 512              sys   0m0,312s

Source file     : filesrc          real  0m0,961s
Destination file: filedst          user  0m0,019s
Buffer size     : 1024             sys   0m0,341s
```

Exercício 2

(a) em média 1200 ns Total elapsed time = 0.000012 s (average 0.12 us)
(b) em média 2500 ns Total elapsed time = 0.000025 s (average 0.25 us)
(c) em média 4160 ns Total elapsed time = 0.004116 s (average 41.16 us)
(d) em média 3438 ns Total elapsed time = 0.003438 s (average 34.38 us)
(e) em média 3470 ns Total elapsed time = 0.003470 s (average 34.70 us)

Os tempos médios obtidos refletem a complexidade de cada operação. A chamada de função é a mais rápida, pois ocorre inteiramente no espaço do utilizador. As chamadas de sistema são mais lentas devido à troca de contexto com o kernel. A criação de processos e a execução de programas são ainda mais custosas devido à alocação de recursos e ao carregamento de binários. Por fim, a criação de threads é mais eficiente do que a criação de processos, pois as threads compartilham recursos com o processo pai.

Exercício 3

Para 1000000000 termos:

Versão sequencial: 3.267s

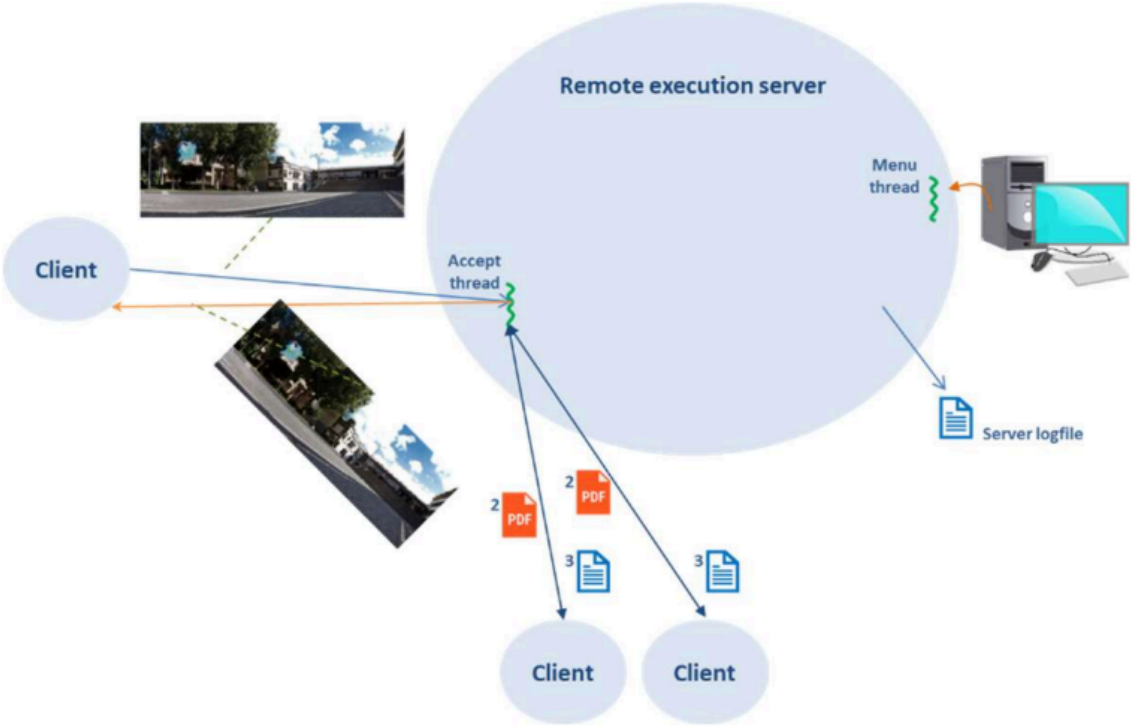
Versão baseada em múltiplos processos (5): 0.763s

Versão baseada em múltiplas tarefas (5): 0.840s

A versão com threads apresenta um tempo de execução ligeiramente maior em comparação à versão com múltiplos processos devido ao compartilhamento de memória entre as threads. Esse compartilhamento pode causar contenção de recursos e a necessidade de sincronização, o que introduz uma pequena sobrecarga. No entanto, a criação de threads é mais leve do que a de processos, o que ainda garante um desempenho próximo ao da versão com múltiplos processos.

Exercício 4

O exercício propõe o desenvolvimento de uma arquitetura Cliente/Servidor concorrente que permita a execução remota de programas no servidor, com base nos dados enviados pelo cliente. A comunicação entre cliente e servidor será realizada através de sockets stream, suportando tanto o domínio Internet (TCP/IP) como o domínio UNIX. O servidor deve ser estruturado de forma concorrente, atribuindo uma tarefa dedicada (por exemplo, um processo ou thread) a cada ligação estabelecida com um cliente, garantindo assim a capacidade de atender múltiplos pedidos em simultâneo de forma eficiente.



Organização da implementação e explicação de como foi feito

A implementação do sistema foi organizada em **oito ficheiros-fonte**, distribuídos em três diretorias principais: **Client**, **Server** e **Models**.

Models

Os ficheiros incluídos na diretoria Models são componentes auxiliares que abstraem funcionalidades essenciais como comunicação por sockets, logging e manipulação de I/O. Estes módulos são independentes da lógica principal da aplicação e podem ser reutilizados em diferentes contextos:

- **socket_utils.c**: Oferece um conjunto de funções auxiliares para criação, ligação e aceitação de sockets, tanto no **domínio TCP/IP** (Internet) quanto no **domínio UNIX** (sockets locais). Ele facilita a construção de servidores e clientes de forma robusta e reutilizável, com tratamento centralizado de erros.

- **log.c**: Implementa o **sistema de registo de eventos (logging)**. Suporta diferentes níveis de severidade (**INFO**, **ERROR**, **DEBUG**) e formata as mensagens com data e hora, tornando os registos úteis para depuração, monitorização e análise posterior. Os logs são escritos num ficheiro especificado durante a inicialização.
- **error.c**: Centraliza o **tratamento de erros do sistema**, oferecendo funções específicas para verificar chamadas do sistema e da biblioteca **pthread**. Também inclui verificação de erros em operações de sockets. Assegura uma terminação controlada da aplicação em caso de falha, evitando a duplicação da lógica de verificação ao longo do código e promovendo uma maior robustez e manutenção do sistema.
- **io.c**: Trata **a leitura e envio dos dados trocados entre cliente e servidor**, com base no protocolo textual definido. Inclui:
  - Leitura e separação do cabeçalho enviado e recebido pelo cliente (**read_client_header** e **parse_client_header**).
    - A função **read_client_header** lê o cabeçalho de uma mensagem de um socket até encontrar a sequência "\n\n", que indica o fim do cabeçalho. Para garantir precisão, o programa lê um byte de cada vez, permitindo detetar essa sequência exata sem consumir dados a mais. Após cada leitura, adiciona a terminação de string para manter o buffer válido.
  - Receção do conteúdo do ficheiro (**receive_file_content**) enviado após o cabeçalho.
  - Preparação e envio da resposta (**output do programa executado**) ao cliente (**send_program_output_to_client**), incluindo:
    - Escrita em ficheiro temporário.
    - Cálculo do tamanho do conteúdo a ser enviado (fstat).
    - Envio do novo cabeçalho e do conteúdo por blocos.
  - Criação do cabeçalho de resposta (**header_to_client**).
  - Execução segura do envio do conteúdo por pipe (**send_content_in_chunks**).
    - A função s**end_content_in_chunks** envia o conteúdo de um ficheiro por um pipe em blocos de tamanho fixo (FILE_SOCKET_BUFFER), garantindo que o envio é feito de forma controlada e eficiente. É ignorado o sinal SIGPIPE para evitar que o processo termine se o pipe estiver fechado. Em cada iteração, escreve um bloco e atualiza o total de bytes enviados. No fim, fecha o descritor da pipe para indicar que o envio terminou.
  - Preparação e limpeza da lista de argumentos para execução de comandos (**setArgumentsToList** e **freeArguments**).
  - Funções auxiliares para determinar extensão do resultado (**get_output_extension**) e nome de ficheiro final com a extensão correta (**get_output_file_with_extension**).

TCP/IP (Internet)

Sobre setsockopt e SO_REUSEADDR

Na função **tcp_server_socket_init**, o uso de **setsockopt**(..., **SO_REUSEADDR**, ...) permite que a aplicação reutilize a mesma porta imediatamente após ser encerrada, mesmo que ela esteja em estado **TIME_WAIT**. Sem esta opção, o sistema pode bloquear temporariamente a reutilização da porta, impedindo a criação de novos sockets que acabava por dificultar também o desenvolvimento deste programa.

- **tcp_server_socket_init(int serverPort)**
  Cria um socket TCP (**AF_INET**), define a opção **SO_REUSEADDR**, associa-o a uma porta específica **(bind)** e fica à espera de conexões (**listen**).

- **tcp_server_socket_accept(int serverSocket)**
  Aceita uma nova conexão a partir de um socket em estado **listen**. Retorna um novo descritor de socket que será utilizado para comunicar com o cliente.

- **tcp_client_socket_init(const char *host, int port)**
  Cria um socket e estabelece uma conexão com o servidor remoto no IP e porta especificada.

Sockets UNIX (Comunicação Local)

Sobre unlink

Na função **un_server_socket_init**, o **unlink(serverEndPoint)** remove qualquer ficheiro de socket pré-existente no caminho fornecido. Como os **sockets UNIX** são representados por ficheiros no sistema de ficheiros (ex.: /tmp/socket), esta remoção evita falhas na operação de **bind**, como o erro **"Address already in use"**, ao reiniciar o servidor.

- **un_server_socket_init(const char *serverEndPoint)**
  Cria um socket local (**AF_UNIX**) no caminho especificado, remove qualquer ficheiro de socket antigo utilizando **unlink**, associa o socket (**bind**) e fica à espera de conexões (**listen**).

- **un_server_socket_accept(int serverSocket)**
  Aceita uma nova conexão local através de um socket UNIX. Retorna um novo descritor de socket para comunicação com o cliente.

- **un_client_socket_init(const char *serverEndPoint)**
  Cria um socket local e conecta-se a um servidor no caminho especificado.

Server

É um processo concorrente baseado em múltiplas tarefas:

- **accept_thread**: Responsável por aceitar conexões de clientes **TCP** e **UNIX** em threads separadas. Cada função (**accept_thread_tcp** e **accept_thread_unix**) escuta novos clientes em seus respetivos sockets, cria uma nova thread para lidar com cada conexão e gerencia contadores de clientes com **mutex**. As threads são criadas de forma assíncrona e automaticamente liberadas após término, permitindo atendimento simultâneo de múltiplos clientes.

Sobre pthread_mutex_lock e pthread_mutex_unlock

Utiliza **pthread_mutex_lock** e **pthread_mutex_unlock** para proteger secções críticas do código onde há acesso ou modificação de variáveis compartilhadas (como os contadores de clientes), garantindo que **apenas uma thread por vez aceda a estes dados**.

- **menu_thread**: Implementa a interface de menu interativa do servidor, executada em uma thread separada. Permite ao utilizador listar o número de clientes conectados (TCP e UNIX), **Opção 1**, ou encerrar o servidor de forma controlada, **Opção 2**. Utiliza **pthread_mutex_lock** e **pthread_mutex_unlock** para garantir acesso seguro às variáveis globais compartilhadas (como os contadores de clientes e **server_running**), evitando condições de corrida durante leitura ou modificação desses dados.
- O servidor ao aceitar uma conexão de cliente, cria uma nova **thread** para o tratar.
- Lê o cabeçalho (**read_header_client**) enviado pelo cliente com os detalhes do ficheiro e do programa e separa-o (**parse_client_header**) da seguinte forma:

  RUN: \n
  ARGS: \n
  FILE: \n
  DIM: \n\n

- Cria dois pipes:
  - Um para enviar os dados do ficheiro ao programa (**stdin**).
  - Outro para ler a saída do programa (**stdout**).

- Gera um processo filho usando **fork()**, redireciona os pipes e executa o programa com **execvp()**.
- Envia o resultado de volta ao cliente com um novo cabeçalho tendo em atenção que tem uma nova dimensão que é corrigida com a função **send_program_output_to_client do io.c.**

Client

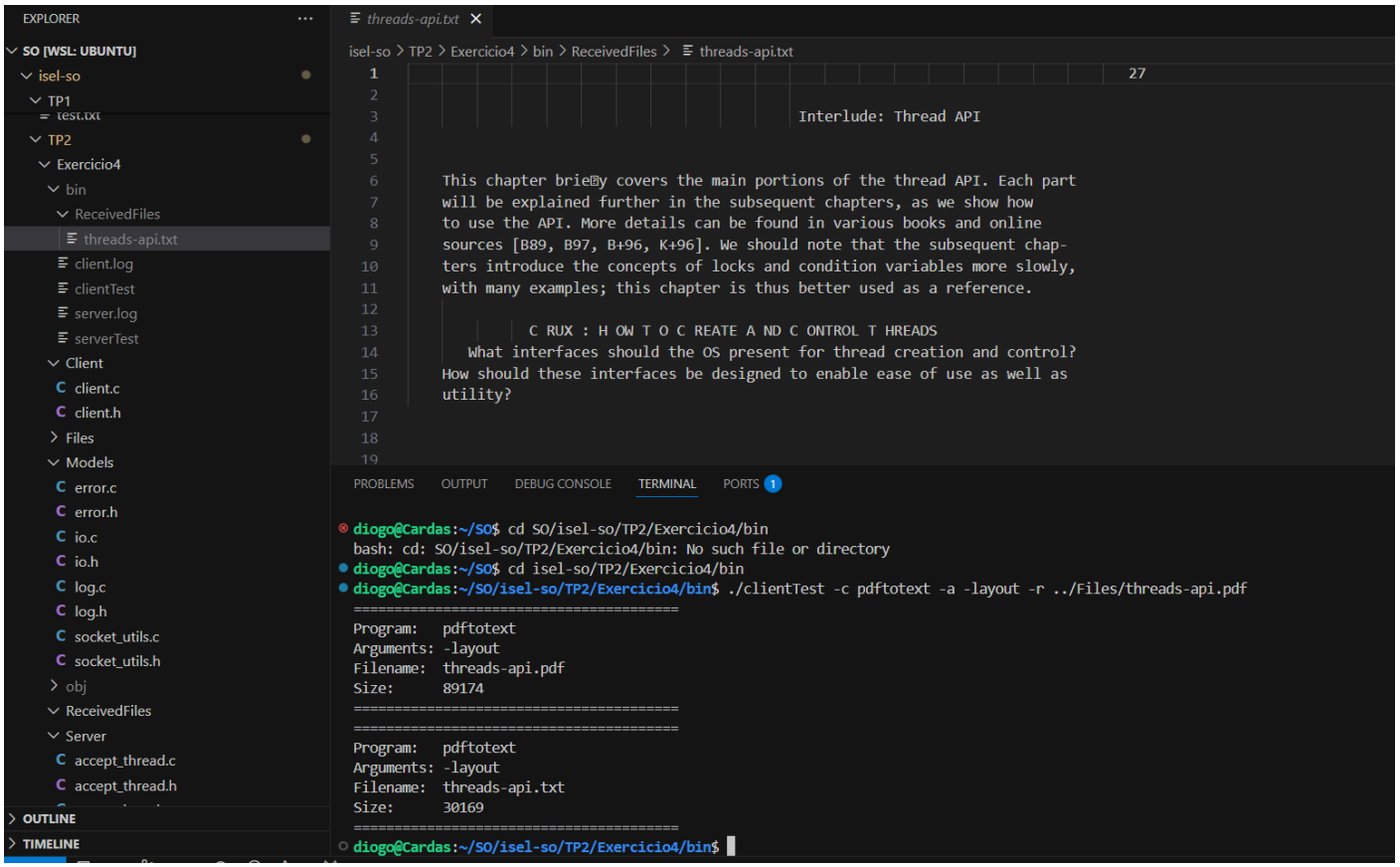
- Lê os argumentos da linha de comandos (host, porta, socket UNIX/TCP, modo receive, diretoria, programa, argumentos, etc.) e passa os argumentos para uma estrutura (**client_config_t**) , de modo a serem usados na execução da main.

- Envia um cabeçalho com informações do ficheiro para o servidor (nome, tamanho, programa e argumentos) de modo a que este possa executar o comando pedido. Envia o conteúdo do ficheiro em blocos através da função **send_content_in_chunks**.
- Se o modo -receive estiver ativo, espera receber o ficheiro processado e guarda-lo na diretoria especificada.

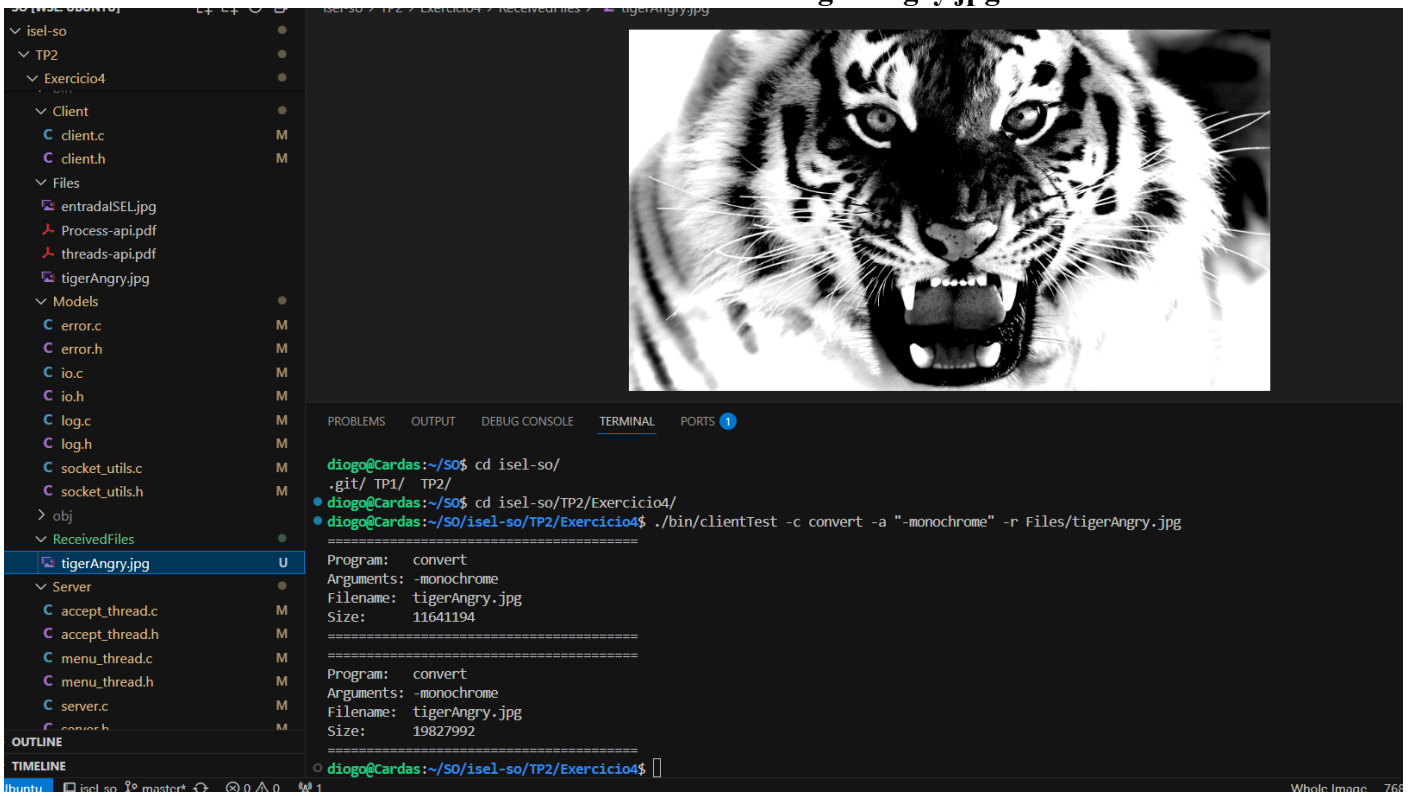
Era suposto ser desenvolvido um **cliente de teste** com capacidade para **estabelecer múltiplas conexões simultâneas**, permitindo realizar t**estes de stress no servidor**, porém não foi possível realizar este de forma **concorrente** e foi apresentada uma versão em que ocorre em **sequência** em que mandamos o **número de ligações na linha de comando** que o cliente vai apresentar e depois através do uso de um **ciclo for** ele faz o tratamento como é pedido no comando.

Testes do Client e Server

./clientTest -c pdftotext -a -layout -r ../Files/threads-api.pdf



./clientTest -c convert -a "-monochrome" -r ../Files/tigerAngry.jpg



II. Questões de escolha múltipla

1.

Para dividir o processamento por ações concorrentes de forma a maximizar a utilização de toda a capacidade de processamento do hardware.	Verdadeiro
Para poder executar dois programas (ficheiros executáveis) diferentes em concorrência e de uma forma mais rápida.	Falso
Para poder realizar operações I/O em simultâneo com outras operações num mesmo processo.	Verdadeiro
Para ter a execução de dois troços de código concorrentes com espaços de endereçamento separados num mesmo processo.	Falso

2.

Os sockets do domínio UNIX e os fifos são identificados através de um ficheiro especial no sistema de ficheiros.	Verdadeiro
O mecanismo de comunicação fifo (named pipe) apenas funciona entre processos com grau de parentesco.	Falso
Nos sockets stream a função bind serve para associar o socket a uma tarefa.	Falso
Os sockets são representados ao nível do núcleo do sistema operativo como um tipo de ficheiro e podem ser usados para o redireccionamento de I/O e a receção e envio de dados realizados através das funções de read() e write().	Verdadeiro

3.

<table>
<tr>
<td rowspan="2">

```
int main () {
  int s = tcp_serversocket_init(HOST, PORT);
  while (1) {
     int ns = tcp_serversocket_accept(s);
     handle_client(ns);
  }
  return 0;
}
```

</td>
<td>Servidor disponível através de um socket no domínio internet atendendo múltiplos clientes em sequência.</td>
<td>Verdadeiro</td>
</tr>
<tr>
<td>Servidor disponível através de um socket no domínio internet atendendo múltiplos clientes em concorrência.</td>
<td>Falso</td>
</tr>
<tr>
<td rowspan="2">

```
void `thHandleCient (void *arg) {
 int ns = *((int *)arg);
 handle_client(ns);
 return NULL;
}

int main () {
 int s = tcp_serversocket_init(HOST, PORT);
 pthread_t th;
 while (1) {
   int ns = tcp_serversocket_accept(s);
   int *ps = malloc(sizeof(int));
   *ps = ns;
   pthread_create(&th, NULL,
                        thHandleClient, ps);
   pthread_join(th);
 }
 return 0;
}
```

</td>
<td>Servidor disponível através de um socket no domínio internet atendendo múltiplos clientes em concorrência.</td>
<td>Falso</td>
</tr>
<tr>
<td>Servidor disponível através de um socket no domínio internet atendendo múltiplos clientes em sequência.</td>
<td>Verdadeiro</td>
</tr>
</table>