Projeto de Sistemas Operativos
2021-22
Enunciado do 2º exercício
LEIC-A/LEIC-T/LETI

O 2º exercício do projeto de SO pretende estender o sistema de ficheiros TecnicoFS com a capacidade de servir vários processos cliente de forma concorrente.

Para tal, o TecnicoFS deixará de ser uma simples biblioteca para passar a ser um processo servidor autónomo, ao qual diferentes processos clientes se podem ligar e enviar mensagens com pedidos de operações.

Ponto de partida

Para resolver o 2º exercício, os grupos devem descarregar o novo código base (tecnicofs_ex2.zip), disponibilizado no site da disciplina.

Este código base estende a versão original do TecnicoFS das seguintes maneiras:

a) As operações principais do TecnicoFS estão sincronizadas usando um único trinco (*mutex*) global. Embora menos paralela que a solução pretendida para o 1º exercício, esta solução de sincronização é suficiente para implementar os novos requisitos que compõem o 2º exercício.
b) São fornecidos 2 programas clientes que podem ser usados como testes iniciais para cada requisito descrito de seguida.
c) Para o requisito 2 (ver abaixo), é fornecido um esqueleto para o programa do servidor TecnicoFS (em `fs/tfs_server.c`) e um esqueleto da implementação da API cliente do TecnicoFS (na diretoria `client`).

Em vez do novo código base, também se permite que os grupos resolvam o 2º exercício sobre a solução que compuseram no 1º exercício. No entanto, esta opção não é necessária nem será valorizada na avaliação do 2º exercício.

O exercício é composto por 2 requisitos, que apresentamos de seguida.

Requisito 1: Função de destruição bloqueante

Pretende-se implementar a seguinte função:

- *int tfs_destroy_after_all_closed();*

A função *tfs_destroy_after_all_closed* é uma variante mais complexa da função *tfs_destroy*. Quando chamada, a função *tfs_destroy_after_all_closed* deve verificar se há algum ficheiro aberto. Caso haja pelo menos um ficheiro aberto, a função deve bloquear até que essa condição se deixe de verificar. Quando já não existirem ficheiros abertos, a função deve finalmente desativar o TecnicoFS (ou seja, fazer aquilo que a alternativa tfs_destroy faz).

Quaisquer chamadas a *tfs_open* que ocorram concorrentemente ou posteriormente ao momento em que a *tfs_destroy_after_all_closed* desativa o TecnicoFS devem devolver erro (-1). Só depois do TecnicoFS ser ativado de novo (por chamada a *tfs_init*) é que as chamadas a tfs_open devem voltar a ser permitidas.

Para implementar este requisito, devem ser evitadas soluções com espera ativa. Para tal, é aconselhado o uso de variáveis de condição (*pthread_cond*). Além disso, devem ser prevenidas situações de míngua ou interblocagem. Naturalmente, a implementação deve também ser *thread-safe*.

Experimente:

Corra o teste disponibilizado em *tests/lib_destroy_after_all_closed_test.c*. Assim que este requisito esteja assegurado, o teste deverá passar com sucesso.

Crie e experimente variações *multi-threaded* deste teste.
Por exemplo, componha um teste em que duas ou mais *threads* executam um ciclo em que, a cada iteração, abrem, lêem/escrevem e fecham ficheiros; no entanto, concorrentemente, a tarefa inicial chama *tfs_destroy_after_all_closed*.

Requisito 2: Arquitetura cliente-servidor

O TecnicoFS deve passar a ser um processo servidor autónomo, lançado da seguinte forma:

```
tfs_server nome_do_pipe
```

Quando criado, o servidor deve criar um *named pipe* cujo nome (*pathname*) é o indicado no argumento acima. É através deste *pipe* que os processos cliente se poderão ligar ao servidor e enviar pedidos de operações.

Qualquer processo cliente pode ligar-se ao *pipe* do servidor e enviar-lhe uma mensagem a solicitar o início de uma sessão. Esse pedido contém o nome de um *named pipe*, que o cliente previamente criou para a nova sessão. É através deste *named pipe* que o cliente receberá as respostas aos pedidos de operações enviados ao servidor no âmbito da nova sessão.

Ao receber um pedido de sessão, o servidor atribui um identificador único à sessão, designado *session_id*, e associa a esse *session_id* o nome do *named pipe* que o cliente indicou. De seguida, responde ao cliente com o *session_id* da nova sessão.

O servidor aceita no máximo S sessões em simultâneo, cada uma com um *session_id* distinto, sendo que *session_id* é um valor entre [0, S - 1], em que S é uma constante definida no código do servidor. Isto implica que o servidor rejeita pedidos de início de sessão que receba quando já tem S sessões ativas.

Uma sessão dura até ao momento em que i) o cliente envia uma mensagem de fim de sessão ou que ii) o servidor detete que o cliente está indisponível. Durante uma sessão, o processo servidor pode receber, através do seu *pipe*, mensagens a solicitar a execução de operações do TecnicoFS. Nas subsecções seguintes descrevemos a API cliente do TecnicoFS em maior detalhe, assim como o conteúdo das mensagens de pedido e resposta trocadas entre clientes e servidor.

API cliente do TecnicoFS

Para permitir que os processos cliente possam interagir com o TecnicoFS, existe uma interface de programação (API), em C, a qual designamos por API cliente do TecnicoFS. Esta API permite ao cliente ter programas que estabelecem uma sessão com um servidor e, durante essa sessão, invocar operações para aceder e modificar o sistema de ficheiros. De seguida apresentamos essa API.

As seguintes operações permitem que o cliente estabeleça e termine uma sessão com o servidor:

- `int tfs_mount(char const *client_pipe_path, char const *server_pipe_path)`
  Estabelece uma sessão usando os *named pipes* indicados em argumento.

O *named pipe* do cliente deve ser criado (chamando *mkfifo)* no nome passado no 1º argumento. O *named pipe* do servidor deve já estar previamente criado pelo servidor, no nome passado no 2º argumento.
Em caso de sucesso, o *session_id* associado à nova sessão terá sido guardado numa variável do cliente que indica qual a sessão que o cliente tem ativa neste momento; adicionalmente, ambos os *pipes* terão sido abertos pelo cliente (para ler e para escrever, respetivamente).
Retorna 0 em caso de sucesso, -1 em caso de erro.

- `int tfs_unmount()`
  Termina uma sessão ativa, identificada na variável respetiva do cliente, fechando os *named pipes* (cliente e servidor) que o cliente tinha aberto quando a sessão foi estabelecida e apagando o *named pipe* cliente.
  Retorna 0 em caso de sucesso, -1 em caso de erro.

Tendo uma sessão ativa, o cliente pode invocar as seguintes operações junto do servidor, cuja especificação é idêntica às operações homónimas do servidor:

- `int tfs_open(char const *name, int flags)`
- `int tfs_close(int fhandle)`
- `ssize_t tfs_write(int fhandle, void const *buffer, size_t len)`
- `ssize_t tfs_read(int fhandle, void *buffer, size_t len)`

E, finalmente, há também a seguinte operação:

- `int tfs_shutdown_after_all_closed()`
  Pede ao servidor que execute a operação *tfs_destroy_after_all_closed* e, de seguida, termine. O servidor deve retornar uma confirmação ao cliente (retornar 0) antes de terminar; ou -1 em caso de erro.

Diferentes programas cliente podem existir, todos eles invocando a API acima indicada (concorrentemente entre si). Por simplificação, devem ser assumidos estes pressupostos:

- Os processos cliente são *single-threaded*, ou seja a interação de um cliente com o servidor é sequencial (um cliente só envia um pedido depois de ter recebido a resposta ao pedido anterior).
- Os processos cliente são corretos, ou seja cumprem a especificação que é descrita no resto deste documento. Em particular, assume-se que nenhum cliente envia mensagens com formato fora do especificado, ou que envia pedidos no contexto de um *session_id* que não foi estabelecido por esse mesmo cliente.
- O servidor mantém apenas uma única tabela de ficheiros abertos, que é usada para servir as chamadas a *tfs_open* feitas por qualquer cliente.

Protocolo de pedidos-respostas

O conteúdo de cada mensagem (de pedido e resposta) deve seguir o seguinte formato:

Função da API cliente
`int tfs_mount(char const *client_pipe_path, char const *server_pipe_path)`

Mensagens de pedido e resposta
(char) OP_CODE=1 | (char[40]) nome do pipe do cliente (para respostas)
(int) retorno (conforme especificação)

Função da API cliente
int tfs_unmount()
Mensagens de pedido e resposta
(char) OP_CODE=2 | (int) session_id
(int) retorno (conforme especificação)

Função da API cliente
int tfs_open (char const *name, int flags)
Mensagens de pedido e resposta
(char) OP_CODE=3 | (int) session_id | (char[40]) name | (int) flags
(int) retorno (conforme especificação)

Função da API cliente
int tfs_close(int fhandle)
Mensagens de pedido e resposta
(char) OP_CODE=4 | (int) session_id | (int) fhandle
(int) retorno (conforme especificação)

Função da API cliente
ssize_t tfs_write (int fhandle, void const *buffer, size_t len)
Mensagens de pedido e resposta
(char) OP_CODE=5 | (int) session_id | (int) fhandle | (size_t) len | (char[len]) conteúdo de buffer
(int) retorno (conforme especificação)

Função da API cliente
ssize_t tfs_read (int fhandle, void *buffer, size_t len)
Mensagens de pedido e resposta
(char) OP_CODE=6 | (int) session_id | (int) fhandle | (size_t) len
(int) n° de bytes lidos (ou -1) | (char[n° bytes lidos]) conteúdo lido

Função da API cliente
ssize_t tfs_shutdown_after_all_closed ()

Mensagens de pedido e resposta
`(char) OP_CODE=7 | (int) session_id`
`(int) retorno (conforme especificação)`

Onde:

- O símbolo | denota a concatenação de elementos numa mensagem. Por exemplo, a mensagem de pedido associada à função tfs_close consiste num byte (char) seguido de dois inteiros.
- Todas as mensagens de pedido são iniciadas por um código que identifica a operação solicitada (*OP_CODE*). Com a exceção dos pedidos de *tfs_mount*, o *OP_CODE* é seguido do *session_id* da sessão atual do cliente (que deverá ter sido guardado numa variável do cliente aquando da chamada a *tfs_mount*).
- As *strings* que transportam os nomes de *pipes* são de tamanho fixo (40). No caso de nomes de tamanho inferior, os caracteres adicionais devem ser preenchidos com '\0'.

Implementação em duas etapas

Dada a complexidade deste requisito, recomenda-se que a solução seja desenvolvida de forma gradual, em 2 etapas que descrevemos de seguida.

Etapa 2.1: Servidor TecnicoFS com sessão única

Nesta fase, devem ser assumidas as seguintes simplificações (que serão eliminadas no próximo requisito):

- O servidor é *single-threaded*.
- O servidor só aceita uma sessão de cada vez (ou seja, S=1).

Experimente:

Corra o teste disponibilizado em *tests/client_server_simple_test.c* sobre a sua implementação cliente-servidor do TecnicoFS. Confirme que o teste termina com sucesso.

Construa e experimente testes mais elaborados que exploram diferentes funcionalidades oferecidas pelo servidor TecnicoFS.

Etapa 2.2: Suporte a múltiplas sessões concorrentes

Nesta etapa, a solução composta até ao momento deve ser estendida para suportar os seguintes aspetos mais avançados.

Por um lado, o servidor deve passar a suportar múltiplas sessões ativas em simultâneo (ou seja, S>1).

Por outro lado, o servidor deve ser capaz de tratar pedidos de sessões distintas (ou seja, de clientes distintos) em paralelo, usando múltiplas tarefas (*pthreads)*, entre as quais:

- A tarefa inicial do servidor deve ficar responsável por receber os pedidos que chegam ao servidor através do seu *pipe*, sendo por isso chamada a *tarefa recetora*.
- Existem também S tarefas trabalhadoras, cada uma associada a um *session_id* e dedicada a servir os pedidos que a tarefa recetora recebe com esse *session_id.* As tarefas trabalhadoras devem ser criadas aquando da inicialização do servidor.

A tarefa recetora coordena-se com as tarefas trabalhadoras da seguinte forma:

- Quando a tarefa recetora recebe um pedido contendo um dado *session_id*, a tarefa recetora deve entregar esse pedido à tarefa trabalhadora associada a esse *session_id*. A comunicação do pedido deve ser feita usando um *buffer* produtor-consumidor associado a cada tarefa trabalhadora. A sincronização do *buffer* produtor-consumidor deve basear-se em variáveis de condição (além de *mutexes*).
- Assim que uma tarefa trabalhadora execute um pedido que recebeu da tarefa recetora (através do *buffer* produtor-consumidor), a tarefa trabalhadora devolve a resposta diretamente ao cliente daquela sessão através do *pipe* de resposta associado à sessão.

Experimente:

Experimente correr os testes cliente-servidor que compôs anteriormente, mas agora lançando-os concorrentemente por 2 ou mais processos cliente.

Acrescente uma chamada a *tfs_shutdown_after_all_closed* num dos processos clientes e confirme que o servidor termina ordeiramente (num momento em que nenhum cliente tem qualquer ficheiro aberto).

Submissão e avaliação

A submissão é feita através do Fénix **até ao dia 4/fevereiro/2022 às 23h59**.

Os alunos devem submeter um ficheiro no formato *zip* com o código fonte e o ficheiro *Makefile*. O arquivo submetido não deve incluir outros ficheiros (tais como binários). Além disso, o comando *make clean* deve limpar todos os ficheiros resultantes da compilação do projeto.

Recomendamos que os alunos se assegurem que o projeto compila/corre corretamente no cluster *sigma*. Ao avaliar os projetos submetidos, em caso de dúvida sobre o funcionamento do código submetido, os docentes usarão o cluster sigma para fazer a validação final.

O uso de outros ambientes para o desenvolvimento/teste do projeto (e.g., macOS, Windows/WSL) é permitido, mas o corpo docente não dará apoio técnico a dúvidas relacionadas especificamente com esses ambientes.

A avaliação será feita de acordo com o método de avaliação descrito no site da cadeira.