Projeto de Sistemas Operativos
2020-21
2º enunciado
LEIC-A/LEIC-T/LETI

Este 2º exercício pretende estender a solução desenvolvida no 1º exercício com duas otimizações importantes e o suporte a uma operação nova. De seguida descrevemos cada novo requisito em detalhe. Assume-se que os alunos já leram o 1º enunciado do projeto.

Todos os requisitos abaixo devem ser suportados **sem recorrer a espera ativa**, tanto quanto possível. Além dos *mutexes* (*pthread_mutex_t*) e trincos de leitura-escrita (*pthread_rwlock),* a solução pode recorrer a variáveis de condição (*pthread_cond_t*). Não se permite o uso de outros mecanismos de sincronização.

Sugere-se fortemente que os requisitos sejam resolvidos pela ordem indicada no enunciado, pois tal permitirá uma melhor sincronização com a matéria de apoio que será abordada nas aulas teóricas e laboratoriais durante as semanas em que o projeto decorre.

Requisitos

1. Sincronização fina dos inodes

Como foi observado no 1º exercício, o desempenho paralelo da solução construída até aqui é relativamente desapontante.

O primeiro objetivo deste exercício é implementar uma nova estratégia de sincronização das estruturas de dados do TecnicoFS que permita maior paralelismo entre comandos que não têm conflitos (por exemplo, a criação de novo ficheiro numa dada diretoria e a pesquisa de outro ficheiro numa diretoria diferente). Esta nova estratégia deve substituir completamente as estratégias desenvolvidas no 1º exercício (i.e., *mutex* e *rwlock*); por isso o TecnicoFS deve deixar de receber o 4º argumento de linha de comandos (*synchstrategy*).

A ideia principal da nova estratégia é que, em vez de um trinco global, deve-se agora recorrer a trincos finos, **um por cada i-node na tabela de inodes do TecnicoFS**. Compete aos alunos decidir se os trincos finos da sua solução serão do tipo *pthread_mutex_t* ou *pthread_rwlock*.

Com esta nova abordagem, à medida que uma dada operação (create, lookup, delete, etc.) precisa aceder a mais um i-node, o trinco respetivo deve ser obtido. Desta forma, no final da operação, esta terá obtido um conjunto de trincos (um para cada i-node acedido durante a operação). Para assegurar a correção da operação, esse conjunto de trincos deverá ser libertado apenas no final da execução da operação

Mais em detalhe, a nova estratégia de sincronização deve executar concorrentemente :

- As operações que não alteram o estado do TecnicoFS (tal como já é assegurado no 1º exercício).

- Além disso, as operações que não alterem elementos comuns do TecnicoFS. Mais precisamente, de forma a garantir a consistência do sistema de ficheiros na presença de operações concorrentes, deverá ser garantida a seguinte propriedade: enquanto está a ser executada uma operação *op* que recebe como parâmetro o caminho *path*, nenhuma das directorias incluídas neste *path* pode vir a ser alterada (i.e., os conteúdos destas diretorias não devem poder mudar) por outras operações concorrentes. Deverá claramente também impedir que enquanto a operação *op* esteja a alterar o estado de alguma diretoria, nenhuma outra operação concorrente possa observar os efeitos (possivelmente incompletos) da execução de *op*.

Abaixo são apresentados alguns exemplos que ilustram combinações de operações para as quais deve ser admitida a execução concorrente e para quais a execução deverá ser necessariamente serializada:

Exemplo de operações que devem executar de forma concorrente:

1ª operação: c /a/b/c f

2ª operação: c /a/d/f f

Estas duas operações alteram o estado das directorias /a/b e /a/d, respetivamente, enquanto que apenas lêem (i.e., não mudam) o estado das directorias / e /a. Logo devem poder executar-se concorrentemente.

Exemplo de operações que não se podem executar de forma concorrente:

1ª operação: l /

2ª operação: c /a d

A primeira operação lê o estado da directoria "/" enquanto a segunda altera o estado da mesma.

2. Execução incremental de comandos

Como segundo objetivo deste exercício, pretende-se substituir a abordagem de carregamento/execução em duas fases sequenciais do 1º exercício (carregamento do ficheiro de *benchmark* em memória, seguido pela execução paralela dos comandos) por uma abordagem mais paralela.

Concretamente, pretende-se que, a partir do momento em que a tarefa inicial carrega um comando e o coloca no vetor, uma tarefa escrava que esteja livre possa imediatamente retirar e executar esse comando. Como consequência, os comandos serão executados incrementalmente *enquanto* o carregamento do ficheiro de entrada decorre em paralelo. Caso uma tarefa escrava esteja livre e o vetor de comandos não tenha nenhum comando, esta deve aguardar até que surja novo comando no vetor ou até que o final do ficheiro de entrada seja alcançado.

Ao contrário do 1º exercício, o carregamento não deve ser terminado quando o vetor de comandos se encontrar totalmente preenchido. Na nova solução, sempre que a situação anterior se verifique, a tarefa que preenche o vetor deve esperar até que novas posições no mesmo sejam libertadas pela execução dos comandos respectivos (por parte das tarefas escravas). Para exercitar esse novo comportamento, a dimensão do vetor de comandos deve passar a ser 10 entradas. Além disso, o vetor deve passar a ser manipulado como um vetor circular.

O tempo de execução (apresentado no *stdout* no final da execução) deve passar a ser medido desde o momento em que o carregamento do ficheiro de entrada começa. Isto representa uma diferença em relação ao 1º exercício, no qual o tempo só era contado após o carregamento ter terminado.

3. Nova operação: mover ficheiro ou diretoria

Deve ser suportada uma nova operação com a mesma semântica da operação *mv* em Linux. Esta operação recebe dois argumentos: o *pathname* (caminho de acesso) atual de um ficheiro/diretoria e o novo *pathname*. Como o nome indica, esta operação retira a entrada (*dirEntry*) com o *pathname* atual e insere uma nova entrada com o novo *pathname*. O *i-number* referido na nova entrada deve manter-se o mesmo que na entrada original.

Esta operação só deve ser executada caso se verifiquem duas condições no momento em que é invocada: existe um ficheiro/diretoria com o *pathname* atual e não existe nenhum ficheiro/diretoria com o novo *pathname*. Caso alguma destas condições não se verifique, a operação deve ser cancelada, não tendo qualquer efeito visível (a outras tarefas escravas que, concorrentemente, estejam a aceder ao TecnicoFS).

Por exemplo, suponha-se que se pretende mover o ficheiro "/dA/f1" para "/dB/f2". Se existir uma entrada "/dA/f1" mas também já existir uma entrada "/dB/f2" na diretoria, a tarefa escrava que tente executar a operação de mover deve ser capaz de detetar que a segunda condição não se verifica e cancelar a execução da operação, sem nunca remover "/dA/f1" da diretoria. Caso contrário, seria possível, num dado período, que outras tarefas escravas pesquisassem "/dA/f1" e, erradamente, não o encontrassem na diretoria.

Para simular invocações a esta nova operação, deverá ser suportado um novo tipo de comando, o comando 'm', no ficheiro de entrada. Este comando é seguido pelos dois argumentos referidos acima, por exemplo:

```
m /dA/f1 /dB/f2
```

Como observação final, a solução para este requisito deve ser compatível com o primeiro requisito deste exercício (sincronização fina dos inodes). Em particular, a solução deve, sempre que possível, prevenir situações de interblocagem ou míngua.

4. Shell script

Em complemento ao TecnicoFS, deve também ser desenvolvido um *shell script* para correr em Linux, chamado *runTests.sh*. Este *script* servirá para avaliar o desempenho do TecnicoFS quando executado com diferentes argumentos e ficheiros de entrada.

O *script* deve receber os seguintes quatro argumentos:

```
runTests inputdir outputdir maxthreads
```

Para cada ficheiro existente na diretoria *inputdir* (não considerando as subdiretorias), o *script* deve executar o TecnicoFS usando esse ficheiro como entrada, considerando diferentes números de tarefas.

Mais precisamente, para um dado ficheiro de entrada existente na diretoria *inputdir*, o TecnicoFS deve ser executado com *numthreads* variando entre 1 e *maxthreads* (por essa ordem).

Antes de executar cada caso descrito acima, o *script* deve imprimir a seguinte mensagem:

```
InputFile=nomeDoFicheiro NumThreads=númeroDeTarefas
```

No final da execução de cada caso, o *output* do programa deve ser filtrado de forma a que apenas seja impressa a seguinte linha:

```
TecnicoFS completed in duração seconds.
```

Em cada execução, o ficheiro de saída deve ser criado na diretoria indicada no argumento *outputdir*. O seu nome deve ser uma combinação do nome (relativo) do ficheiro de entrada usado nesta execução e do número de tarefas considerado, da seguinte forma:

nomeFicheiroEntrada-númeroDeTarefas.txt

Experimente

- Experimente executar os testes variando o número de tarefas *(numThreads)*. Caso já tenha desenvolvido o *shell script,* use-o para automatizar esta experiência. Com alguns testes, observará uma diferença muito grande no desempenho de cada alternativa. Como explica essa diferença?
- Instrumente o código para medir e imprimir (no *stdout*) o tempo total em que o carregamento de comandos (a partir do ficheiro) foi bloqueado devido ao vetor estar lotado. Experimentando diferentes dimensões do vetor (por exemplo: 1, 10, 20, 40 entradas, etc.) e número de tarefas escravas, observe como esse tempo varia em função desses parâmetros. Consegue encontrar uma boa explicação para o comportamento que observou?
- Experimente um ficheiro de entrada que, após criar duas diretorias, "/d1" e "/d2" e as preencher com muitos ficheiros, tenha uma longa sequência de operações 'm' (mover) que movem ficheiros de uma diretoria para a outra e vice-versa, intercalando essas operações 'm' em sentidos opostos. Em teoria, este conjunto de operações, quando executado em paralelo, poderá originar interblocagem. A sua solução consegue prevenir essa situação?

Nota: a resposta às perguntas acima não faz parte da avaliação do exercício.

Entrega e avaliação

Consultar o enunciado geral.