Projeto de Sistemas Operativos
2019-20
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 semáforos POSIX (*sem_t*). Não se permite o uso de outros mecanismos de sincronização.

Requisitos

1. Sincronização fina da diretoria

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 abordagem de sincronização da diretoria, substituindo o trinco global por uma estratégia com sincronização mais otimizada que permita maior paralelismo entre tarefas que não têm conflitos.

Para tal, a implementação atual (BST única) deve ser transformada numa estrutura alternativa que facilitará a implementação de uma paralelização mais fina. Mais precisamente, deve ser usada uma tabela de dispersão (*hash table*) em que cada posição da tabela aponta para uma BST distinta. Mais precisamente, a BST da entrada *i* da tabela de dispersão manterá o conjunto de entradas da diretoria cujo *hash* tem o valor *i*. Esta variante pode ser sincronizada com trincos mais finos, um por cada BST, permitindo um maior paralelismo que a solução com trinco global.

A dimensão da tabela de dispersão é especificada por um argumento de linha de comandos adicional (ou seja, o TecnicoFS passa a receber 4 argumentos de linha de comando), tal como de seguida:

```
tecnicofs inputfile outputfile numthreads numbuckets
```

A função de dispersão a utilizar é fornecida pelos docentes. Para tal, consultar no site da cadeira o código fonte fornecido.

Uma vez que a diretoria passa a ser implementada por múltiplas BSTs, a impressão do conteúdo final (no ficheiro de saída) deve passar a imprimir o conteúdo de cada BST da seguinte forma: primeiro é impresso o conteúdo da BST que guarda os ficheiros cujo valor de dispersão é 0, depois a BST associada ao valor de dispersão 1, e por aí fora.

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 este novo comportamento, a dimensão do vetor de comandos deve passar a ser 10 entradas.

Nota: o argumento de linha de comandos *numthreads* determina o número de tarefas escravas, excluindo a tarefa que carrega o vetor de comandos.

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: renomear ficheiro

Deve ser suportada uma nova operação que renomeia um ficheiro existente. Esta operação recebe dois argumentos: o nome atual de um ficheiro e o novo nome. Como o nome indica, esta operação retira a entrada com o nome atual da diretoria e insere uma nova entrada com o novo nome. O *i-number* deve manter-se o mesmo na nova entrada.

Esta operação só deve ser executada caso se verifiquem duas condições no momento em que é invocada: existe uma entrada com o nome atual e não existe nenhuma entrada com o novo nome. 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 à diretoria).

Por exemplo, suponha-se que se pretende renomear o ficheiro “f1” para “f2”. Se existir uma entrada “f1” mas também já existir uma entrada “f2” na diretoria, a tarefa escrava que tente executar a operação de renomear deve ser capaz de detetar que a segunda condição não se verifica e cancelar a execução da operação, sem nunca remover “f1” da diretoria. Caso contrário, seria possível, num dado período, que outras tarefas escravas pesquisassem “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 ‘r’, no ficheiro de entrada. Este comando é seguido pelos dois argumentos referidos acima, por exemplo:

```
r f1 f2
```

Como observação final, a solução para este requisito deve ser compatível com o primeiro requisito deste exercício (sincronização fina da diretoria). 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 numbuckets
```

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 e de entradas da tabela de dispersão.

Mais precisamente, para um dado ficheiro de entrada existente na diretoria *inputdir*, o TecnicoFS deve ser executado das seguintes formas (pela ordem indicada abaixo):

1. Na variante *nosync* com 1 tarefa apenas e 1 entrada na tabela de dispersão;
2. Na variante *mutex*, considerando os diferentes números de tarefas entre 2 e *maxthreads* (por exemplo, se *maxthreads* for 4, deve ser executado com 2, 3 e 4 tarefas). Em todas estas execuções, a tabela de dispersão deve ter o número de entradas especificado em *numbuckets.*

Nota: por simplificação, o script não deve testar a variante rwlock.

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)* e o número de entradas na tabela de dispersão (*numBuckets)*. Caso já tenha desenvolvido o *shell script,* use-o para automatizar esta experiência. Preste atenção a dois casos: 1) quando o número de tarefas é superior ao número de entradas na tabela de dispersão; 2) o caso oposto. Com alguns ficheiros de entrada, 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 tenha um número de operações 'r' (renomear) que movem ficheiros de uma entrada, *i*, da tabela de dispersão para outra, *j*, e operações 'r' que movem ficheiros no sentido oposto (de *j* para *i*). 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

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. Tal como no 1º exercício, a *Makefile* deve assegurar que, quando executado o comando *make* sem argumentos, sejam gerados 3 executáveis: `tecnicofs-nosync`, `tecnicofs-mutex` e `tecnicofs-rwlock` (com o mesmo significado definido no 1º exercício). Além disso, o comando *make clean* deve limpar todos os ficheiros resultantes da compilação do projeto.

O exercício deve **obrigatoriamente** compilar e executar nos computadores dos laboratórios, incluindo o *shell script*. 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 submissão é feita através do Fénix até ao dia 11/novembro às 23h59.

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

Os alunos devem consultar regularmente a plataforma piazza. É lá que serão prestados esclarecimentos sobre este enunciado.