Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Mestrado em Engenharia Informática

Sistema de Ficheiros Transaccional sobre FUSE

Nuno Lopes Luís (26323)

Lisboa
(2009)

Universidade Nova de Lisboa
Faculdade de Ciências e Tecnologia
Departamento de Informática

Dissertação de Mestrado

Sistema de Ficheiros Transaccional sobre FUSE

Nuno Lopes Luís (26323)

Orientador: Prof. Doutor João Lourenço

Dissertação apresentada na Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa para a obtenção do Grau de Mestre em Engenharia Informática.

Lisboa
(2009)

Para os meus adorados pais.

Agradecimentos

Embora uma dissertação seja, na sua índole, um trabalho individual, há contributos de natureza diversa que não podem deixar de ser destacados. Por essa razão, desejo expressar os meus sinceros agradecimentos:

- Ao suporte consedido pela Sun Microsystems and Sun Microsystems Portugal sobre o "Sun Worldwide Marketing Loaner Agreement #11497", pelo Centro de Informática e Tecnologias da Informação (CITI) e pela Fundação para a Ciência e Tecnologia (FCT/MCTES) no seu projecto de investigação Byzantium PTDC/EIA/74325/2006
- Ao Professor João Lourenço, meu orientador, pela sua disponibilidade, conselhos dados, ajuda e incentivos na realização deste trabalho e principalmente pelo alento e força que me conseguiu transmitir, sem os quais, estou seguro, não teria chegado ao fim.
- Ao grupo de investigação que integrei, *Software Transactional Memory* no seio do CITI, o qual me ajudou, realizando criticas e propondo ideias que se revelaram fundamentais. Em especial ao Diogo Sousa que me auxiliou na realização de testes de desempenho.
- Aos meus colegas e amigos, com os quais convivi e trabalhei, e que me motivaram e ajudaram ao longo do tempo. No fundo a amizade é o que fica, por isso da vossa simpatia, nunca mais me esquecerei: Ana Lameira, João Soares, David Navalho, Bruno Félix, Simão Mata, Pedro Sousa, Pedro Bernardo, João Cartaxo, Rui Domingues, Nuno Boavida, Ricardo Silva, David Costa, André Mourão, Danilo Manmohanlal, Emanuel Couto, Filipe Grangeiro, Sofia Cavaco, Gonçalo Martins, Nuno Silva, Marta Cristovão, Luís Oliveira, Pedro Amaral, João Rodrigues, Hélio Dolores, João Araújo, Joana Matos, Ricardo Martins, Inês Cristovão, Paulo Alexandre, Pedro Gomes ...
- Aos meus pais, por todo o suporte, compreensão e incentivo, dado de forma continuada, no desenrolar dos meus estudos e pela excitação e orgulho com que reagiram aos meus resultados académicos, pois sempre acreditaram em mim.
- Aos vários professores que tive ao longo da minha formação académica, que muito além do conhecimento partilhado, ajudaram-me a delinear um caminho académico.
- A todos aqueles que de forma, directa ou indirectamente, contribuíram para realização deste trabalho.

A todos, o meu muito obrigado !

Sumário

Com o surgimento e generalização do uso de microprocessadores com múltiplos núcleos (*multi-cores*), verifica-se um interesse crescente pela programação concorrente e, em particular, pela programação paralela, tanto pela comunidade académica, como pela indústria de desenvolvimento de software. Contudo, o desenvolvimento de programas concorrentes é difícil, em parte devido à complexidade inerente acrescida destes programas. Os mecanismos de controlo de concorrência que mais frequentemente são usados nos programas concorrentes apresentam um nível de abstracção baixo, sendo portanto de difícil utilização e dando origem a muitos erros de sincronização e controlo de concorrência.

O modelo transaccional é utilizado com sucesso no contexto dos sistemas de bases de dados há muitos anos. Estes sistemas permitem que múltiplos clientes acedam à bases de dados em concorrência, garantido que esses acessos beneficiam do conjunto de propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade). Estas propriedades das transacções permitem o desenvolvimento de aplicações que, apesar de acederem em concorrência ao repositório de informação, beneficiam de uma semântica essencialmente sequencial, mais previsível e fácil de usar.

Os sistemas de bases de dados podem ser usados para guardar grandes quantidades de informação de forma persistente e com suporte para processamento de transacções. No entanto, o acesso a estes sistemas é feito através de uma interface específica que requer software adicional, impondo assim limitações ao seu uso. Por outro lado, os sistemas de ficheiros estão disponíveis em praticamente todos os sistemas computacionais, sendo acessíveis através de uma interface bem definida e normalizada e utilizados com frequência pela maioria das aplicações para guardar os seus dados de forma permanente. No entanto, o controlo de concorrência em sistemas de ficheiros obtém-se através de uma interface com baixo nível de abstracção e funcionalmente limitada, difícil de usar a tendencialmente causadora de erros de utilização.

Nesta dissertação propõe-se a arquitectura e desenho de um sistema de ficheiros

que suporta o modelo transaccional para controlo de concorrência. Descreve-se também uma implementação do sistema de ficheiros proposto sobre o FUSE, uma infraestrutura base para implementação de sistemas de ficheiros. Neste novo sistema de ficheiros transaccional, as transacções podem ser iniciadas pelas aplicações de forma explícita ou implícita. No primeiro caso, o programador indica explicitamente o início e fim de um conjunto de acessos ao sistema de ficheiro que deverão ser tratados como uma transacção (bloco transaccional). No segundo caso, necessário para garantir a compatibilidade com aplicações e sistemas legados, os blocos transaccionais serão delimitados implicitamente pelas operações clássicas de abertura (`open`) e fecho (`close`) de ficheiros. O sistema propostos e implementado foi ainda avaliado nas perspectivas de funcionalidade e desempenho. A primeira para garantir que a semântica transaccional estava a ser correctamente suportada e disponibilizada às aplicações. A segunda para melhor permitir avaliar a penalização no desempenho resultante do suporte transaccional no sistema de ficheiros.

Palavras-chave: Sistema Transaccional, Sistema de Ficheiros, Sistema de Ficheiros Transaccional, FUSE, Controlo de Concorrência.

Summary

With the emergence and widespread use of microprocessors with multiple cores, there is a clear trend in a growing interest for concurrent programming, and in particular for parallel programming, by both the academic community and the industry of software development. However, the development of concurrent programs is difficult, partly because of the increased inherent complexity of these programs. The mechanisms for concurrency control most commonly used have a low level of abstraction and are therefore difficult to use and error prone.

The transaction model has been used successfully in the context of databases systems for many years. These systems allow multiple clients to access a single database concurrently, with such accesses verifying the ACID properties (Atomicity, Consistency, Isolation and Durability). These properties from transactions allow the development of applications that, although accessing concurrently the data repository, have a *near sequential* semantics, more predictable and easier to use.

The databases systems can be used to persistently store large amounts of data with support for transactional processing. However, access to these systems is done through a specific database interface, thus imposing limitations on their use. Moreover, file systems are accessible through a well defined and standardized interface and are frequently used by many applications to store data permanently. However, concurrency control in filesystems is achieved with a very limited set of low level services, limiting the functionality of applications and its capabilities to exploit concurrency in access to data in the file system.

This dissertation aims at proposing the architecture and design of a transactional filesystem and describes its implementation over the FUSE infrastructure. In this new file system, transactions may be initiated by applications either explicitly or implicitly. In the former, the programmer explicitly specified the transaction boundaries that enclose a set of accesses to the file system. In the latter, aiming at ensuring compatibility with legacy systems, transactional blocks are defined implicitly by the classical

file system operations of opening and closing files. The implementation of the proposed system was also assessed in the perspectives of functionality and performance. Functional assessment aimed at ensuring that the transactional semantics is properly supported and available to applications. Performance evaluation aimed at assessing the performance penalty introduced by the transactional support in file system.

Keywords: Transaction System, File System, Transaction File System, FUSE, Concurrency Control

Conteúdo

Lista de Figuras

Lista de Tabelas

1

Introdução

A capacidade de processamento dos computadores tem, nos últimos anos, evoluído num sentido diferente do que até então. O aparecimento dos processadores com múltiplos núcleos (*multi-core*), com suporte para a execução de programas em paralelo, veio colocar a programação concorrente no plano actual do desenvolvimento de aplicações informáticas. Por esse facto o tema da programação concorrente tem sido recentemente alvo de muita atenção por parte da comunidade científica. O desenvolvimento de aplicações num contexto de concorrência ainda é algo difícil, pois para lidar com a concorrência são normalmente utilizados mecanismo de baixo nível, como os *locks* ou semáforos, que embora sejam eficientes apresentam múltiplos problemas e limitações.

Neste contexto e como forma de facilitar o desenvolvimento de aplicações em ambientes concorrentes, está a surgir um renovado interesse pelos sistemas transaccionais, que inicialmente surgiram no contexto das bases de dados e que actualmente já chegam a outros domínios, como é o exemplo da memória. A noção de transacção oferece um modelo de abstracção, com uma semântica bem definida, onde uma sequência de instruções é executada de forma atómica, isolada, consistente e durável.

Contudo existem operações difíceis de incorporar dentro de um contexto transaccional, na medida em que estas operações podem ser irreversíveis, o que impede a cumprimento de atomicidade e isolamento das transacções. Um exemplo deste tipo de operações são as operações de I/O, como o escrever num terminal. No entanto, as operações sobre um sistema de ficheiros podem ser utilizadas dentro de um contexto transaccional, desde que o sistema de ficheiros faça a gestão/controlo de conflitos e permita a recuperação de uma versão anterior, permitindo o cumprimento da atomici-

dade e outras propriedades.

1.1 Contexto e motivação

Muitas das aplicações informáticas lidam com grandes quantidades de informação que, por várias razões, não pode permanecer em memória primária. Isto pode dever-se ao facto da memória ser pequena demais ou porque a aplicação necessitar de preservar os dados de forma persistente. Essa informação tem, portanto, de ser mantida em dispositivos de memória secundária não volátil como por exemplo discos rígidos ou cd-roms.

Os sistemas de ficheiros oferecem uma interface muito conveniente que permite às aplicações lidar com grandes quantidades de dados e abstraírem-se das especificidades dos suportes físicos onde a informação é guardada. Os sistema de ficheiros definem a noção de ficheiro como um agregado de informação relacionada identificada por um nome, que depois é mapeado do dispositivo físico subjacente.

Um exemplo de padronização nos serviços de acesso ao sistema de ficheiros muito conhecido é o POSIX 1003.1 [POS92]. O POSIX é uma normalização historicamente derivada do sistema UNIX, que define um conjunto de interfaces que devem de ser oferecidas pelo sistema operativo às aplicações, como por exemplo, a gestão de processos, funções da biblioteca C e acesso a ficheiros e directorias. Esta normalização é respeitada, na sua totalidade ou em larga parte, por muitos sistema operativos, como por exemplo, Mac OS X, Solaris, BSD Unix e Linux, o que permite a sua generalizada utilização por parte das aplicações, inclusive por sistemas embutidos, com recursos limitados.

Contudo, nos tempos actuais, os programas estão sujeitos a novos desafios, devidos ao aumento do nível de concorrência e paralelismo, motivado em grande parte pelas novas arquitecturas. Assim, o acesso a recursos partilhados, como é o caso dos os ficheiros, apresenta novos desafios.

1.2 Identificação do problema

O desenvolvimento de aplicações, que precisem de guardar informação em ficheiros, tem de incorporar muitas vezes a implementação de código com a possível concorrência e situações de falha. Este código previne situações onde a consistência dos dados se perde, onde existem conflitos no acessos aos dados e onde existe a necessidade de recuperar o estado anterior. Tal funcionalidade não é garantida pelo sistema de ficheiros.

Por exemplo, considere-se uma aplicação na qual se encontra aberto um projecto de larga dimensão e disperso por diversos ficheiros. Quando for necessário gravar as alterações realizadas ao projecto, irá ser realiza uma sequência de operações de escrita para o sistema de ficheiros (uma ou mais para cada ficheiro). No entanto, se no decorrer dessas operações ocorrer uma falha, como seja a falta de energia, todo o projecto pode ficar num estado inconsistente e isso levar à perda total ou parcial de informação. A consistência pode ser perdida pelo simples facto de alguns dos ficheiros do projecto incorporaram as novas alterações enquanto que outros o não fazem.

Outra dificuldade no sistemas de ficheiros é a forma como lidar com os acessos concorrentes a um mesmo ficheiro. Uma situação frequente passa por utilizar *locks* como mecanismo de controlo de concorrência, só que este mecanismo é limitativo quanto ao nível de concorrência que permite e não é de fácil utilização podendo inclusive levar a situações de *deadlock*. Outra alternativa para lidar com a concorrência no acesso a informação é a utilização de bases de dados, só que estas apresentam outra forma de organização e metodologia de acesso, que pode não ser tão conveniente como um sistema de ficheiros, para além de não estarem disponíveis em todos os sistemas.

1.3 Solução proposta

Para dar resposta aos problemas apresentados, foi desenhado um sistema que disponibiliza o modelo transaccional para as aplicações que trabalham com um sistema de ficheiros respeitando a API definida na norma POSIX 1003.1. Esta disponibilização é feita por duas forma distintas.

Um primeira forma consiste na disponibilização de um conjunto de serviços para delimitar os acessos a ficheiros que segue o modelo transaccional. O que implica, por um lado, que as aplicações que desejem obter garantias transaccionais, têm de refazer toda a sua implementação mas, por outro lado, oferece ao programador maior controlo sobre o comportamento das mesmas.

Numa segunda forma, é oferecido às aplicações garantias de acessos transaccionais a ficheiros de forma transparente contabilizando o número de ficheiros abertos para cada aplicação. O inicio de uma transacção ocorre quando o contador transita de 0 para 1, e a transacção termina quando esse contador transita de 1 para 0. Isto permitira garantir as propriedades transaccionais sobre todas as operações realizadas entre o primeiro *open* e o último *close*. Esta forma pode ser mais limitativa nas possibilidade que oferece às aplicações, mas permite uma fácil utilização do sistema e suporte para sistemas legados.

1.4 Principais contribuições

Realizou-se um estudo de assuntos e sistemas relacionados com sistemas transaccionais e sistemas de ficheiros. Nomeadamente uma recolha e avaliação de sistema de ficheiros que já ofereçam algum tipo de suporte transaccional.

Concebeu-se um sistema de ficheiros que assume a responsabilidade de gestão de concorrência e conflitos. Desta forma, é possível delegar no próprio sistema de ficheiros a capacidade de oferecer as propriedades de atomicidade, consistência, isolamento e durabilidade.

Desenhou-se um sistema de ficheiros que permite às aplicações recorrer a noção de transacção aquando da manipulação de ficheiros. Para tal oferece-se duas formas de operação, uma que implica a delimitação de transacções explicita por parte do programador, e uma segunda que o sistema infere as fronteiras de uma transacção, facilitando assim o trabalho do programador.

Implementou-se o sistema de ficheiros desenhado recorrendo a utilização do *framework* FUSE, com o objectivo de, por um lado, facilitar o desenvolvimento do novo sistema de ficheiros, e outro de permitir uma maior portabilidade e estabilidade do mesmo.

Avaliou-se a implementação do sistema de ficheiros desenvolvido, permitindo retirar ilações e estabelecer comparações com outros sistemas de ficheiros. Nomeadamente qual o impacte da gestão transaccional no desempenho do sistema de ficheiros.

1.5 Organização do Documento

Depois deste capitulo introdutório, o restante documento está organizada da seguinte forma: o capitulo 2 apresenta uma análise de trabalho relacionado com os sistema transaccionais e de ficheiros; no capitulo 3 é feita a descrição da solução, da sua arquitectura e desenho; no capitulo 4 é apresentada a implementação realizada do sistema desenvolvido.; no capitulo 5 é apresentada a validação da implementação alcançada; e por fim, no capitulo 6, são expostas as conclusões desta dissertação.

2

Trabalho relacionado

2.1 Sistemas transaccionais

Uma transacção consiste, de uma forma lógica, em uma unidade de trabalho composta por operações de acesso e actualização a dados. Uma transacção realiza a transição de um sistema de informação de um estado coerente para outro estado coerente. A sequência de acções que compõem uma transacção, definida geralmente numa linguagem de alto nível, está delimitada por marcas de início e fim. A transacção pode ter como resultado um de dois valores, ou sucesso ou falha.

Uma transacção respeita um conjunto de propriedades comuns nos sistema de transacções, que são conhecidas como propriedades ACID [Gra81, HR83] do acrónimo de Atomicidade, Consistência, Isolamento e Durabilidade (do inglês *Atomicity*, *Consistency*, *Isolation*, *Durability*). Estas propriedades são apresentadas em [TvS06] como:

Atomicidade: todas as operações que constituem uma transacção, ou são aplicadas como um todo ou nenhuma é aplicada. Isto dá uma visão da transacção como sendo indivisível, atómica, embora esta seja na realidade composta por uma sequência de operações.

Consistência: a transacção não viola os invariantes do sistema (as regras ou restrições aos dados). Esta noção de consistência depende sempre da lógica dos dados e do sistema. Mas esta propriedade refere-se à transacção como um todo, por isso, no decorrer de uma transacção, a consistência de dados pode não ser sempre respeitada. No entanto, o importante é que no final da transacção a consistência seja novamente reposta.

Isolamento: quando existem duas ou mais transacções a ser executadas concorrentemente, os seus efeitos têm de ser isolados. Deve de ser possível entender o comportamento de uma transacção sem ter em conta os possíveis efeitos da concorrência.

Durabilidade: uma vez dada a confirmação de sucesso da transacção, os efeitos desta devem de ser tornados persistentes, a fim de manter a informação, mesmo em caso de falha do sistema.

Com a execução sequencial de um conjunto de transacções é possível garantir que o sistema está sempre num estado coerente, pois cada transacção parte de um estado coerente para outro estado coerente. Contudo, por necessidade inerentes ao desempenho, é necessária uma execução concorrente de transacções, em que se permite o entrelaçamento temporal das operações individuais de múltiplas transacções. A introdução de concorrência nas transacções traz consigo a necessidade de isolamento a fim de garantir um determinado nível de consistência.

Neste contexto define-se um ordenamento como uma lista de acções, sendo estas de leitura, escrita e finalização (*commit* ou *abort*), constituída por todas as acções pertencentes a n transacções, onde as acções aparecem na lista na mesma ordem em que foram executadas nas transacções. Um ordenamento representa uma possível sequência da execução das acções das transacções. Se a ordem pela qual as acções aparecem no ordenamento for igual a uma execução sequencial dessas transacções, esse ordenamento diz-se sequencial.

Um ordenamento serializável, sobre um conjunto de transacções, define-se como um ordenamento que produz os mesmos resultados que um ordenamento sequencial. Ou seja, o estado do sistema resultante da execução de um ordenamento serializavel é equivalente à execução de um ordenamento sequencial.

2.1.1 Concorrência e isolamento

É desejável que um sistema tenha a capacidade de executar várias transacções de forma concorrente . Por um lado permite-se um aumento da taxa de atendimento (*throughput*), pois enquanto uma transacção está à espera que os dados sejam lidos outra pode ser processada. Por outro lado, é possível melhorar o tempo de resposta, pois com o entrelaçamento de uma transacção de longa duração T_1 com outra de curta duração T_2, vai-se permitir que T_2 termine mais rapidamente do que no caso em que um ordenamento sequencial coloque a T_2 depois da T_1.

Só que a introdução de concorrência traz consigo o problema de potenciais conflitos. Em geral duas acções concorrentes sobre o mesmo objecto estão em conflito se

pelo menos uma delas for de escrita. Mais especificamente os conflitos entre a duas transacções T_1 e T_2 podem ser descritos pela forma como as suas acções conflituam: escrita-leitura, leitura-escrita e escrita-escrita [RG02].

Leitura de dados não confirmados (escrita-leitura): este primeiro tipo de problema resulta da situação em que uma transacção T_2 poder ler um objecto cujo valor tenha sido alterado por outra transacção T_1 que ainda não realizou *commit*. Tal leitura é denominada de *dirty read*. Um exemplo deste tipo de conflito é apresentado na Figura 2.1, onde $R(X)$ e $W(X)$ denota a uma leitura e uma escrita sobre o objecto X respectivamente.

T1	T2
R(A)	
W(A)	
	R(A)
	W(B)

Figura 2.1: Ordenamento com leitura de dados não confirmados

Neste caso a transacção T_2 está a ler um valor de A escrito por T_1 que ainda não realizou *commit*, tratando-se portanto de um valor ainda não confirmado e que será inválido caso T_1 aborte.

Leitura não repetível (leitura-escrita): um segundo tipo de conflito resulta do caso em que uma transacção T_2 tem a possibilidade de alterar o valor de um objecto que tenha sido lido por T_1 enquanto esta ainda está em execução. Assim se T_1 tentar ler novamente o valor do objecto vai obter um resultado diferente do da sua primeira leitura. A esta situação dá-se o nome de *unrepeatable read*. Um exemplo deste tipo de conflito é apresentado na Figura 2.2.

T1	T2
R(A)	
W(A)	
	R(A)
W(A)	
	W(B)
	R(A)

Figura 2.2: Ordenamento com leitura não repetível (escrita-escrita)

Neste caso a transacção T_2 está a ler um valor de A que mais tarde é alterado por T_1, sendo que uma segunda leitura por parte de T_2 vai ter um valor

diferente, o que não faz sentido na visão de T_2, pois viola a propriedade do isolamento.

Reescrita de dados não confirmados(escrita-escrita): esta terceira situação ocorre se uma transacção T_2 puder reescrever o valor de um objecto que já tenha sido modificado anteriormente por T_1 enquanto esta ainda esta em execução. A esta situação dá-se o nome de *lost update*. Um exemplo deste tipo de conflito é apresentado na Figura 2.3.

T1	T2
W(A)	
	W(A)
R(A)	

Figura 2.3: Ordenamento com reescrita de dados não confirmados

Neste caso a transacção T_1 está a escrever o valor de A que mais tarde é alterado por T_2. Assim uma posterior leitura por parte de T_1 vai devolver um valor diferente daquele que ela própria tinha escrito.

Controlo de concorrência

Para permitir garantir que a introdução de concorrência num sistema transaccional não afecte o nível de isolamento das transacções, podem ser utilizados diversos mecanismos. Alguns deste mecanismos irão ser descritos e analisados de seguida.

Locks Os *locks* são um mecanismo geral de sincronização, que permite evitar conflitos de acesso a objectos partilhados, permitindo que os dados sejam acedidos por uma entidade de cada vez. A utilização de *locks* leva a que uma transacção que pretenda realizar uma operação sobre um objecto que conflitue com outra operação já realizada por outra transacção sobre o mesmo objecto, é obrigada a aguardar até que esta ultima transacção liberte o objecto partilhado. Contudo esta situação pode ser optimizada fazendo a diferenciação de *locks* de acordo com os tipos de acesso:

- ***Lock* de Leitura**, caso as acções associadas sejam exclusivamente de leitura. Este tipo de *lock* poderá, portanto, ser partilhado por um conjunto de transacções que estejam a realizar apenas operações de leitura sobre o objecto.
- ***Lock* de escrita**, caso as acções associadas incluam operações de escrita. Neste caso o *lock* é exclusivo a uma transacção, pelo que fica garantido o isolamento entre as transacções concorrentes.

O sistema de gestão de *locks* é responsável por gerir uma fila de espera para cada recurso, na qual coloca as transacções de acordo com os conflitos possíveis (ver Tabela 2.1), assim quando uma transacção origina um conflito, esta é bloqueada.

	Tipos de *locks*		
	Livre	Leitura	Escrita
Pedido de leitura	*Ok*	*Ok*	*Conflito*
Pedido de escrita	*Ok*	*Conflito*	*Conflito*

Tabela 2.1: Existência de conflitos de acordo com o tipo do *lock* e da acção

Um propriedade importante dos *locks* é a sua granularidade. A granularidade (ou grão) é a medida da quantidade de dados que um *lock* está a proteger. De uma forma geral, um grão grosso dá origem um pequeno número de *locks*, em que cada *lock* protege uma grande quantidade de dados. Um grão grosso origina um menor *overhead* na gestão dos *locks* mas, por reduzir a concorrência no sistema, limita o desempenho devido à contenção no acesso aos dados partilhados.

No caso de grão fino, que resulta num grande número de *locks*, em que cada *lock* protege uma pequena quantidade de dados, obtêm-se um maior *overhead* na gestão dos *locks*. No entanto o nível de contenção reduz-se, aumentado o desempenho global do sistema e a probabilidade de ocorrência de *deadlocks*.

Este mecanismo de *locks* permite resolver de forma eficaz o problema dos conflitos, apresentando no entanto alguns problemas, de entre os quais se salienta o *deadlock*. Um *deadlock* é uma situação de impasse devido a necessidade de recursos bloqueados mutuamente ou em ciclo. Um *deadlock* ocorre, por exemplo, quando uma transacção T_1 adquire um *lock* exclusivo sobre A, enquanto que T_2 adquire um *lock* exclusivo sobre B depois T_1 pede acesso para escrita a B e T_2 requer acesso para escrita a A. Assim T_1 fica à espera de T_2 e T_2 fica à espera de T_1, resultando num *deadlock*.

Uma forma simples de resolver este problema é o recurso à temporização (*timeouts*), onde se assume que uma transacção que esteja bloqueada à espera de um recurso durante um determinado período de tempo (pre-definido) está numa situação de *deadlock* e é abortada.

Um protocolo que é muitas vezes utilizado para a gestão de *locks* é o *Two Phase Locking* (2PL). O 2PL divide a gestão de *locks* no decorrer de uma transacção em duas etapas consecutivas:

Fase de obtenção de *locks*: onde um processo realiza a aquisição de *locks* e não realiza nenhuma libertação.

Fase de libertação de *locks*: fase na qual se procede à libertação dos *locks*, não

sendo possível adquirir novos *locks* até todos terem sido libertados e a transacção terminar.

Embora exista como grande vantagem a garantia de que qualquer ordenamento produzido que respeite o protocolo 2PL é um ordenamento serializavel [EGLT76], este restringe consideravelmente o nível de concorrência alcançável e não elimina a existência de *deadlocks*.

Estampilhas Outro mecanismo de controlo de concorrência é o recurso a estampilhas para impor a ordem total pela qual as transacções vão ser executadas [SKS05]. Neste mecanismo todas as transacções quando iniciam recebem um estampilha TS, uma marca que define uma relação de ordem com as outras transacções em execução. Uma transacção mais antiga terá um valor de estampilha inferior.

Quando se determina existir um conflito entre uma operação de T_1 e outra de T_2, a sua ordem de execução vai ser decidida com base na comparação entre $TS(T_1)$ e $TS(T_2)$. Normalmente é a transacção que detecta que a ordem das estampilhas não está a ser respeitada que é abortada e reiniciada.

Multiversão Este método de controlo de concorrência [SKS05] baseia-se no conceito de versão tentativa de um objecto. Uma é uma versão de um objecto privada a uma transacção, na qual são reflectidas todas as operações de escrita, e uma versão global que corresponde à última versão efectiva, que é comum a todas as transacções do sistema e onde são realizadas as operações de leitura. Deste modo cada operação de leitura nunca vai ser bloqueada e não vai falhar, o que é uma vantagem, no caso de a carga de trabalho sobre o sistema ser na sua maioria de leituras.

Níveis de isolamento

O intuito dos mecanismos de controlo de concorrência é garantir a correcta execução das transacções na presença de concorrência (isolamento). Contudo, os ordenamentos correctos gerados por estes métodos podem limitar o desempenho do sistema. Para permitir um maior nível de concorrência foram definidos, no domínio das base de dados, vários níveis de isolamento de acordo com o tipo de conflitos que permitem. O nível de isolamento de uma transacção mede a tolerância desta em relação às alterações nos dados, por ela lidos e que foram modificados por outras transacções.

São definidos quatro níveis de isolamento tendo em conta três situações que podem ocorrer quando são executadas simultaneamente duas transacções, estes são: *dirty read*, *unrepeatable read* que já foram descritos na Secção 2.1.1 e *phantom read*, que se trata de uma situação muito similar ao *unrepeatable read*, e exemplifica-se da seguinte forma: no

decorrer de uma transacção é executa por uma segunda vez uma consulta, restrita por uma determinado condição, que devolve um conjunto de resultados e constata-se que esse conjunto é diferente do devolvido da primeira vez, isto deve-se ao facto de outra transacção ter alterados dados entre as duas consultas. Na Tabela 2.2 são apresentados os níveis isolamento, com a noção das situações que permitem ou impedem.

	Problemas		
	Dirty read	*Unrepeatable read*	*Phantom read*
Read uncommited	Permite	Permite	Permite
Read commited	Impede	Permite	Permite
Repeatable read	Impede	Impede	Permite
Serializable	Impede	Impede	Impede

Tabela 2.2: Níveis de isolamento e problemas resolvidos

2.1.2 Recuperação

Para permitir que uma transacção seja atómica é necessário implementar mecanismos de recuperação para que as transacções possam ser revertidas no caso de abortarem, ou seja, para que possa ser reposto o estado anterior à sua execução.

O *Logging* é uma técnica que consiste na manutenção de um diário (*Log*) utilizado para registar as alterações realizadas por uma transacção. Este *log* é consultado para permitir ao sistema voltar a um estado anterior a execução de uma transacção. Existem dois tipos de *log* [Gra81]: o *undo log* e o *redo log*. Em ambos os casos é mantido um registo para cada objecto com os valores escritos ou os valores que a escrita substituiu. A diferença entre os dois tipos baseia-se na forma com é gerido este registo.

Undo log Nesta técnica, por cada operação de escrita realizada por uma transacção, é registado no *log* o valor actual do objecto a actualizar, sendo de seguida actualizado o valor do objecto. Esta estratégia é muitas vezes também denominada de *direct-update*. Assim, quando é necessário reverter uma transacção, é percorrido o *log* e o objecto é actualizado para o valor anterior ao inicio da transacção. Quando uma transacção realiza *commit*, o *log* é simplesmente descartado.

Redo Log Na técnica de *redo*, cada operação de escrita realizada pela transacção implica um registo no *log* do novo valor do objecto, mantendo-se o objecto inalterado. Esta estratégia é muitas vezes denominada de *deferred-update*. Quando uma transacção pretende ler o valor de um objecto é necessário consultar o *log* a fim de ler o valor

mais recente e, no caso de não existir nenhuma entrada para esse objecto, é então consultado o objecto directamente. Quando a transacção aborta este *log* simplesmente é descartado, enquanto que no caso da transacção realizar *commit* os valores presentes no *log* têm de ser aplicados aos respectivos objectos.

2.1.3 Bases de dados

A maioria dos sistema de base de dados actualmente suportam o modelo transaccional, pois este define muitas das propriedades necessárias a este tipo de sistemas.

O SQL, que é uma linguagem especifica para o domínio das base de dados, definindo comandos para lidar com transacções:

`BEGIN`: Marca o inicio de uma transacção, a partir deste ponto todas as alterações à base de dados poderão permanecer invisíveis aos outros utilizadores, dependendo dos seus níveis de isolamento.

`COMMIT`: Marca o fim de uma transacção realizada com sucesso, tendo todos os efeitos desta transacção sido aplicados a base de dados.

`ROLLBACK`: Faz com que a transacção seja revertida, desfazendo todas as modificações realizadas desde o inicio da transacção vão ser desfeitas.

Tanto o `COMMIT` como o `ROLLBACK` terminam a transacção, pelo que após a sua aplicação é necessário invocar o `BEGIN` para iniciar uma nova transacção.

Alguns exemplos deste tipo de sistema de base de dados são: o PostgreSQL [Pos08], o MySQL [Sun08], o Oracle Database [Ora08] e o SQL Server [Mic08]. Todos este sistema implementam, por vezes de uma forma limitada, os conceitos descritos anteriormente para os sistema transaccionais.

2.1.4 Memória transaccional

A evolução da capacidade de processamento dos sistemas até há pouco tempo dependia, na maioria dos casos, apenas do simples incremento da velocidade de relógio do CPU. Mas este incremento contínuo da velocidade do CPU tornou-se difícil de prosseguir devido às necessidades de energia e de refrigeração que as frequências elevadas exigiam [Sut05].

Para dar resposta a este problema surgiram os processadores com múltiplos núcleos (*multicore*), pois estes são compostos por n núcleos (normalmente idênticos mas não obrigatoriamente) e utilizando memória partilhada ou outros mecanismos para troca de informação. Assim, teoricamente, incrementando o número de núcleos nos processadores permitindo continuar a aumentar a capacidade de processamento, mas

com um menor consumo energético. Esta abordagem está actualmente a ser adoptada por parte de todos os construtores de hardware, estando a tornar-se comum nos computadores pessoais. A Intel está já a estudar as implicações destas arquitecturas com centenas de núcleos, dispondo mesmo um prototipo com 80 núcleos [Res08].

Esta nova arquitectura de hardware veio massificar a computação paralela, que até há pouco tempo estava acessível essencialmente a grupos de investigação e a grandes empresas, vindo assim reactivar e renovar toda a problemática associada à programação paralela.

Na programação paralela é necessário evitar o processamento de dados inconsistentes, que são consequência do acesso não controlado a regiões de dados partilhados. A consistência é normalmente garantida recorrendo à definição de regiões de exclusão mútua, onde somente é permitido o acesso aos dados por uma *thread* e todas as demais que queiram aceder à mesma região têm de aguardar. Isto pode ser conseguido através da utilização de *locks* mas, contudo, este método apresenta alguns problemas [MMW07], como a possibilidade de *deadlocks*, a inversão de prioridade e o bloqueio em caso de falha.

Dado os inconvenientes dos *locks* e a necessidade de tornar a programação paralela acessível às massas, motivado pela generalização das novas arquitecturas *multicore*, surge um crescente interesse pelo modelo de memória transaccional, inicialmente introduzida por Lomet [Lom77] em 1977 e mais recentemente especificada por Herlihy [HM93] em 1993. No modelo de memória transaccional, o código que acede a dados partilhados em memória é delimitado por marcas de início e fim de transacção, dando uma semântica transaccional ao código executado no contexto de uma transacção (ver Figura 2.4, adaptada de [KCH+09]), ou seja, existe a garantia de respeito pelas propriedades ACI. A propriedade da durabilidade em memória não tem a mesma relevância do que nas bases de dados, já que esta é um dispositivo de natureza volátil.

Para possibilitar a utilização deste modelo, aparece a noção de bloco atómico definido em linguagens de programação. Um bloco atómico delimita uma sequência de instruções que devem de ser executadas numa transacção, assumindo-se que esse bloco beneficia das propriedades de atomicidade e isolamento.

O construtor atómico apresenta como vantagem não nomear os recursos/dados partilhados nem os mecanismos de sincronização utilizados ou a utilizar. As transacções especificam os objectivos da execução e deixam a cargo do sistema de memória transaccional a sua implementação. Assim, quando uma transacção faz *commit*, o seu resultado passa a fazer parte do estado visível do programa, enquanto que se fizer *abort* o estado deste fica inalterado. Não está definido o caso em que a transacção não termina. Um exemplo da sua utilização pode ser visto na Figura 2.5 (retirada

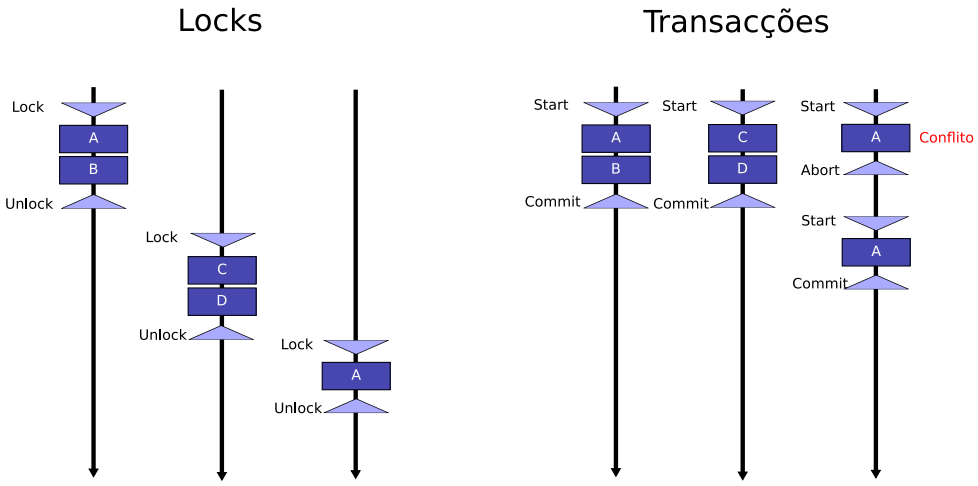


Figura 2.4: Comparação de utilização entre *locks* e memória transaccional

de [KCH+09]), que apresenta também a mesma sequência de instruções utilizando *locks*.

```
atomic {
    A = A - 10;
    B = B + 10;
    ...
    if ( error )
        abort_transaction;
}
```

```
lock(L1); lock(L2);
    A = A - 10;
    B = B + 10;
    ...
    if ( error )
        recovery_code();
unlock(L1); unlock(L2);
```

Figura 2.5: Definição de um bloco atómico em memória transaccional e o seu equivalente utilizado *locks*

Este modelo oferece ao programador uma forma simples de desenvolver código paralelo. A memória transaccional pretende ser tão simples de utilizar como os *locks* de grão grosso e apresentar um desempenho próximo do alcançado com *locks* de grão fino e sem certos problemas dos *locks* (*deadlocks*). Um exemplo onde este modelo alcança as propriedades descritas é apresentado em [SATH+07].

A implementação de memória transaccional apresenta algumas dificuldades. Uma das principais dificuldades é lidar com as operações que ultrapassam o contexto transaccional, como sejam acessos a bases de dados, comunicações entre processos e mesmo acesso a ficheiros.

As operações executadas no contexto da memoria transacional podem ser de dois tipos [DLC08]:

Operações não transacionais : estas são as operações para as quais não existe forma de as anular ou reverter. Uma solução será adiar a sua aplicação até a confirma-

ção final. Mas nem sempre o método de realizar o adiamento da execução de operações é possível devido à lógica da aplicação, se por exemplo esta for interactiva. Um dos exemplos onde este problema se coloca é nas operações de I/O, e em particular nas operações sobre ficheiros (que irá ser abordado no Capitulo 3). Em [Dia08] apresenta-se uma abordagem para suportar I/O para as bases de dados, através da unificação dos modelos transaccionais de memória e de base de dados.

Operações transacionais : estas operações podem ser desfeitas e como tal podem ser realizadas no contexto das transacções em memória. Estas operações podem ainda ser subdivididas em três tipos: puramente transacional, reversível e compensaveis. As operações puramente transacionais podem ser automáticamente revertidas quando necessário, como é exemplo as operações de leitura e escrita de memória. Quando as reversívies, estas podem ser revertidas com o acrescimo de código, sendo exemplos deste tipo as operações de gestão expliciata de memória. Por fim, as operações compensáveis, são as operações que podem ser compensadas com outra operção realizada noutra transacção, como por exemplo ser truncado um ficheiro que foi anteriormente expandido por outra transação.

2.2 Sistemas de ficheiros

O sistema operativo oferece uma visão uniforme da informação guardada em memória secundária, permitindo abstrair das propriedades físicas dos dispositivos ao definir uma unidade lógica de armazenamento: o ficheiro. Um ficheiro é uma conjunto de informação inter-relacionada (normalmente guardado em memória não volátil) à qual está associado um nome.

O conjunto de atributos associado a cada ficheiro permite garantir uma adequada gestão e controlo destes. O nome, uma cadeia de caracteres, é o principal atributo de um ficheiro, pois permite identificar o conteúdo do ficheiro, tornando-o independente do seu criador. Os demais atributos associados variam de sistema para sistema mas, no essencial, podem ser resumidos em: tipo; localização; protecção; data/hora e criador.

O ficheiro é uma estrutura de dados abstracta e, como tal, existe um conjunto de operações que se podem realizar sobre estes, sendo os principais: criar, escrever, ler, reposicionar, apagar e truncar um ficheiro. Um exemplo deste conjunto de operações disponibilizadas é definido pela norma POSIX 1003.1 [POS92].

Para evitar a constante pesquisa de informação nos meta-dados que as operações sobre ficheiros implicam, normalmente os sistemas impõem a necessidade de encaixar as operações sobre um ficheiro entre uma operação de abertura (*open*) e uma outra

de fecho (*close*). A partir do momento em que o ficheiro é aberto a única informação necessária para o manipular é o respectivo descritor de ficheiro aberto.

Alguns sistema permitem realizar *lock* sobre um ficheiro ou parte deste. Isto permite a um processo obter acesso a um ficheiro e impedir o acesso de outros, evitando assim os acesso concorrentes e garantindo a coerência dos conteúdos. Dependendo de cada sistema, os *locks* disponibilizados podem ser unicamente exclusivos ou ser também partilhados.

Os mecanismos de *lock* sobre ficheiros podem ser obrigatórios ou apenas recomendados. Se o *lock* é obrigatório, quando é aberto um ficheiro o sistema garante que mais nenhum processo está ou vai aceder a esse ficheiro enquanto o processo o mantiver aberto. Na caso de os *locks* serem recomendados, o sistema não oferece garantias quanto a restrições de acesso, sendo da responsabilidade do programador realizar a implementação da exclusão mútua. O Windows utiliza *locks* obrigatórios enquanto se procede a abertura de um ficheiro para escrita, enquanto que o UNIX emprega *locks* recomendados, qualquer se seja o modo de abertura.

Para fazer a correspondência entre o nome e a meta-informação de um ficheiro existe a noção de directoria. A directoria é uma lista de entradas que associa um nome a um ficheiro. A directoria oferece um conjunto de serviços de serviços para manipulação das suas entradas, como seja adicionar entradas, mover, listar e renomear.

Para permitir um agrupamento lógico de ficheiros, uma estruturação de domínios de nomes dos ficheiros e uma melhor eficiência na pesquisa de um ficheiro, as directorias estão frequentemente organizadas em forma de árvore, em que uma directoria contém entradas de ficheiros e de sub-directorias.

2.3 Consistência e recuperação

Existem estruturas, tanto em disco como em memória, que são a meta-informação de gestão dos dados guardados em ficheiros e em directorias. Uma simples operação sobre um ficheiro desencadeia várias acções sobre a meta-informação no sistema de ficheiros. No caso de ocorrer uma falha (não prevista), enquanto decorre a execução de uma operação, esta meta-informação pode ficar inconsistente ou ilegível, o que pode levar à perda de dados.

No caso de uma falha não prevista, como seja a falta de energia, as duas imposições anteriores não são garantidas trivialmente, o que leva à possível existência de inconsistências no sistema de ficheiros ou mesmo à perda de dados.

2.3.1 Verificação

Muitas das vezes a forma de lidar com estas inconsistências é *post factum*. Tal forma consiste na execução de um programa que vai verificar a consistência do sistema de ficheiros, a fim de encontrar e corrigir as informações erradas no sistema. Um exemplo desses problemas é o caso em que os meta-dados do ficheiros não condizem com a informação guardada na estrutura de directorias.

O verificador de consistência compara os dados presentes em toda a estrutura de directorias com os blocos em disco e tenta reparar qualquer inconsistência que detecte. Esta verificação consiste em percorrer toda e estrutura de directorias e encontrar os casos em que um *inode* não é referenciado ou que o número de *links* assinalado é diferentes do inferido.

Contudo esta solução não impede muitas vezes a perda de dados, pois algumas vezes não é possível realizar a recuperação da meta-informação de forma a ser possível aceder aos dados.

2.3.2 *Soft Updates*

A técnica de *Soft Updates* [GMSP00] consiste em garantir que os blocos são escritos para disco numa ordem pré-definida e sem a utilização de operações síncronas de I/O. De forma geral, um sistema com *Soft Updates* tem de manter a informação de dependência entre os vários blocos de dados que vão ser escritos para disco. Por exemplo, quando um ficheiro é criado, o sistema tem de garantir que o novo *inode* está em disco antes da entrada na directoria que o refere estar actualizada. Para tal o sistema tem de ter informação de que o bloco de dados referente a directoria está dependente do novo *inode*. Assim o bloco que corresponde à directoria nunca é escrito antes do *inode* ter sido escrito em disco.

Esta técnica implica que cada escrita seja adiada para uma cache, a fim de respeitar as possíveis dependências entres os blocos. Isto pode ter implicações benéficas no desempenho do sistema, já que, segundo os autores, pode existir uma redução de 40 a 70 por cento na quantidade de escritas. Sendo o beneficio mais evidente no caso da criação de ficheiros temporários, pois estes podem nem chegar a ser escritos para disco.

Assim, em caso de falha do sistema, as únicas inconsistências que podem surgir no disco são blocos e *inode* marcados como ocupados quando, na realidade, não o estão. Ou seja, as inconsistências podem levar à perda de dados, mas é sempre possível recuperar os meta-dados para um estado consistente.

2.3.3 *Journaling*

Outra abordagem para lidar com a inconsistência num sistema de ficheiros consiste em impor a noção de transacção às operações sobre ficheiros, directorias e demais meta-dados, evitando assim a existência de inconsistências. É possível alcançar tal objectivo com a utilização de um *log* (diário) em memória estável onde são registadas todas as operações antes destas serem aplicadas no sistema de ficheiros.

Quando se realiza uma alteração no sistema de ficheiros esta informação é registada no *log* como o conjunto de todas as suas sub-operações. A alteração assume-se como confirmada (*committed*) quando escrita no *log* e permite ao cliente prosseguir a partir desse momento. Depois de registadas as alterações no *log* estas são aplicadas de forma assíncrona e, à medida que vão sendo aplicadas, é actualizado um apontador que refere quais as acções aplicadas e quais as que ainda não o foram. Quando as sub-operações de uma alteração são todas aplicadas, esta entrada é removida do *log*. As alterações no *log* são processadas segundo uma ordem FIFO (*First In, First Out*).

Assim em caso de falha, no arranque do sistema o *log* é percorrido e todas as alterações que tenham sido confirmadas mas não executadas vão ser aplicadas ao sistema de ficheiros, enquanto que as não confirmadas vão ser ignoradas.

Existem dois grandes tipos de *journaling*, que diferem na quantidade de informação preservada no *log*. Um dos tipos, meta-data e data *journaling*, guarda no ficheiro de *log* tanto a meta-informação necessária para manter a consistência do sistema de ficheiros, como os dados escritos para cada bloco, permitindo assim uma recuperação total em caso de falha. No entanto esta aproximação implica uma perda considerável de desempenho, já que todas as escritas são duplicadas e o espaço requerido para o *log* necessita de ser muito maior. O outro tipo, meta-data *journaling*, apenas guarda a meta-informação no ficheiro de *log* somente, o que melhora o desempenho e minimiza a utilização de espaço. Esta aproximação oferece menos garantias que a anterior, já que não garante a capacidade de recuperar o conteúdo de um bloco escrito.

A utilização de *journaling* permite uma recuperação muito mais rápida do que a verificação do sistema de ficheiros. Contudo, tem como contrapartida uma perda de desempenho devido à necessidade de realizar mais operações de IO para gestão do ficheiro de *log* e, dependendo do tipo utilizado, esse impacto pode ser maior ou menor. Na actualidade, este é um mecanismo muito utilizado nos sistemas de ficheiros actuais (e.g., Ext3 [Twe98], NTFS [SC98], JFS [Ste09]).

2.4 Suporte à implementação

Para implementar um sistema de ficheiros é necessário criar uma ligação com o núcleo do sistema operativo, pois é o sistema operativo que é responsável pela gestão de ficheiros. Para realizar tal tarefa, o programador do sistema de ficheiros tem de definir um conjuntos de funções que dêem resposta às necessidade do sistema operativo. Existem essencialmente duas formas de o fazer: através do acréscimo de funcionalidades ao núcleo do sistema ou através de um sub-sistema já definido, que faz a redirecção das chamadas ao sistema de ficheiros para uma implementação realizada em espaço de utilizador.

2.4.1 *Virtual File System*

Os sistemas operativos modernos necessitam de suportar diversos tipos de sistemas de ficheiros. Um sistema de ficheiros virtual (abreviado como VFS, do inglês *Virtual File System* [KM86]), é uma camada de abstracção que está acima da implementação específica de um sistema de ficheiros.

Esta camada de abstracção oferece às aplicações transparência no acesso a diferentes tipos de sistemas de ficheiros, quer eles sejam locais ou remotos. Assim, as aplicações lidam com os diferentes sistemas de ficheiros de forma uniforme, sem necessidade de conhecer os seus detalhes específicos. Um esquema da arquitectura do VFS no sistema Linux está na representado na Figura. 2.6 (extraída de [IBM09]). Este componente é responsável por dois grandes serviços [SGG04] :

- Fazer uma clara separação entre as operações sobre um sistema genérico e a sua implementação. Várias implementações para a interface VFS podem coexistir na mesma máquina, permitindo assim o acesso transparente a diversos tipos de sistemas de ficheiros.
- Fornecer um mecanismo para representar um ficheiro de forma genérica. Para isso, usa uma estrutura de dados que representa o ficheiro, designada por *vnode*, que é completamente independente da implementação.

O VFS mantém, para um conjunto pré-definido de funções, o registo da sua implementação de acordo com cada tipo de sistema de ficheiros.

vnode: que representa um ficheiro em abstracto;

file: que representa um ficheiro aberto;

superblock: que representa um sistema de ficheiros;

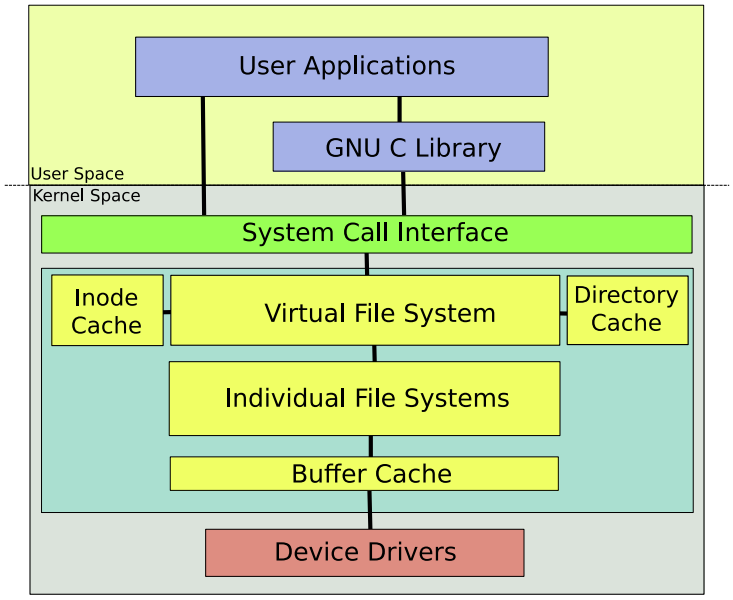


Figura 2.6: Arquitectura do VFS no sistema Linux

dentry: que presenta uma entrada numa directoria.

Para realizar o desenvolvimento de um novo sistema de ficheiros recorrendo ao VFS é necessário trabalhar ao nível do *kernel* do sistema. Trabalhar a este nível tem, por um lado, a vantagem de permitir um melhor desempenho. Por outro lado é necessário lidar com estruturas e níveis de abstracção de baixo nível, logo mais complexos de gerir. Acrescendo ainda as possíveis implicações de estar a trabalhar junto com o *kernel* do sistema. Destas implicações podemos destacar o difícil *debug*, com consequências graves caso haja erros e problemas de segurança.

2.4.2 *Filesystem in Userspace* (FUSE)

O sistema *Filesystem in Userspace* (FUSE) [HSP+08] começou por ser desenvolvido no seio de um outro projecto intitulado *A Virtual File System* (AVFS) [HS08]. Este projecto tinha como objectivo permitir aos programas terem acesso ao conteúdo de arquivos, ficheiros comprimidos ou ficheiros remotos, sem ser necessário recompilar os programas ou o núcleo do sistema. Mas o FUSE rapidamente se tornou um projecto independente, pois permite uma utilização mais genérica, possibilitando desenvolver um sistema de ficheiros a nível utilizador, sem a necessidade de alterar o núcleo do sistema. Isto permite uma visão em que "tudo pode ser um sistema de ficheiros". O FUSE está actualmente disponível nos sistemas operativos Linux, FreeBSD, NetBSD, OpenSolaris e Mac OS X. O FUSE é composto por três componentes principais, que estão ilustrados na Figura 2.7 (extraida de [Wik09]).

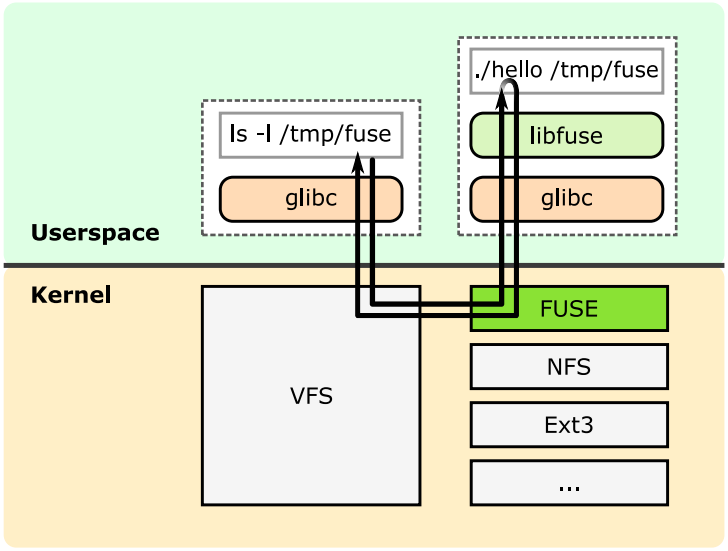


Figura 2.7: Exemplo da estrutura do sistema FUSE

O primeiro componente é um módulo no *kernel* do sistema que se apresenta perante o VFS como um novo sistema de ficheiros. Adicionalmente disponibiliza um *device* especial para comunicação com processos nível utilizador. O segundo componente é um biblioteca (*libfuse*) que comunica com o *device* numa linguagem própria. Por último temos a definição do sistema de ficheiros, que com recurso a biblioteca consegue registar um ponto de montagem e definir um conjunto de *handlers*.

Uma operação sobre um ficheiro que está localizado num sistema de ficheiros desenvolvido com recurso ao FUSE vai desencadear os seguintes passos: o VFS recebe o pedido, constata que se trata de um pedido para um ponto de montagem registado como sendo FUSE e invoca as funções registadas pelo módulo; o módulo, ao receber a invocação, verifica o registo para o ponto de montagem e envia o pedido respeitante ao serviço invocado para a respectiva implementação; a implementação processa o pedido e reenvia o resultado para o módulo; o módulo reenvia essa informação para o VFS.

No nível utilizador o FUSE disponibiliza uma biblioteca que comunica com o módulo do núcleo, aceitando os pedidos vindos do *device* e fazendo a sua tradução para chamadas de funções similares às do interface do VFS. Essas funções possuem nomes como `open()`, `read()`, `write()`, `rename()`, `symlink()`, etc. Embora obviamente exista uma perda de desempenho, pela necessidade de redireccionamento da chamadas por parte do módulo para a implementação concreta, existe uma tentativa de minorar este efeito através da implementação de uma comunicação rápida e a interface de chamadas ser muito similar à oferecida pelo VFS.

Por fim existe um componente que implementa o sistema de ficheiros da forma

desejada. Este componente é responsável por preencher uma estrutura de dados, a `fuse_operations` (ver Figura 2.8) que faz parte da biblioteca do FUSE, com funções que implementem as operações invocadas de acordo com a lógica do sistema de ficheiros. Este é o componente pelo qual o programador do sistema de ficheiros é responsável.

```
struct fuse_operations {
    int (*getattr) (const char *, struct stat *);
    int (*readlink) (const char *, char *, size_t);
    int (*getdir) (const char *, fuse_dirh_t, fuse_dirfil_t);
    int (*mknod) (const char *, mode_t, dev_t);
    int (*mkdir) (const char *, mode_t);
    int (*unlink) (const char *);
    int (*rmdir) (const char *);
    int (*symlink) (const char *, const char *);
    int (*rename) (const char *, const char *);
    int (*link) (const char *, const char *);
    int (*chmod) (const char *, mode_t);
    int (*chown) (const char *, uid_t, gid_t);
    int (*truncate) (const char *, off_t);
    int (*utime) (const char *, struct utimbuf *);
    int (*open) (const char *, struct fuse_file_info *);
    int (*read) (const char *, char *, size_t, off_t, struct fuse_file_info *);
    int (*write) (const char *, const char *, size_t, off_t,struct fuse_file_info *);
    int (*statfs) (const char *, struct statfs *);
    int (*flush) (const char *, struct fuse_file_info *);
    int (*release) (const char *, struct fuse_file_info *);
    int (*fsync) (const char *, int, struct fuse_file_info *);
    int (*setxattr) (const char *, const char *, const char *, size_t, int);
    int (*getxattr) (const char *, const char *, char *, size_t);
    int (*listxattr) (const char *, char *, size_t);
    int (*removexattr) (const char *, const char *);
};
```

Figura 2.8: Alguns das operações definidas na estrutura `fuse_operations`

Actualmente existe uma grande quantidade de sistemas de ficheiros desenvolvidos com recurso ao *framework* FUSE, desde NTFS [NTF09] a ZFS [Sun09, Ric09].

Ao desenvolver um sistema de ficheiros recorrendo ao FUSE vamos ter vantagens e desvantagens. Por um lado, como estamos a introduzir *overhead* no processamento das chamadas aos ficheiros, com a consequente penalização no desempenho. Por outro lado, vamos facilitar o desenvolvimento, tanto no processo de criação como do se *debug*, visto que estamos a trabalhar a um nível de abstracção mais elevado. Para além de que se assegura uma maior segurança e estabilidade ao sistema operativo, pois um erro que exista na implementação do sistema de ficheiros vai implicar uma falha local e não geral.

O FUSE oferece uma interface simples e que permite desenvolver um sistema de ficheiros para diversos sistemas operativos sem nos obrigar a lidar com a variação das especificações do núcleo do sistema e sem ter de recompilar o mesmo, já que a nossa aplicação somente vai interagir com o sub-sistema FUSE, que actua como um *middleware* para a nossa aplicação.

2.5 *Benchmarking*

A eficiência é, em certos domínios, um requisito muito importante, para o que são utilizados *benchmarks*. Um *benchmark* consiste num programa ou conjunto de programas que permite avaliar o desempenho relativo de um sistema.

Idealmente seria o utilizador final a executar diversos *benchmarks* com uma carga de trabalho apropriada aos seus objectivos e a obter os resultados. No entanto tal tarefa é difícil e implica um esforço acrescido para o utilizador, para além de que exige metodologias de teste e capacidade de análise de resultados que muitos utilizadores não dispõem. Assim é normal serem os criadores do sistema ou utilizadores especializados a realizarem os testes de *benchmarking*. Os *benchmarks* podem então ser categorizados da seguinte forma [TZJW08]:

***Macrobenchmark*:** consiste num teste que envolve múltiplas operações, com base numa carga de trabalho representativa de um ambiente real. Este tipo de *benchmarks* permite obter uma boa avaliação do desempenho global do sistema, muito embora a carga utilizada possa diferir da pretendida na utilização final.

***Trace based benchmark*:** consiste em aplicar as operações que foram capturadas anteriormente num cenário de utilização real.

***Microbenchmark*:** consiste num teste sobre um número restrito de operações, com a finalidade de isolar a sua especificidade e avaliar o seu comportamento.

No processo de criação de um *benchmark* é necessário tem em atenção um conjunto de vertentes, como seja, qual o hardware, qual a carga do sistema, quais os pontos que podem causar uma perda ou ganho de desempenho. A partir deste ponto pode-se pensar em prováveis resultados que mais tarde serão comprovados na prática.

As características do sistema onde o *benchmark* é executado podem afectar significativamente os resultados deste, pois factores como o estado da cache e o nível de carga do sistema podem ser um impacte não negligenciavel nos resultados obtidos na execução dos testes.

Os testes têm de ser executados várias vezes em condições idênticas a fim de obter um nível de precisão satisfatório. A duração da execução dos testes deve de ser significativa, a fim de levar o sistema a alcançar um estado estável por na maior parte do tempo. Por último os testes devem ser automatizados com o intuito de evitar os erros associados a execução manual.

Os resultados devem de ser apresentados, descrevendo a configuração e ambiente de execução utilizados, recorrendo a gráficos e mostrando qual a noção de erro que foi assumido. Para uma boa validação devem de ser fornecidas as informações suficiente sobre hardware e software para que seja possível compreender a base do teste, e a sua repetição, levando a uma possível comparação de resultados.

2.6 Sistemas de ficheiros com suporte transaccional

Actualmente já existem sistema de ficheiros que, para garantir a consistência dos meta-dados e uma forma rápida de recuperação depois da ocorrência de falhas, implementam o conceito de transacções ao nível das operações internas do próprio sistema de ficheiros.

Contudo, ao nível do utilizador do sistema de ficheiros, podem também ser disponibilizadas as propriedades transaccionais, sendo então designadas como transacções externas. Neste caso considera-se que o sistema de ficheiros é um sistema de ficheiros transaccional. Neste tipo de sistemas de ficheiros a semântica das propriedades ACID pode ser exemplificada da seguinte forma:

Atomicidade: um grupo de operações sobre ficheiros, é agregada como um bloco que tem de ser aplicado como um todo.

Consistência: esta propriedade já é garantida pelos sistema de transacções internas, pois para a aplicação é definido que o sistema de ficheiros está consistente.

Isolamento: a presença de concorrência no acesso transaccional a ficheiros requer alguma garantia de isolamento das diversas transacções. Contudo, o sistema de ficheiros não necessita de disponibilizar, necessariamente, um nível de isolamento máximo, podendo este ser relaxado. Por exemplo, o sistema pode oferecer um nível de *read uncommitted* e permitir que outras transacções vejam as suas operações sobre o sistema de ficheiros mesmo antes destas serem confirmadas.

Durabilidade: esta propriedade implica que todas as operações envolvidas numa transacção sejam reflectidas em disco. Isto pode ser conseguido com o forçar de *flush* dos *buffers* de I/O aquando da confirmação da transacção.

De seguida são apresentados alguns sistema de ficheiros transaccionais e algumas das suas principais características. Alguns destes sistemas de ficheiros resultam de projectos de investigação onde foi utilizado como suporte interno um sistema de base de dados para garantir as propriedades ACID.

2.6.1 *Inversion File System*

A fim de oferecer suporte a um grupo de investigação cientifica, que necessitava de gerir grande quantidades de informação com uma alguma especificidade e com algumas garantias contra possíveis situações de falha, foi desenvolvido o sistema *Inversion File System* (IFS) [Ols93]. Este sistema disponibiliza, através de uma biblioteca, um número limitado de funções de acesso a ficheiros muito similares com as tradicionais nos sistemas POSIX.

O IFS oferece a noção de transacção para o utilizador e permite-lhe consultar versões anteriores dos dados. Para além disto, este sistema permite a definição de funções especificas associadas a determinados tipos de ficheiros, como a realização de perguntas sobre meta-dados ou dados.

O sistema está implementado com recurso a uma base de dados *PostgreSQL*, com o qual dialoga utilizando uma linguagem de perguntas. No *PostgreSQL* são guardados tanto os meta-dados bem como os dados em forma de tabelas, num esquema relacional. O IFS utiliza esta base de dados para oferecer todas as funcionalidades transaccionais e de controlo e gestão de múltiplas versões.

Este sistema fornece uma visão interessante sobre as capacidades possíveis de alcançar através da utilização de transacções nos sistemas de ficheiros, contudo apresenta um sistema de ficheiros separado de todos os demais e impõe a utilização de funções próprias para acesso a ficheiros.

2.6.2 *Database File System*

Com o intuito de avaliar a possível implementação eficiente de um sistemas de ficheiros sobre uma base de dados, foi desenvolvido o *Database File System* (DBFS) [MTV02]. O DBFS utiliza um sistema de base de dados embutido, o Berkeley DB [OBS99], no qual se baseia a fim de fornecer as propriedades de ACID ao nível do sistema de ficheiros, contudo estas propriedades não são acessíveis ao nível do utilizador.

O BDFS é implementado como uma biblioteca ou como um servidor de NFS de nível utilizador. Como a versão em biblioteca não utiliza uma interface POSIX, pelo que as aplicações que o queiram utilizar terão de ser modificadas. A versão que utiliza NFS evita esta dificuldade mas acrescenta um nível adicional de *overhead* no processamento das chamadas ao sistema.

O Berkeley DB é utilizado para guardar todos os dados e meta-dados dos ficheiros, distribuídos por três tabelas: *blocks* onde são guardados os conteúdos dos ficheiros identificados por blocos e *inode*; *metadata* que mantém todas a meta-informação para cada *inode*; e *dirtree* onde é guardada informação da árvore de directorias.

Segundo os autores o DBFS as operações de leitura podem chegar a ser na ordem de 5 vezes mais lentas e as de escrita 40 vezes mais lentas, quando comparadas em teste com o *Fast File System* (FFS) [MJLF84].

2.6.3 *Amino File System*

Um pouco no seguimento dos projectos anteriores foi desenvolvido o sistema *Amino File System* (AFS)) [WSSZ07], um sistema de ficheiros que oferece as propriedades ACID para o nível da aplicação. Este software foi desenvolvido novamente utilizando um sistema de base de dados embutido, o Berkeley Database [OBS99] que fornece as infra-estruturas fundamentais de uma base de dados.

O Amino FS foi implementado como um monitor de nível aplicacional, utilizando o *ptrace* como o componete de intercepção de chamadas das aplicações ao sistema de ficheiros, transferindo a sua realização para o sistema de ficheiros AFS. Todos os dados como meta-dados são guardados na base de dados embutida.

O Amino exporta, através da sua API, a noção de transacção para o nível utilizador, atribuindo a cada transacção um identificador único, o que permite que outro processo, diferente daquele que o criou, obtenha controlo sobre essa transacção.

O sistema foi avaliado utilizando várias tarefas e *microbenchmarks* que foram comparados com execuções sobre um sistema Ext3. Estes testes demonstraram que é possível adicionar as propriedades ACID, sem que para isso o desempenho seja muito penalizado, obtendo no pior caso 16% de *overhead* quando comparado com o Ext3.

2.6.4 *PerDiS File System*

O PerDiS é um infraestrutura de desenvolvimento de aplicações que acedam a dados persistentes de forma transaccional. Para a tarefa de armazenamento persistente tolerante a falhas para ambientes distribuídos de larga escala foi desenvolvido o PerDiS FS [GFG98].

Os dados no PerDiS são representados por grafos de objectos que são guardados segundo um modelo de alcance. Esses grafos são agrupados em *clusters* e guardados em ficheiros que são vistos como sequências de bytes. Os ficheiros são referenciados por URLs, que permitem uma localização rápida e a utilização de uma abordagem de *forwarding pointers* para permitir a mobilidade dos objectos.

O PerDiS FS utilizada um protocolo optimista com um sistema de notificações para gestão das transacções o que permite a partilha de dados e transacções de longa duração. Para tratamento das transacções distribuídas é utilizado o protocolo *two-phase-commit* não bloqueante [SKS05]. Quando um cliente faz *commit*, os dados são escritos

para disco, no qual se delega a tarefa da manutenção estável das actualizações.

O PerDiS FS utiliza um sistema de versões com ramos o que permite que as transacções optimistas não sejam abortadas. Podem ocorrer problemas de reconciliação que, no caso de não poderem resolvidos de uma forma automática, terão de ser resolvidos pelo utilizador, de forma manual.

No seguimento do projecto PerDiS foi, desenvolvido o Sinfonia [AMS+07]. Este sistema apresenta um novo conceito, o das mini-transacções, que permite um acesso mais eficiente aos dados. Contudo este tipo de sistemas são específicos para o desenvolvimento de aplicações em ambientes distribuídos.

2.6.5 *Transactional Flash File System*

A memória *flash* é uma memória de computador do tipo EEPROM [1], isto é, uma memória que pode ser programada e apagada várias vezes, electronicamente. Esta memória é do tipo não volátil o que significa que não precisa de energia para manter as informações armazenadas, contudo existe um limite do número de programações (escritas) neste dispositivo na ordem de 100.000 a 1 milhão de vezes.

Os sistemas de ficheiros desenvolvidos para os discos magnéticos não se adequam a este tipo de memória, pois esta não beneficia da localidade nas operações de leitura e escrita, mas já a localidade no caso de apagar dados é relevante. Por outro lado, os sistemas de ficheiros actualmente desenvolvidos para pequenos sistemas embutidos necessitam de grandes quantidades de RAM[2], o que não é muito apropriado para micro controladores.

O Transactional Flash File System(TFFS) [GT05] foi desenvolvido visando os tipos de memória *flash* que são utilizadas nos micro controladores de sistemas num só *chip* e *chips flash* de baixo custo.

O TFFS tenta oferecer suporte às aplicações embutidas de alta disponibilidade, fazendo uma eficiente utilização de RAM e do armazenamento *flash*. Assim e para facilitar a implementação das aplicações o sistema suporta a noção de transacção ao nível das APIs que disponibiliza.

O TFFS utiliza uma unidade de armazenamento para um *log*, para poder garantir a atomicidade das operações. As restantes são utilizadas pela memória do sistema para guardar blocos de tamanho variável. Este sistema utiliza uma nova estrutura de dados chamada de *efficient versioned search trees* (desenvolvida especialmente para este sistema) para suportar de forma eficiente operações atómicas sobre ficheiros e o mapeamento dos ficheiros com os nomes. Assim quando uma sequência de operações

[1] Electrically-Erasable Programmable Read-Only Memory
[2] Memória de acesso aleatório (do inglês Random Access Memory).

é completada, realizado um *commit*, a árvore é congelada e torna-se somente acessível para leitura, enquanto que é formada uma nova árvore pela duplicação de nós da anterior.

Cada transacções recebe um identificador único no sistema que representa o momento de início da transacção, assim uma transacção com identificador t pode observar todas as alterações realizadas por transacções finalizadas com identificador $t-1$ ou mais baixo. Quando uma transacção inicia é criada uma nova versão da árvore que se mantém acessível para leitura e escrita até que a transacção termine.

2.6.6 *Transaction-Safe* FAT

O sistema de ficheiros FAT baseia-se essencialmente no mapear dos blocos utilizados numa tabela (a FAT) e, com essa informação, é possível determinar a localização de um ficheiro. Neste sistema, como em muitos outros, existe a possibilidade das operações de alteração de dados sejam interrompidas a meio, o que pode levar o sistema de ficheiros para um estado inconsistente.

Figura 2.9: Diagrama da arquitectura da *Transaction-Safe* FAT

Como forma de solucionar este problema foi desenvolvido o *Transaction-Safe* FAT (TFAT) [MSD08]. O TFAT mantém duas cópias independentes da tabela FAT (Figura 2.9), a FAT#2 que é utilizada como versão temporária , enquanto que a FAT#1 é utilizado como versão estável.

As alterações realizadas na FAT#2 são aplicadas após todas as operações da transacção tenham sido realizadas com sucesso. Se a transacção falhar, o sistema de ficheiros fica de acordo com a visão anterior ao inicio da transacção. Quando a transacção termina é realizada a operação de substituição da FAT#1 pela FAT#2.

Embora este sistema não ofereça primitivas transaccionais às aplicações, o TFAT oferece ao utilizador a garantia de que as alterações aos ficheiros vão ser aplicadas de forma atómica, a cada escrita ou quando realizado um *close* do ficheiro, contudo não suporta isolamento pois somente existe uma FAT como versão temporária.

Por norma somente as alterações a uma directoria e a tabela FAT são recuperáveis no decorrer de uma transacção, contudo é possível alterar as definições do sistema de maneira a que as alterações do conteúdo de ficheiros sejam realizados em novos blocos, o que permite que também as modificações sobre ficheiros sejam recuperáveis. Para tal

a TFAT reserva um novo bloco, realiza uma cópia do conteúdo antigo e redirecciona a entrada na tabela FAT#2 para este novo bloco.

A TFAT exige hardware específico que permita realizar operações sobre blocos de forma atómica e é algo penalizante em termos de desempenho pois são realizadas operações de cópia de dados. Assim este sistema permite garantir a atomicidade e consistência em relação a meta-dados como a dados. Embora seja limitativo quanto a concorrência e tenha um custo acrescido devidas às cópias realizadas, este sistema demonstra que é possível de forma simplista fornecer algumas garantias transaccionais no acesso a ficheiros.

2.6.7 *Transactional* NTFS

A versão do sistema de ficheiros NTFS com suporte transaccional (TxF) [Mic09] é um componente recente, que surgiu na versão 6.0 do sistema operativo Windows (Vista). Este traz para o domínio das aplicações o conceito de transacções no sistema de ficheiros.

O TxF permite que tanto os ficheiros como as directorias sejam modificadas, criadas e apagadas de uma forma atómica, permitindo com isto evitar problemas de inconsistência de dados relacionados com falhas. Assim um conjunto de operações sobre ficheiros ou directorias são aplicadas somente em caso de sucesso da transacção ou em caso de falha nenhuma delas é aplicada.

O TxF está implementado com recurso ao gestor transaccional do núcleo do sistema, o *Kernel Transaction Manager* (KTM). É este que fornece a noção de transacção para objectos do *kernel* do sistema. O KTM é um gestor transaccional implementado ao nível do sistema e que permite utilizar a noção de transacção tanto ao nível de sistema como de utilizador. O KTM, embora pertença ao núcleo do sistema operativo, permite desenvolver aplicações de nível utilizador que podem recorrer à utilização de transacções. Para além disso permite também a gestão de transacções distribuídas.

O KTM recorre a um subsistema de *logging* para implementar a noção de atomicidade. O *Common Log Fle System* (CLFS) é um subsistema que permite o registo tanto de eventos como de dados. No caso especifico do TxF, o CLFS é utilizado para registar as alterações realizadas antes destas serem efectivamente aplicadas ao sistema de ficheiros, sendo assim um sistema de actualizações em diferido.

Embora o NTFS já implemente de forma interna a noção de transacção, suportada pela utilização de *jornaling* para registar as operações de baixo nível, como por exemplo a escrita de blocos, o TxF expande essas capacidades realizar operações atómicas sobre um ficheiro, para operações atómicas sobre múltiplos ficheiros locais ou remotos.

Isolamento

O TxF oferece um nível de isolamento *read commited*, o que se traduz, no caso de uma transacção abrir um ficheiro para leitura e, ao mesmo tempo, outra mantiver o mesmo ficheiro aberto para escrita, a primeira não vai ver nenhuma modificação que seja entretanto realizada pela segunda.

Uma transacção que abra o ficheiro para escrita vê a versão mais recente dos dados, que incluem as alterações realizadas na transacção corrente. Unicamente pode existir uma transacção com direitos de escrita sobre um ficheiro. Os acessos a ficheiros não transaccionais são sempre bloqueados.

Características

Para a utilização das funcionalidades transaccionais deste sistema estão disponíveis um conjunto de funções, similares às oferecidas para manipulação de ficheiros, somente diferem no facto de ser necessário mais um parâmetro, o identificador da transacção. Existe também a possibilidade de, de forma implícita tornar, transaccionais todas as chamadas ao sistema de ficheiros. Tal é conseguido com o lançar do programa por outro que realiza o inicio e fim da transacção.

Embora ainda existam alguma limitações neste sistema, como seja, a sua exclusividade ao NTFS e ao sistema operativo Windows, bem como a limitação de somente uma transacção poder escrever sobre um mesmo ficheiro, este sistema constitui um grande avanço na integração das noções transaccionais no seio de um sistema operativo.

2.6.8 *Transactional File System*

Com o objecto de construir um sistema de ficheiros com uma semântica transaccional foi desenvolvido o *Transactional File System*(TFS) [Mar08], um sistema que disponibiliza para o domínio da aplicação uma interface transaccional para acesso ao sistema de ficheiros. Este sistema está implementado sobre a forma de uma biblioteca que é ligada com a aplicação, implementando uma camada intermédia entre a aplicação e um sistema de ficheiros. Este oferece ao programador um conjunto de funções de acesso a ficheiros muito similares, baseadas (mas diferentes) das disponibilizadas pelas normas POSIX 1003.1 [POS92], tentando assim ser de fácil utilização.

Como a implementação é disponibilizada como uma biblioteca, isto implica que cada aplicação que a utilize possui um motor transaccional independente e que, portanto, duas aplicações que acedam a ficheiros comuns não obtêm entre sim uma semântica transaccional. No entanto, duas aplicações independentes não vão conseguir aceder de forma concorrente, a uma mesmo ficheiro através do TFS, visto que a bibli-

Figura 2.10: Diagrama da ligação entre o TFS e o sistema de ficheiros real

oteca realiza a abertura de um ficheiro com *locks* de escrita e, assim, somente uma das aplicações vai conseguir progredir nas suas operações sobre o ficheiro.

De forma a possibilitar o isolamento, o TFS mantém em memória uma cópia dos blocos do ficheiro que foram alterados, dando-lhes a designação de blocos virtuais cuja dimensão pode ou não ser igual a dimensão dos blocos do sistema de ficheiros. Estes blocos virtuais constituem a unidade transaccional base do sistema, ou seja, estes blocos podem ser acedidos de forma concorrente e as validações transaccionais acontecem sobre estes blocos. Quando a transacção termina os blocos são escritos para o ficheiro e a memória libertada.

Arquitectura

O sistema está estruturado em dois módulos principais, como representado na Figura 2.11 (extraida de [Mar08]), o *TFS Core* e o *Cache Manager*, em que cada um é responsável por uma determinada lógica da biblioteca e pela implementação de um conjunto de serviços.

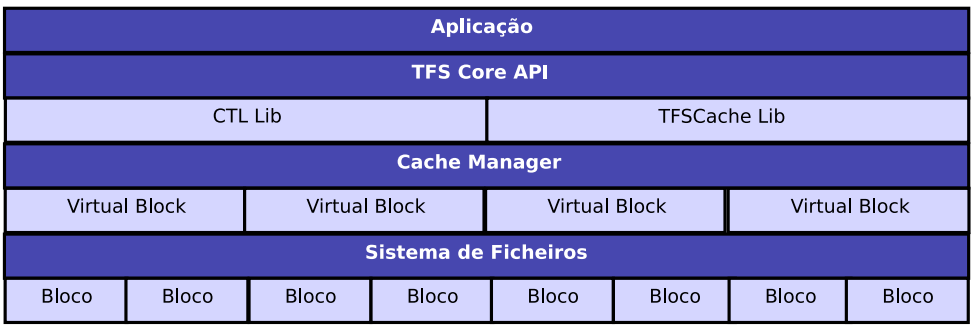


Figura 2.11: Diagrama da arquitectura do TFS

TFS Core Este módulo fornece todas as funções necessárias para que o programador possa agregar os acessos a ficheiros dentro de blocos transaccionais. Este módulo mantém dois tipos principais de informação:

- Informação das transacções em curso, o que inclui o identificador da transacção, o *write set* de blocos e informação sobre os ficheiros acedidos pela transacção.

- Meta-informação relacionada com os ficheiros abertos, que normalmente é gerida pelo sistema operativo, tais como o tamanho do ficheiro.

Cache Manager Este componente constitui a parte de mais baixo nível da biblioteca, sendo responsável pela gestão dos blocos virtuais relativos a ficheiros, disponibilizando métodos para ler e escrever o conteúdo de um ficheiro para memória.

Cada ficheiro é identificado com um identificador construído pelo *id* do *device* e pelo número do *inode*, o que o torna único em todo o sistema operativo. Assim, a cada ficheiro aberto está associada, para além do ID, uma lista de blocos virtuais que constituem o núcleo da funcionalidade de *caching*.

Nesta perspectiva, o bloco virtual define a granularidade do sistema, pois cada um é acedido transaccionalmente e, assim, cada bloco virtual pode ser acedido de forma independente sem perturbar a acções realizadas nos demais blocos.

A dimensão do bloco virtual constitui um *trade-off* no TFS pois, por um lado, caso este valor seja muito pequeno, vão ser necessárias mais computações na validação do contexto transaccional e usado mais espaço/memória para manutenção da meta-informação, no caso contrario, vai-se limitar a concorrência no acesso ao dados.

Este gestor implementa um sistema de *journaling* em disco, que permite garantir a propriedade de atomicidade sobre os meta-dados. Para garantir que os acessos aos blocos virtuais são realizados de forma transaccional, este sistema recorre internamente à utilização de um framework de memória transaccional por software, o CTL [Cun07]. De forma a simplificar a implementação, existe um byte que representa todo o bloco virtual e assim cada acesso de leitura ou escrita a um endereço dentro de um bloco vai implicar um leitura transaccional registada perante o CTL como uma operação de leitura e escrita sobre esse byte de representação. Assim, quando duas tarefas escrevem no mesmo bloco virtual de um mesmo ficheiro, vão registar perante o CTL duas operações de escrita sobre uma mesma posição de memória, o byte que representa este bloco, e quando se proceder à tentativa de realizar *commit*, as duas transacções vão validar os seus *read set* e *write set* e se existir um conflito é relançada um das transacções.

Características O TFS possibilita a uma aplicação a realização de operações concorrentes de leitura e escrita, sobre blocos de um mesmo ficheiro, onde essa operações são realizadas de forma transaccional.

O TFS é um protótipo e não oferece todas as funcionalidades desejáveis [Mar08]. Exemplos são a truncatura de ficheiros, que não está implementada, bem como a limitaçãoquanto ao número de blocos virtuais possíveis pois se este valor for muito elevado vão ser geradas muitas entradas no *write set* do CTL que não consegue ainda acomodar

tal esforço. O CTL ainda ainda não suporta *read/write set* de tamanho variável. Esta é uma limitação para a implementação de transacções de longa duração.

Este protótipo não é de fácil utilização, pois requer que todas as chamadas da aplicação sobre ficheiros tenham de ser reescritas de forma a utilizarem a biblioteca. Como consequência não existem testes sobre aplicações reais que possam demonstrar a real aplicabilidade do sistema.

2.6.9 Sumário

Muito embora sejam aqui apresentados sistema de ficheiro que já apresentam algum tipo de suporte transaccional, este não oferecem em pleno todas as capacidades que pretendemos.

Este os sistemas analisados existem alguns que são para um domínio especifico (por exemplo hardware), não sendo aplicáveis a sistemas de uso comum. Deste tipo são exemplo o TFAT, o Perdis e o TFFS.

Alguns deles recorrem a uma base de dados para guardar e oferecer as propriedades transaccionais, como são o DBFS, o Inversion e o Amino Este facto implica uma transformação das operações sobre um sistema de ficheiros, bem como a utilização de um sistema (base de dados) que foi desenhada para outros fins.

Embora o TxF seja o mais completo em termos de capacidade é limitativo quanto a concorrência que permitem pois que limita o acesso a escrita a só um escritor. Para além desse facto, o TxF é um sistema proprietário do qual não se sabe todos os detalhes.

Outro ponto em atenção é a necessidade de alterar a aplicação para que se possa usufruir de um sistema com suporte transaccional, pois existem sistemas que implicam a alteração da aplicação, como são exemplo os TFS e o TxF.

3

Sistema de ficheiros transaccional

Neste capitulo, apresentamos os requisitos para um sistema de ficheiros transaccional, propondo-se de seguida um sistema que satisfaz esses requisitos.

3.1 Requisitos

Propomo-nos a oferecer ao programador a possibilidade de beneficiar de um acesso transaccional aos dados mantidos num sistema ficheiros.

3.1.1 Propriedades

O sistema deve permitir ao utilizador agregar um conjunto de acessos ao sistema de ficheiros num único bloco. Este bloco confere um contexto transaccional aos acessos que agrega. Todos os dados e meta-dados acedidos neste contexto têm uma semântica transaccional, sendo vistos como uma transacção. Pretende-se que as operações englobadas por este bloco beneficiem das propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade).

Atomicidade

O bloco que agrupa um conjunto de acessos ao sistema de ficheiros tem de ser considerado atómico, tanto no acesso a dados como a meta-dados. Isto que dizer que as operações que fazem parte do bloco não podem ser existir individualmente.

As operações que constituem o bloco/transacção são todas executadas em caso de sucesso ou nenhuma dela é executada no caso de falha. Esta propriedade confere a este bloco a impossibilidade de se dividir uma ou um subconjunto das operações no bloco.

Por exemplo, no contexto de uma transacção uma aplicação altera n ficheiros diferentes. A expectativa da aplicação é que as alterações sejam todas aplicadas em todos os ficheiros, ou em caso de falha, nenhuma delas seja aplicada.

Isolamento

Define-se isolamento como a propriedade de impor uma separação dos acessos e efeitos entre os blocos/transacções concorrentes. Uma operação realizada numa determinada transacção não pode ser visível para as operações que façam parte de outra transacção que esteja a ser executada em concorrência.

Duas aplicações que executem concorrentemente vão ter visões diferentes do conteúdo do sistema de ficheiros. Esta separação deve de incluir o mapeamento do nome, bem como a própria estrutura de directorias.

Por exemplo, supondo que, no contexto de uma transacção, uma aplicação altera o nome de um determinado ficheiro. Essa alteração só ficara visível para as outras aplicações (quer seja num contexto transaccional ou não) quando a transacção tiver sucesso. Somente no contexto da transacção onde a alteração é realizada é que o novo nome é visível antes do final da transacção. Igual comportamento se espera se uma aplicação criar ou remover um ficheiro de uma directoria.

Consistência

A consistência pode ser vista a dois níveis distintos. Um primeiro nível correspondente ao sistema de ficheiros e um segundo ao nível das aplicações que utilizam esse mesmo sistema de ficheiros.

Um sistema de ficheiros guarda informação, a que chamamos de dados. Para fazer a gestão desses dados é necessário informação de gestão, os meta-dados. A informação mantida nos meta-dados tem de ser correcta e consistente para garantir o correcto processamento dos dados.

Um exemplo de uma possível inconsistência na visão do sistema de ficheiros por parte de uma aplicação é a seguinte: no decorrer de uma transacção é alterada a dimensão de um ficheiro, para um valor maior relativamente ao actual; no mesmo contexto e ao ler o conteúdo do ficheiro obtém-se um erro de final de ficheiro para uma posição

onde devia de agora ser possível aceder. Esta é uma situação de inconsistência que não deve de ser possível.

Ao nível da aplicação existe outra visão de consistência. Uma aplicação pode criar uma dependência entre vários ficheiros. Mas a gestão da consistência a este nível fica a cargo da própria aplicação.

Durabilidade

Algumas das operações inseridas num bloco transaccional vão provocar alterações no sistema de ficheiros. Essas alterações quando aplicadas não se podem perder mesmo que ocorra uma falha. Ao terminar um contexto transaccional com sucesso, as alterações realizadas neste necessitam de ser aplicadas de forma persistente. Esta garantia de durabilidade é já oferecida por muitos dos actuais sistemas de ficheiros.

3.1.2 Delimitação

Existe um conjunto de serviços padrão (POSIX 1003.1) que envolvem o sistema de ficheiros que podem ser utilizados por uma aplicação. Pretende-se que a invocação desses serviços possa ter um contexto transaccional. Para conseguir alcançar este objectivo é necessário definir quais as chamadas/invocações que formam uma transacção (um bloco transaccional).

Tem de se disponibilizar uma forma de definir o início e final de um bloco de chamadas que vai ser tratado como uma transacção. A definição das fronteiras de uma transacção pode ser feita de duas formas distintas, implicita e explicitamente.

Definição de forma explicita

Este forma consiste na utilização de serviços adicionais, para além dos já disponibilizados pela interface POSIX do sistema de ficheiros. Este novos serviços vão delimitar o início e fim de um contexto transaccional.

A ideia base do seu funcionamento é expressa pelo bloco de código da Figura 3.1. Neste bloco de código faz-se uso das primitivas $fstart()$ e $fcommit()$ para marcar o início e final respectivamente da transacção.

Como se tratam de serviços adicionais, estes têm de ser invocados directamente pelo programador da aplicação final. Este facto permite ao programador, por um lado, ter a noção clara de quais os blocos com contexto transaccional e decidir quando deve ser feita a sua utilização. Mas, por outro lado, impede que uma aplicação já desenvolvida possa utilizar as capacidades do sistema, pois este modelo de utilização implica a

```
TX* tx = fstart();
FILE* f = fopen("/directory/file.txt", "w");
fprintf(f, "%s", "Line of Text");
fclose(f)
fcommit(tx);
```

Figura 3.1: Exemplo de delimitação explicita

alteração do código fonte das aplicações.

Definição de forma implícita

A forma de delimitação implícita consiste em inferir os limites do bloco transaccional. Esta inferência é realizada tendo como base as invocações realizadas dos serviços já disponibilizados pela interface do sistema de ficheiros. Isto pode ser conseguido associando a uma chamada a marca de início e a outra chamada a marca de final.

O meio mais intuitivo de implementar esta forma de delimitação é associar à invocação do $open$ (de um ficheiro) ao início de uma transacção. Da mesma forma associar à invocação do $close$ ao respectivo final da transacção. Desta forma quaisquer operações (de acesso ao sistema de ficheiros) realizadas entre um $open$ e um $close$ formam uma transacção.

Contudo, podem existir vários ficheiros abertos em simultâneo. Neste caso considera-se que o primeiro $open$ marca o início e o último $close$ marca o final. Para tal são contabilizados quantos ficheiros abertos há abertos em cada instante. Quando é aberto o primeiro ficheiro e o contador passa de 0 para 1, considera-se o início de uma transacção. Subsequentes aberturas ($open$) de ficheiros irão incrementar este contador. Quando o contador transita de 1 para 0 (foi fechado o último ficheiro) então considera-se que a transacção termina. Um exemplo deste funcionamento é expresso pelo bloco de código da Figura 3.2.

```
FILE* f1 = fopen("/directory/file1.txt", "w"); // contador = 1, inicio da transacção
FILE* f2 = fopen("/directory/file2.txt", "w"); // contador = 2
fprintf(f1, "%s", "Line of Text");
fprintf(f2, "%s", "Line of Text");
fclose(f1);                                    // contador = 1
fclose(f2);                                    // contador = 0, fim de transacção
```

Figura 3.2: Exemplo de delimitação implícita

Este modelo de delimitação implícita tem como vantagem não implicar a alteração das aplicações já existentes. Assim, embora se perca algum controlo sobre a delimita-

ção das transacções, permite-se uma rápida e fácil utilização do sistema de transacções no contexto de acesso ao sistema de ficheiros.

3.1.3 Controlo de concorrência

Como pode existir mais do que uma transacção a executar ao mesmo tempo no sistema (concorrência) as questões de isolamento e resolução de conflitos têm de ser equacionadas. As transacções partilham um recurso que é o sistema de ficheiros, sendo este composto de meta-informação e dados de vários tipos. Para suportar concorrência no acesso a este recurso é necessário suporte para detecção de conflitos e uma política de resolução dos mesmos.

Assim, têm de ser considerados diferentes tipos conflitos, dependendo dos diferentes tipos de ficheiros existentes. Vamos somente discutir os possíveis conflitos em ficheiros de dados e directorias.

Ficheiros de dados

O acesso a um ficheiro pode ser considerado em diferentes níveis de granularidade. Pode ser considerado conflito o caso em que duas transacções acedem exactamente ao mesmo ficheiro, e neste caso temos uma granularidade grossa. Esta granularidade limita a concorrência entre transacções, por isso há que reduzir a unidade de conflito (grão)). Definindo o bloco do ficheiro como unidade de conflito temos uma granularidade mais fina. Neste caso permite-se que duas transacções acedam ao mesmo ficheiro, desde que a dados localizadas em blocos distintos.

Directorias

As directorias são compostas por uma sequência de entradas de nomes e respectivos identificadores de ficheiro (*inode*). Assim, podemos considerar que duas transacções entram em conflito se ambas alterarem alguma das entradas da directoria mas isto é considerar um nível de granularidade elevado. As situações de conflito entre alterações nas directorias ocorrem pela alteração do mesmo nome que identifica a entrada. Como tal a situação de conflito deve de ser considerada ao nível da entrada e não ao nível de toda a directoria.

Resolução de conflitos

Depois de se detectar um conflito entre duas transacções é necessário realizar alguma acção a fim de evitar esta situação. Para tal é necessário definir uma politica

de resolução de conflitos. É necessário definir como são resolvidos os conflitos, o que implica que uma das transacções vai ter de ser abortada.

A escolha de qual a transacção que vai de ser abortada pode ser feita por vários critérios, como seja o tempo de execução, o número de operações realizadas ou simplesmente a transacção que detecta o conflito aborta. Por outro lado, a resolução de um conflito pode ser decidida unilateralmente por uma das transacções, que ao detectar o problema decide terminar e não aplicar as suas alterações. Também pode ser determinada uma solução por uma terceira entidade. Esta terceira entidade tem como única funcionalidade a de decidir qual a transacção que deve de ser abortada em caso de conflito.

3.2 Arquitectura

Tendo como base os requisitos apresentados anteriormente, vamos especificar a arquitectura de uma solução. Esta arquitectura tenta dar uma boa resposta às dificuldades que se apresentam, bem como suportar os requisitos apresentados.

3.2.1 Visão geral

O nosso sistema de ficheiros está situado entre as aplicações que acedem aos ficheiros e um repositório de dados. As aplicações actuam sobre um interface padrão para manipulação e acesso a ficheiros, enquanto que o repositório de dados mantém o conteúdo final dos dados utilizados pelas aplicações.

A arquitectura geral do sistema está divida na sua essência em três partes distintas, tal como ilustrado na Figura 3.3, que vão ser descritas seguidamente.

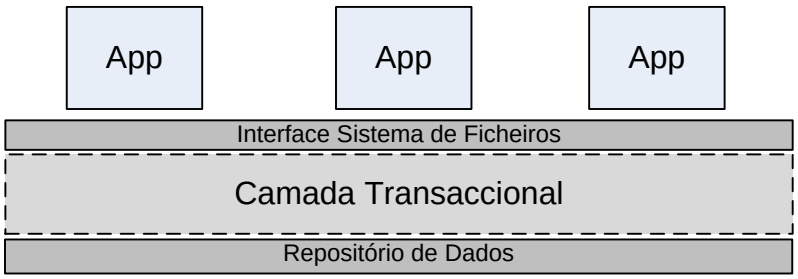


Figura 3.3: Arquitectura geral do sistema

O nosso sistema de ficheiros funciona como um *middleware* que fornece à aplicação uma visão transaccional das suas alterações em ficheiros e directorias. Para conseguir fornecer esta visão, o sistema tem de manter associada a cada aplicação a informação de quais as alterações efectuadas. A informação associada a cada aplicação serve para

aplicar essas mesmas alterações no repositório de dados. Essa alterações podem ou não estar num contexto transaccional.

A camada superior do sistema disponibiliza para as aplicações uma completa interface de um sistema de ficheiros. Oferece às aplicações um conjunto de serviços, como seja, o ler um ficheiro, obter as entradas de uma directa ou modificar as permissões de um ficheiro. Como vai ser requerido que cada transacção esteja associada a uma aplicações, é esta camada que permite realizar este mapeamento.

A segunda camada, a camada transaccional, acrescenta a noção de transacção, bloco de operações sobre ficheiros, as noções do sistema de ficheiros. Neste nível é necessário oferecer suporte para a noção de transacção para a sua definição e aplicação. Assim esta camada tem de permitir agregar um conjunto de operações temporariamente até serem aplicadas e permitir também a separação das diferentes aplicações que possam estar em execução. Deve possibilitar às aplicações uma visão temporária das alterações realizadas num contexto transaccional.

Por fim a camada do sistema onde são guardadas as versões definitivas dos dados e que oferece persistência aos mesmos. O repositório de ficheiros é o local onde são colocadas as versões finais dos dados. Aqui vão estar sempre as versões definitivas dos dados e meta dados, como seja, por exemplo, os conteúdos dos ficheiros. Este é um recurso partilhado por todas as transacções do sistema.

Transacção

Os sistemas de ficheiros tradicionalmente guardam a informação de cada ficheiro em estruturas de dados de gestão. O *inode* é a estrutura de dados que guarda para cada ficheiro a sua meta-informação. Cada *inode* é identificado por um identificador único no sistema. Algumas das informações tipicamente guardadas por um *inode* incluem o dono e grupo, modos de acesso, número de links, tipo do ficheiro, entre outras. Assim um *inode* representar um ficheiro guardado no sistema.

Sendo uma transacção um conjunto de alterações sobre ficheiros, a informação mantida num *inode* vai pode ser alterada durante a execução de uma transacção. Assim a transacção vai conter um conjunto de *inodes* modificados durante o contexto transaccional. Tal ideia é ilustrada na Figura 3.4.

Mas o *inode* contém somente a informação sobre os ficheiros, não os seus conteúdos. Cada ficheiro pode ser de um determinado tipo e isso dá indicação sobre o seu conteúdo. Nesta arquitectura vamos considerar que um ficheiro pode ser de três tipos diferente: ficheiro de dados, directoria e link simbólico.

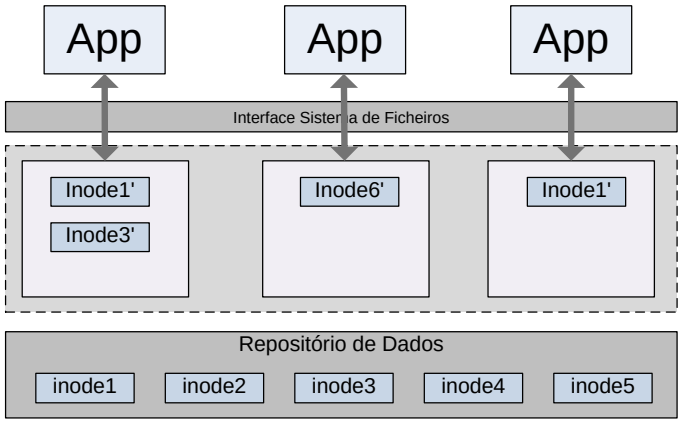


Figura 3.4: Esquema particular de uma transacção neste contexto

3.2.2 Principais componentes

Os diversos componentes do nosso sistema de ficheiros transaccional estão ilustrados na Figura 4.3.

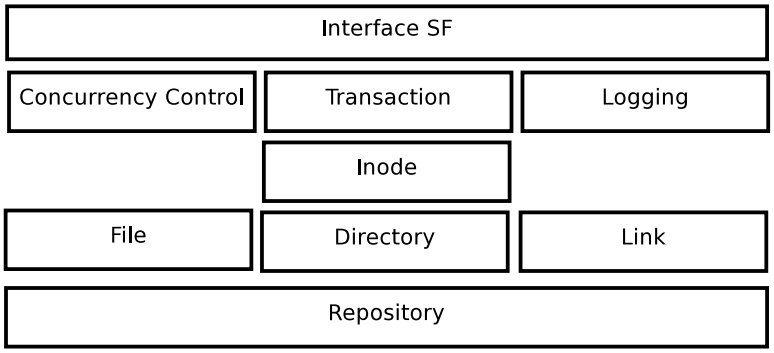


Figura 3.5: Arquitectura de módulos detalhada

De seguida detalhamos em mais pormenor as características de cada um estes componentes.

Interface File System

Este módulo é o ponto de comunicação entre as aplicações e o sistema implementado. Este implementa um interface de serviços de sistema de ficheiros. Este conjunto de serviços é disponibilizado às aplicação para que estas acederem aos ficheiros. Os serviços são serviços padrão (POSIX 1003.1) para que a aplicações não tenham de alterar a sua implementação.

A responsabilidade de receber o pedido (explicitamente) ou de os inferir (implicitamente) cabe também a este componente. Explicitamente, através de serviços adicionais

ao padrão, implicitamente através da contagem de ficheiros abertos.

Para suportar a associação entre uma aplicação e uma transacção, e como forma de delimitar implícita uma transacção, este módulo identifica cada fluxo de execução (*thread*). Desta forma um fluxo de execução tem no máximo uma transacção associada.

Transaction

Este módulo representa uma transacção. Neste contexto, uma transacção é representada por um conjunto de alterações/operações sobre ficheiros. O módulo *transaction* disponibiliza todos os serviços que podem ser invocados por uma aplicação para acesso a ficheiros. Para além disso, fornece serviços de início e finalização da transacção, onde podem ou não ser aplicados os seus efeitos.

O módulo *transaction* guarda um conjunto de *inodes* que tenham sido modificados ou acedidos no decorrer da transacção. Esta informação é utilizada para saber se determinado *inode* já foi manipulado no decorrer de uma transacção e ir buscar a nova versão, bem como, para aplicar as alterações no final da transacção.

Concurrency Control

O módulo *concurrency control* é responsável por tratar as situações de conflito que possam acorrer. Este oferece serviços de ordenação de transacções baseado num relógio global. Como tal, cada transacção vai ter uma estampilha temporal única.

Para conseguir realizar um controlo de concorrência entre as várias transacções foi escolhido um mecanismo de relógio global. Este mecanismo consiste na existência de um relógio lógico global a todo o sistema. Todos os objectos presentes no sistema vão ter uma estampilha que deriva desse relógio. Uma transacção quando inicia lê para si o valor actual do relógio global. O valor da estampilha presente no objecto corresponde ao valor da estampilha da transacção que o modificou.

Assim, quando no contexto de uma transacção se vai ler um *inode*, é verificado se o valor da estampilha do *inode* é menor ou igual ao valor da estampilha da transacção. Se esta condição se verificar então a transacção prossegue, no caso contrário a transacção passa ao estado *abort*. Este algoritmo permite detectar quando um objecto lido numa transacção não esta consistente. Evita assim que uma transacção execute num estado inconsistente.

Mas como estamos a dividir *inodes* em três tipos diferentes (ficheiros, directorias e *links* simbólico) vamos ter de os tratar de formas diferentes, pois podem ter diferentes níveis de tolerância à concorrência. Admite-se uma granularidade de concorrência de

um bloco para os ficheiros de dados, enquanto que para as directorias será o identificador de cada entrada, o nome. Como tal tem de existir uma estampilha para cada nível (estampilhagem para blocos de ficheiros e entradas em directorias). Relativamente ao *links* simbólicos, estes podem ser tratados como um todo, pois os *links* simbólicos são imutáveis.

Recai sobre este módulo também a responsabilidade de decidir qual das transacções deve de ser abortada em detrimento de outra.

Logging

Uma alteração de um nome de uma entrada numa directoria é dependente de alterações nas suas directorias pai. As directoria formam uma hierarquia, pelo que a criação de nomes ou alteração dos mesmos tem de ser realizada segundo a sua dependência. Por exemplo, é necessário criar primeiro a directoria pai e só depois é possível criar a os respectivos filhos.

Como resposta para este problema, a solução encontrada é de registar as alterações de toda a estrutura de directorias. Este módulo gere um *log* onde são registadas as alterações realizadas sobre os nomes da estrutura de directorias no decorrer de uma transacção. Cada entrada no *log* contém o tipo de operação realizada, qual o identificador do *inode* da directoria pai, o identificador do *inode* modificado e os parâmetros das alterações.

Este *log* de operações permite a aplicar as alterações de uma forma correcta. Quando se pretende aplicar as alterações à estrutura de directorias é percorrido o *log* e realizada a operação indicada em cada entrada.

Inode

Este módulo gere a meta-informação de um ficheiro no sistema, guardando as seguintes informações:

inode number
protection
number of hard links
user ID of owner
group ID of owner
total size
time of last access
time of last modification

time of last status change

Disponibiliza serviços para alteração destas informações, bem como serviços para a aplicação das modificações no repositório de dados.

File

O módulo file oferece serviços de manipulação do conteúdo de um ficheiro, como é o *read* e *write*.

Como forma de estruturar a informação guardada em ficheiros é realizando a divisão do seu conteúdo em blocos de bytes. Para preservar os dados alterados no decorrer de uma transacção, realizou-se a duplicação dos blocos do ficheiro que tenham sido alterados durante a transacção. Isto é muitas vezes designado técnica como um técnica de *copy-on-write*.

Quando é realizado um *write* é verificado se já existe uma cópia privada dos blocos correspondentes. Caso não exista é criada então cópia do conteúdo original e realizada a escrita sobre esta cópia. A mesma verificação ocorre no caso de um *read*, sendo preferencialmente acedidos os blocos privados existentes.

Como são só guardados os blocos com conteúdo privado, a aplicação das modificações num ficheiro realiza-se pela escrita desses mesmo blocos na versão original do ficheiro. São iterados todos os blocos privados e o seu conteúdo é escrito para a respectiva localização no ficheiro original.

Directory

Este módulo representa uma directoria no sistema, como tal oferece suporte para a gestão de entradas numa directoria. Para cada directoria é guardada uma lista das suas entradas e é essa lista que é manipulada. Estas entradas são utilizadas para as operações de pesquisa.

A entidade directoria é composta por esta lista, obtida no momento da abertura da mesma directoria. A entidade directoria permite adicionar e remover entradas, bem como retornar a listagem da mesma.

É também guardada uma outra lista, dos nomes modificados (inseridos ou removidos) nas operações de manipulação da directoria. Esta lista é utilizada para gestão de conflitos de nomes e verificada na fase de validação da transacção.

Link

Em relação aos *links* temos de considerar que existem dois tipos de *links*, os *hard* e os simbólicos. Um *hard link* é uma referencia para um ficheiro numa directoria, ou seja, é uma entrada numa directoria. Por este factor não é necessário ter um componente para guardar esta informação, pois a componente *Directory* já trata deste caso.

Contudo, um *link* simbólico é um pouco diferente, pois trata-se de um ficheiro com conteúdo especial, o caminho até ao destino. Este módulo representa um *link* simbólico no sistema. Basicamente guarda a informação de qual o caminho de destino do *link*. Este caminho nunca é alterado desde o momento da criação.

Num sistema de ficheiros um *link* simbólico não pode ser modificado, por essa razão a aplicação das modificações (trata-se unicamente do caso em que é criado um novo *link* simbólico) é realizada pela simples operação de criar um novo *link* simbólico, no repositório de dados, que aponta para o caminho de destino.

Repository

Este módulo representa um repositório final de dados, ou seja, o local onde os ficheiros são guardados na sua versão final. Este oferece os serviços de um sistema de ficheiros tradicional e adicionalmente permite criar e utilizar ficheiros temporários.

3.3 Comportamento

Nesta secção vamos apresentar alguns dos comportamentos definidos para situações relevantes, de forma a ilustrar o comportamento do sistema. Estes comportamentos são apresentados sobe a forma de algoritmos com o objectivo de facilitar a sua compreensão.

3.3.1 Transacção

Para ajudar a perceber o intuito da entidade transacção vamos descrever o comportamento no caso em que se adiciona uma entrada a uma directoria no Algoritmo 1.

Algorithm 1 Adicionar uma entrada a uma directoria

transaction_link(transaction, parent_inode, name, inode):

```
if transaction_state(transaction) == ABORT then
  return EABORT
else
  if transaction_check_inode(transaction, parent_inode) == FALSE then
    transaction_load_inode(transaction, parent_inode)
  end if
  parent = transaction_get_inode(transaction, parent_inode)
  if inode_type(parent)! = DIRECTORY then
    return ENOTDIR
  end if
  if directory_add_entry(parent, name, inode) == FALSE then
    return EEXIST
  end if
end if
```

Primeiro é necessário verificar o estado da transacção, pois se a transacção estiver no estado de abortada a função deve de devolver o erro correspondente. Depois é verificado se a informação do *inode* pai já esta presente na transacção e, se não estiver, tem de ser carregada a partir do repositório de dados. De seguida é verificado o tipo do *inode* dado como pai, pois tem de ser verificado se este corresponde a uma directoria. No caso de não ser uma directoria é devolvido o erro correspondente. Por fim é adicionada a nova entrada à lista de entradas na directoria. Esta operação também pode falhar, no caso em que já existe uma entrada com o mesmo nome.

Outro evento que é importante no sistema é a aplicação das alterações, onde é também necessário validar a transacção. Esta operação de aplicação é descrita no Algoritmo 2.

Algorithm 2 Aplicar as alterações de uma transacção

transaction_apply(transaction):

if $\left\{ \begin{array}{l} transaction\_state(transaction) == \texttt{ABORT and} \\ transaction\_validade(transaction) == \texttt{FALSE} \end{array} \right\}$ **then**
  $return$ EABORT
else
  $transaction\_apply\_entry\_log(transaction)$
  for all $inode \in transaction\_inodes(transaction)$ **do**
    $inode\_apply(inode)$
    if $inode\_type(inode) == \texttt{FILE}$ **then**
      $file\_apply(inode)$
    else if $inode\_type(inode) == \texttt{DIRECTORY}$ **then**
      $directory\_apply(inode)$
    else if $inode\_type(inode) == \texttt{SYM\_LINK}$ **then**
      $sym\_link\_apply(inode)$
    end if
  end for
end if

Inicialmente é necessário verificar o estado da transacção e realizar a sua validação, e caso esta esteja marcada como abortada, é retornado o erro correspondente. Caso contrario são primeiramente aplicadas as alteração guardadas pelo *log* seguidas das alterações de todos os *inodes* mantidos na transacção. Primeiro são aplicadas as alterações das informações mantidas no próprio *inode*, seguidas das alterações especificas para cada tipo de conteúdo.

3.3.2 Ficheiro

O comportamento mais relevante no casos dos ficheiros é a manipulação do seu conteúdo, neste caso como lidar com as cópias privadas, separando-as do conteúdo original. Os procedimentos realizados suportar uma leitura de uma dada quantidade de bytes a partir de uma determinada posição de um ficheiro aberto é apresentado no Algoritmo 3.

Algorithm 3 Leitura de um ficheiro

```
file_read(file, inode, offset, buffer, size):
  block ⇐ offset/block_size
  last_block ⇐ min(inode.size, (offset + size))/block_size
  size_readed ⇐ 0
  while block ≤ last_block do
    if block_have_private_copy(inode, block) then
      block_read_private(inode, block, temp)
    else
      block_read_public(inode, block, temp)
    end if
    if block × block_size ≥ offset then
      start ⇐ offset mod block_size
      memcpy(buffer, temp + start, min(size, block_size − start))
      size_readed = min(size, block_size − start)
    else
      memcpy(buffer, temp, min(size, block_size))
      size_readed = min(size, block_size)
    end if
    buffer ⇐ buffer + size_readed
    block + +
  end while
  return size
```

No caso da leitura é inicialmente calculado o intervalo de blocos que estão envolvidos. Na leitura de todos os blocos desse intervalo, existe dois casos especiais que é, se for o primeiro bloco e se for o último. Caso seja o primeiro pode ser necessário realizar uma copia de bytes parcial a partir do *offset*. Caso seja o último bloco há que só de ter em conta não realizar a cópia de mais bytes do que é necessário.

3.3.3 Directoria

Uma directoria é composta por uma lista de entradas (nome, *inode*). A entidade directoria suporta essencialmente três operações: pesquisar uma entrada, adicionar uma entrada e remover uma entrada. A pesquisa numa directoria é apresentada no Algoritmo 4.

Algorithm 4 Pesquisa na directoria

```
directory_resolve(dir, name):
  entry = list_find(entries, name)
  if entry == NULL then
    return FALSE
  else
    return entry_get_inode(entry)
  end if
```

Neste procedimento é pesquisada a lista onde são mantidas as entradas. Caso não exista uma entrada com o nome dado, é devolvido o erro de não existência da entrada. Se existir é devolvido o respectivo *inode* encontrado.

Outro procedimento é a adição de uma nova entrada a directoria, este é apresentado no Algoritmo 5.

Algorithm 5 Adição de entrada na directoria

```
directory_add_entry(dir, name, inode):
  entry = list_find(entries, name)
  if entry == NULL then
    new_entry ⇐ entry_new(name, inode)
    list_add(entries, new_entry)
    list_add(entries_dif, new_entry)
    return TRUE
  else
    return FALSE
  end if
```

Neste caso é verificada a existência de alguma entrada com o mesmo nome e caso já exista é devolvido o erros correspondente. Caso ainda não exista uma entrada com este nome, é então criada uma nova entrada e inserida na lista de entradas. A nova entrada também é adicionada a lista de diferenças, que vai ser utilizada para detectar conflitos.

4

Implementação

Neste capitulo vai ser discutida, com algum detalhe, a implementação realizada do protótipo do sistema. Vão igualmente ser apresentados os problemas e as soluções encontradas durante o processo de desenvolvimento.

4.1 Visão geral

A solução desenhada foi implementada recorrendo ao *Framework* FUSE [HSP+08]. O FUSE é uma ferramenta que permite o desenvolvimento de novos sistemas ficheiros. A escolha pela utilização do FUSE deveu-se ao facto de este oferecer uma boa infra-estrutura para desenvolver um novo sistema de ficheiros, sendo a confiabilidade desta plataforma garantida pelos inúmeros projectos que dela fazem uso. Para além do FUSE recorreu-se a biblioteca GLib [GTK09] como forma de suporte às estruturas de dados.

Como forma de representar um repositório de informação final vai-se recorrer à utilização de um sistema de ficheiros já existente. O seu enquadramento, relativamente a implementação do FUSE e ao repositório de dados, é ilustrado na Figura 4.1.

Este protótipo foi implementado, em linguagem C, num sistema GNU/Linux, recorrendo à utilização de funções assinaladas como pertencendo à norma POSIX. Desta forma este protótipo pode ser facilmente portado para outros sistemas de operação que respeitem a norma POSIX e para os quais exista uma implementação do FUSE, como é exemplo do sistema operativo Mac OS X.

O TFSoF é posto em funcionamento através da execução de um programa. O programa implementa na sua totalidade o nosso sistema de ficheiros. Este programa re-

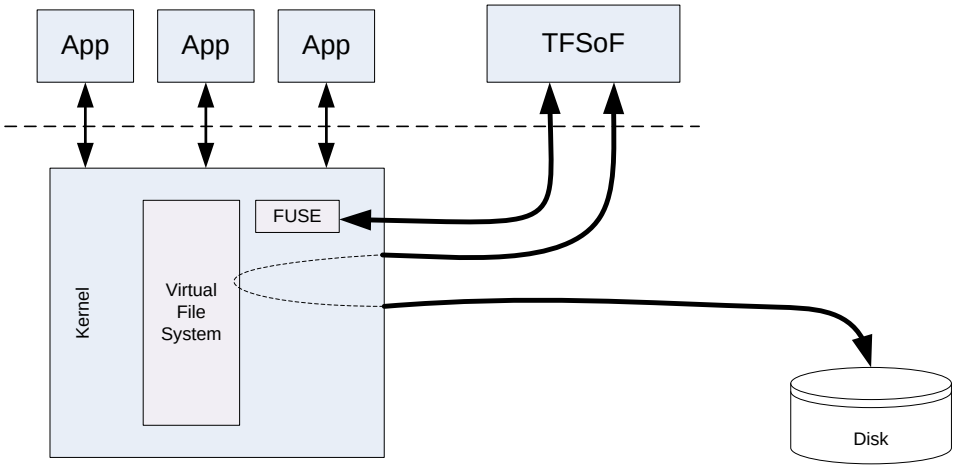


Figura 4.1: Visão geral da arquitectura da implementação

cebe dois parâmetros: ponto de montagem e a directoria base do repositório. O ponto de montagem, que é uma noção dos sistema UNIX, é a localização na árvore de directorias do sistema operativo onde o nosso sistema de ficheiros vai ficar acessível. O ponto de montagem têm de ser uma directoria vazia. A directoria base é a localização na árvore de directorias dos sistema de operação que vai ser base do repositório final de ficheiros.

4.2 Utilização do *Framework* FUSE

O *Framework* FUSE é utilizado para implementar a componente de interface do nosso sistema de ficheiros. Por um lado apresenta-se como uma forma fácil e transparente de disponibilizar o nosso sistema ficheiros às aplicações, e por outro lado, permitindo evitar problemas relativos à implementação.

A implementação de um novo sistema de ficheiros, utilizando o FUSE, faz-se essencialmente pela definição de um conjunto de *handlers*. Este *handlers* são funções que são invocadas como resposta a eventos gerados pelo sistema de operação quando uma aplicação acede ao sistema de ficheiros. As acções e o retorno desses *handlers* definem o comportamento do novo sistema de ficheiros.

Como é ilustrado pela Figura 4.2, uma aplicação invoca um determinado serviço do sistema de ficheiros. Essa invocação é tratada pelo VFS e transmitida ao módulo do FUSE. Depois é o FUSE (módulo + biblioteca) que invoca o *handler* definido para responder ao evento. Nesse *handler*, a implementação do sistema de ficheiros realiza as operações que desejar e devolve a resposta ao FUSE, seguindo esta depois até a aplicação.

O FUSE permite identificar a aplicação que desencadeou a invocação do *handler*,

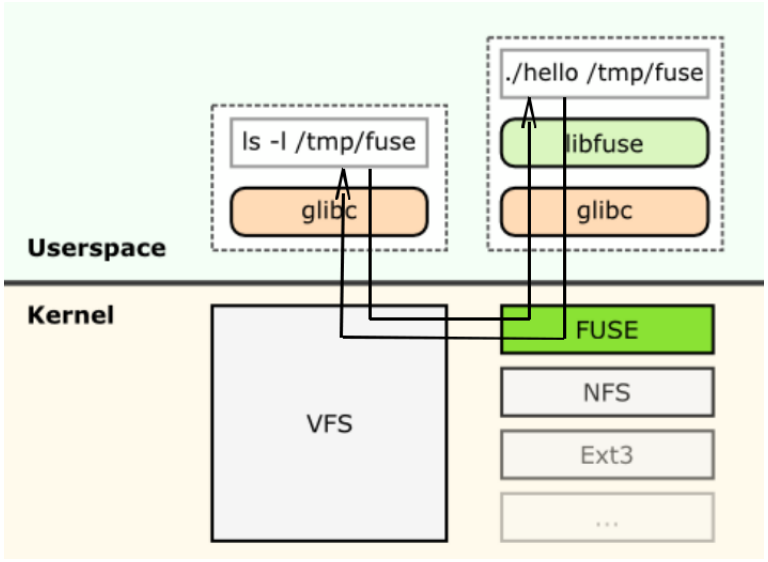


Figura 4.2: Visão de como os funcionam os *handlers* do FUSE

através do seu identificador do processo (*pid*). Esta funcionalidade vai permitir associar informação à aplicação/fluxo de execução.

O FUSE oferece duas formas de realizar a definição de um novo sistema de ficheiros, uma de alto nível e outra de baixo nível. Na versão de alto nível os *handlers* da implementação do sistema de ficheiros recebem a identificação dos ficheiros como sendo o seu caminho. Por oposição, a vertente de baixo nível, somente conhece a noção de identificador de *inode* (identificador único de um elemento no sistema de ficheiros).

Os *handlers* de alto nível constituem um nível de abstracção mais elevado e constitui um acréscimo de complexidade relativamente ao VFS. A API de baixo nível é muito similar a presente no VFS e, por isso, pode oferecer potencialmente melhor desempenho. Para além disto, lidar com identificadores em vez de caminhos facilita a gestão da informação. Por estas razões o TFSoF foi implementado utilizado a interface baixo nível do FUSE, facilitando ainda o desenvolvimento futuro de uma versão a integrar directamente no *kernel* do sistema operativo.

4.3 Componentes

Nesta secção vamos descrever os componentes que fazem parte do sistema implementado. Um esquema destes componentes é apresentada na Figura 4.3.

De seguida são descritas com maior pormenor as diversas operações para cada componente.

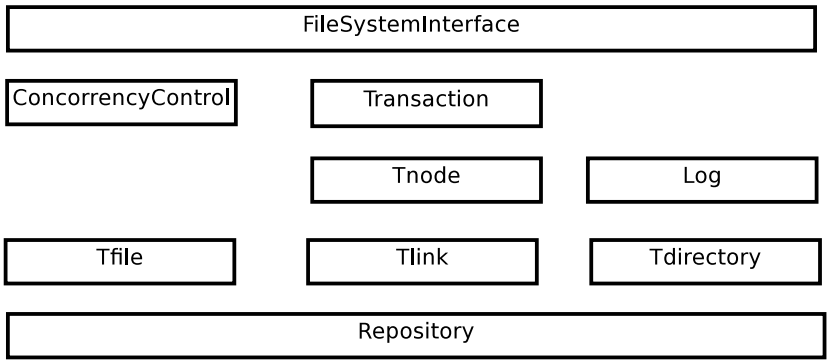


Figura 4.3: Esquema de componentes implementados

4.3.1 *File System Interface*

Como forma de permitir o suporte de acções sobre o repositório de dados final que não tenham cariz transaccional, este módulo mantém uma tabela de dispersão de identificadores de *inode*, que vamos referir como *inodepath*. Esta tabela guarda o caminho para o elemento dentro do sistema de ficheiros que implementa o repositório de dados.

4.3.2 *Handlers*

Para concretizar a implementação do nosso sistema foi necessário definir um conjunto de *handlers*. Estes *handlers* oferecem resposta aos serviços desencadeados pela invocação por parte das aplicações de funções de alto nível de manipulação de ficheiros. Seguidamente apresentamos para cada um desses *handlers* implementados uma descrição do seu comportamento.

Estas descrições referem-se somente às invocações onde está uma transacção activa, ou seja, a aplicação que realiza a invocação do serviço já iniciou uma transacção. No caso de não existir contexto activo a invocação é passada quase que directamente para o repositório de dados.

Lookup Dado o identificador da directoria pai e o nome da entrada, faz uma procura da directoria e devolve o identificador do *inode* que tenha nome igual. Este *handler* é invocado quando é necessário realizar a tradução de um caminho para um identificador de *inode*. A pesquisa pelo *inode* da directoria é realizada na transacção corrente.

A forma como se chega ao caminho no repositório é a seguinte: com o identificador do *inode* pai é pesquisado na *inodepath* e é devolvido o caminho para o pai. Com esse caminho é feita a concatenação com o nome da entrada procurada. Caso o caminho seja válido, este é adicionado a *inodepath* e

retornado o valor de identificação do *inode* no repositório. Caso contrario é devolvido um erro.

Por exemplo se uma aplicação abrir o ficheiro "/dir/a", a *handler lookup* vai ser invocada inicialmente com o par de parâmetros *parent*=0 e *name*="dir". É então pesquisado na *inodepath* pelo *inode* 0 o que devolve o caminho "/". Com nome "/dir" é obtido o identificador do *inode*, vamos supor que é 23, sendo este o valor que é devolvido ao FUSE. É também adicionado a *inodepath* a entrada "/dir" com chave 23. Em seguida o *handler* é novamente invocado com os paramentos *parent*=23 e *name*="a" e todo o processo repete-se até chegar ao último elemento do caminho.

Getattr Obtém os atributos de um *inode*, como seja as permissões do ficheiro. Os atributos são obtidos a partir do *inode* presente nessa transacção. Esta função devolve como resultado um estrutura *stat* padrão dos sistemas POSIX.

Setattr Altera os atributos de um ficheiro, como seja as permissões do ficheiro. Este *handler* recebe um estrutura *stat* e um máscara que especifica quais são os atributos a alterar. Os atributos são alterados no *inode* presente nessa transacção.

Open Abre um ficheiro para leitura e/ou escrita. Nesta *handler* é verificada existência de alguma transacção corrente, se não existir é então criada uma nova transacção. De seguida é invocado na transacção, o serviço para abrir um ficheiro. Este método devolve um identificador do objecto que representa o ficheiro aberto dentro da transacção.

Para cada ficheiro aberto o FUSE mantém internamente uma estrutura de dados onde são mantidas algumas informações. Esta estrutura é passada como parâmetro para os *handlers* de *read*, *write* e *release* que sejam invocados para o ficheiro aberto. Nesta estrutura encontra-se um campo que pode ser livremente afectado pela implementação, denominado por *file handler*. Então este *file handler* é utilizado pela nossa implementação para guardar o identificador do objecto que representa o ficheiro aberto dentro da transacção.

Read Lê um determinada quantidade de bytes de um ficheiro aberto. Este *handler* retira do *file handler*, que recebe como parâmetro, o identificador do ficheiro aberto. Esta leitura tem de reflectir todas as alterações feitas anteriormente pela aplicação no contexto da transacção corrente. Para tal esta operação é delegada ao objecto que representa a transacção, passando-lhe o identificado do ficheiro aberto.

Write Escreve uma determinada quantidade de bytes num ficheiro aberto. Tal

como no *handler* de *read*, esta função delega na entidade transacção corrente a tarefa de fazer a escrita de uma quantidade de bytes num ficheiro aberto.

Release Este *handler* é invocado quando ocorre o último *close* de um ficheiro aberto. Isto porque podem existir, em alto nível, mais do que um descritor referente a este mesmo ficheiro (tal pode ser conseguido com o utilização da função *dup2*). Neste *handler* é verificado o número de ficheiros abertos no contexto da transacção corrente e se este tiver o valor de 1 é então invocado o *commit* da transacção corrente.

Opendir Abertura uma directoria. Verifica se a directoria existe na transacção corrente e caso a directoria ainda não esteja presente na transacção é então carregada do repositório. É então invocada a função do objecto transacção que abre a directoria e que devolve um identificador à semelhança do que acontece no *handler open*. É também afectado o valor da *file handler* para ser utilizado nos *handler* futuros referentes a directoria agora aberta.

Readdir Leitura as entradas da directoria anteriormente aberta. É invocada a função específica da transacção, que devolve a lista de entradas na directoria existentes no contexto da transacção corrente.

Releasedir Invocado quando se pretende fechar uma directoria aberta. Este *handler* só é relevante para as directorias presentes no contexto de uma transacção, pois só neste caso é que é invocada a função especifica para fechar uma directoria aberta.

Readlink Dado um identificador de um link simbólico, devolve o caminho para o ficheiro apontado. O caminho de destino do link é retornado do contexto da transacção.

Mknod Criação de um novo ficheiro com o modo dado. O novo ficheiro é criado nesse contexto e é somente acessível através deste contexto. Neste ponto é criando um novo *Tfile* que descreve o novo ficheiro no repositório na directoria temporária.

Mkdir Criação de uma nova directoria com um determinado modo. A nova directoria é criando nesse e somente nesse contexto. Neste ponto é criando um novo *Tdirectory* que descreve a nova directoria.

Symlink Criação de um novo *link* simbólico. Este novo *link* é criado nesse contexto. Neste ponto é criando um novo *Tlink* que descreve o novo *link*.

Link Criação de um *hard link*. Este *handler* adiciona uma nova entrada à directoria pai da nova localização do ficheiro e incrementa o número de *links* no

inode do ficheiro. Isto será realizado na transacção corrente ou directamente do repositório de dados, caso a transacção não exista transacção corrente.

Unlink Remoção de um ficheiro ou *link*. Este *handler* remove a entrada presente a directoria pai da localização do ficheiro e decrementar o número de *links* no *inode* do ficheiro.

Rmdir Remoção de uma directoria. Este *handler* remove a entrada presente na directoria pai da localização dada, assumindo que esta localização corresponde a uma directoria vazia, caso que também é verificado.

Rename Remoção de uma entrada numa directoria e adiciona noutra. Este *handler* remove a entrada presente na directoria pai da localização inicial e adiciona na directoria pai da localização final um nova entrada.

Em todos os *handlers* anteriormente descritos são verificadas diversas situações de erro, como seja, a não existência da entrada, o identificador não corresponde a uma directoria ou a entrada já existente. Contudo foi necessário devolver um novo valor de erro, correspondente a situação em que na execução da transacção está num estado de *abort*, neste caso é devolvido um valor de erro correspondente ao *abort*.

Os novos ficheiros criados num sistema de ficheiros implicam novas entradas na respectivas directorias onde são criadas. No contexto de uma transacção é criado um novo ficheiro (*mknod, mkdir, symlink rmdir, rename*), tem de ser verificado se o respectivo *inode* da directoria pai já se encontra presente na transacção, senão o mesmo tem de ser carregado, para permitir adicionar a nova entrada.

4.3.3 *Transaction*

O módulo *transaction* gere uma transacção ao nível do sistema de ficheiros. Um objecto do tipo *transaction* está associado a uma aplicação e agrega um conjunto de objectos *tnode*. Os objectos *tnode* representam os *inodes* alterados no decorrer da transacção. Um *inode* guarda a informação de um ficheiro no sistema mas o conteúdo do ficheiro pode ser de vários tipos (ficheiro de dados, directoria e *link* simbólico). Este objecto mantém como informação a estrutura de dados descrita na Figura 4.4, cujos campos são posteriormente descritos na Tabela 4.1.

Cada objecto *transaction* tem um *log* de alterações realizadas em directorias. Este *log* é composto por entradas com o tipo da operação e com os respectivos valores de parâmetros. Este *log* é necessário pois as alterações de entradas de directorias são dependentes entre si e portanto é necessário serem aplicadas por uma determinada ordem.

O módulo *transaction* disponibiliza um conjunto de funções que aplicam internamente os serviços da interface do sistema de ficheiros (*lookup, open, read, write*, etc.). No

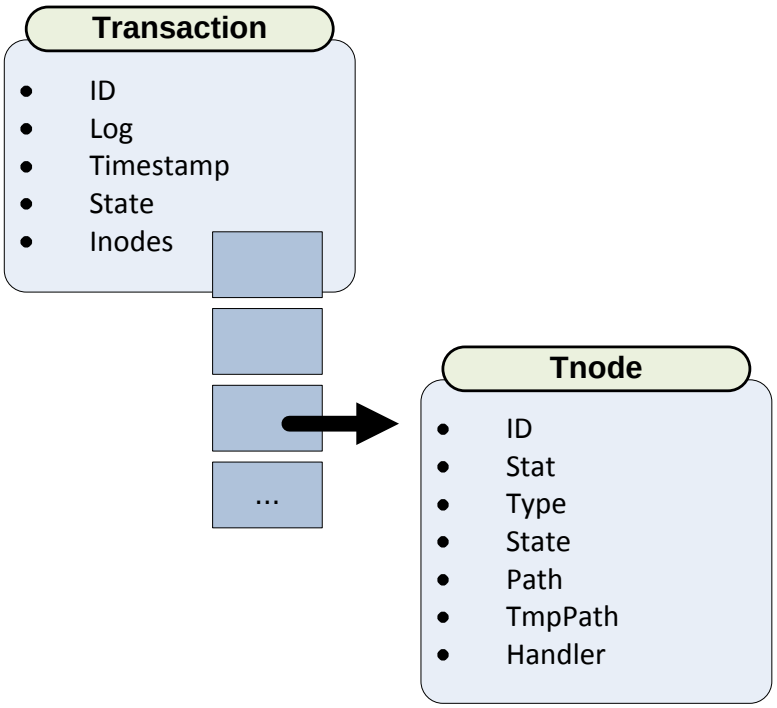


Figura 4.4: Informação mantida por uma transacção e por *tnode*

Campo	Descrição
id	identificador da transacção.
openFiles	número de ficheiros abertos no decorrer desta transacção.
inodes	*hashtable* que permite o acesso aos *inodes* manipulados no contexto desta transacção.
log	log onde são guardadas as informações sobre as operações realizadas sobre as entradas de directorias.
timestamp	estampilha temporal que foi atribuída a esta transacção. Informação utilizada na gestão de concorrência.
state	estado da transacção, se está activa ou se já esta definida como abortada.

Tabela 4.1: Campos do objecto *transaction*

entanto existem mais funções que são importantes de descrever.

start Inicializa um objecto *transaction* e todos os seus membros.

load_inode Dado um caminho para um ficheiro já existente no repositório, é carregado para o contexto da transacção toda a informação relativa a esse *inode*.

check_inode Verifica se determinado *inode* já está presente no contexto de uma transacção.

commit Valida todas as alterações presentes no contexto de uma transacção e aplica as alterações ao repositório. Esta aplicação de alterações consiste em percorrer todo o *log* e aplicar as alterações de forma sequencial. Seguidamente são percorridos todos os *tnodes* presentes na transacção e aplicadas as alterações de cada um.

transaction_abort Descarta todas as alterações efectuadas numa transacção e finaliza a transacção.

4.3.4 *Tnode*

O modulo *tnode* trata toda a informação relativa a um *inode*. Cada objecto deste tipo mantém um conjunto de atributos descritos na tabela 4.2.

Campo	Descrição
id	identificador
state	origem deste *tnode*, trata-se de um novo ou de uma cópia de um já existente
metadata	estrutura do tipo *stat* onde são guardados os valores dos atributos do *tnode*
destiny	caminho de destino onde o conteúdo do *tnode* vai pertencer, numa visão futura
temp	caminho do conteúdo numa localização temporária
handler	objecto que detalha e perversa a informação para cada tipo de dados

Tabela 4.2: Campos do objecto *tnode*

Este objecto mantém toda a informação que está presente no *inode* dum sistema que respeite a norma POSIX, como seja o modo de acesso, *group id*, *user id*, dimensão dos dados, entre outros presentes na estrutura de dados *stat*. Quando uma aplicação modifica um deste atributos no decorrer de uma transacção é neste objecto que são efectuadas as alterações.

A origem da informação de um objecto *tnode* pode ser uma de duas. Ou se trata de um novo elemento no sistema ou trata-se de um cópia de dados já existentes. Caso se trate de uma novo elemento é criando um novo ficheiro na directoria temporária do repositório com o fim de obter um novo identificador de *inode*.

Na solução desenvolvida estamos a fazer diferenciação entre três tipos diferentes de conteúdos. Cada objecto do tipo *tnode* tem uma ligação para um objecto que especifica e permite lidar com o tipo de conteúdo. Assim existem diferentes tipos de objectos, um para ficheiros, outra para directorias e outro para *links* simbólico. Este objectos são apresentados na Figura 4.5 e são depois descritos nas próximas secções.

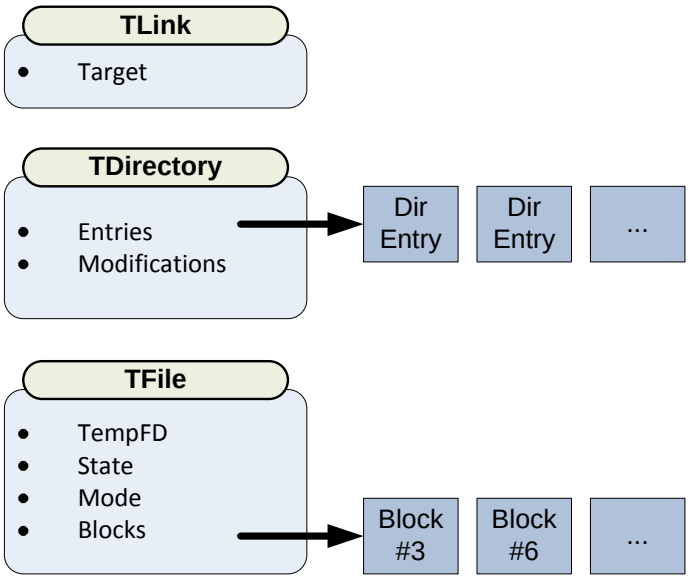


Figura 4.5: Informação mantida pelos diversos objectos presentes numa transacção

A aplicação das alterações das informações de um *tnode* são realizadas da seguinte forma. Primeiramente são aplicados os vários atributos através da invocação dos vários métodos do repositório que permitem a alteração de meta dados (*chowner*, *utime* e *truncate*). Seguidamente é chamada a função específica para cada tipo, que vai aplicar as alterações de conteúdo. A interface deste módulo é extremamente simples e resume-se aos seguintes elementos:

load carrega a partir de uma localização dada todas as informações do *tnode*.

setattr altera os atributos presentes no *tnode*.

getattr devolve os atributos actuais do *tnode*.

apply aplica os valores actuais do atributos ao elemento de destino.

Outro aspecto a considerar é a aplicação das alterações dependendo da origem do *tnode*, pois este pode dizer respeito a uma cópia privada das informações de um ficheiro já existente ou um novo ficheiro. Caso se trate de uma cópia, as alterações são

aplicadas directamente ao caminho de destino, mas se for um novo *tnode*, estão primeiramente o ficheiro é movido da sua localização temporária para o seu destino e depois aplicadas a modificações.

4.3.5 *Tfile*

O objecto *tfile* representa um ficheiro acedido no contexto de uma transacção. Este objecto fornece o acesso a todo o conteúdo do ficheiro aberto e permite uma visão privada deste. Os campos que estão presentes num objecto do tipo *tfile* são apresentados na tabela 4.3.

Campo	**Descrição**
filedesc	descritor do ficheiro original, caso este exista
blocks	objecto que onde são guardados os blocos alterados no contexto deste ficheiro aberto em especifico
state	estado o ficheiro, se está aberto ou fechado
mode	*mode* de abertura do ficheiro
tnode	apontador para o *tnode* com a meta informação relativa a este ficheiro

Tabela 4.3: Campos do objecto file

O objecto *tfile* mantém um sub-objecto, *fblock*, que é responsável pela gestão de todos os blocos alterados neste ficheiro aberto. Estes blocos privados à transacção são mantidos num ficheiro temporário. Este ficheiro temporário é composto por uma sequência de blocos de tamanho fixo, para o qual um objecto *fblock* mantém a informação de qual o *offset* onde está presente determinado bloco. Para tal, este *fblock* mantém uma lista com o *offset* dos blocos bem como o descritor do ficheiro temporário onde os blocos modificados estão guardados. A API do módulo *tfile* é também de fácil compreensão e é descrita da seguinte forma:

read lê uma quantidade de bytes no ficheiro aberto. O conteúdo lido pode ter como origem blocos privados e/ou blocos originais.

write escreve uma quantidade de bytes no ficheiro aberto. A escrita vai ser realizada em blocos privados a transacção corrente.

apply aplica as alterações efectuadas no ficheiro aberto. Ou seja, percorre todos os blocos considerados privados escrevendo o seu conteúdo sobre o conteúdo original no bloco respectivo.

No acto de escrita e leitura é sempre verificado se para o bloco referente já existe uma cópia privada. Se já existir uma cópia privada, este bloco é lido para memória

do ficheiro temporário, e é utilizado para a operação de leitura ou escrita. Se não existir uma cópia privada, então a leitura do bloco é realizada no ficheiro original. No caso de não existir um cópia e a operação ser de escrita, então o bloco é lido do ficheiro original, a operação é realizada em memória e de seguida o bloco é colocado no ficheiro temporário.

4.3.6 Tdirectory

O módulo *tdirectory* oferece o suporte para as operações sobre directorias. Uma directoria mantém uma lista de entradas, compostas por pares identificador de *tnode* e um nome. Como tal é mantido uma lista composta por entradas do tipo (nome, identificador de *tnode*). Este modulo permite a adição e remoção de entradas, bem como a enumeração das entradas presentes na directoria. Um objecto deste tipo mantém os campos descritos na Tabela 4.4.

Campo	**Descrição**
entries	lista das entradas da directoria
alterations	lista dos nomes das entradas alteradas desde da altura da criação deste objecto directoria
tnode	apontador para o *tnode* que contem a meta informação desta directoria

Tabela 4.4: Campos do objecto *directory*

Este módulo disponibiliza métodos para adicionar uma entrada, remover uma entrada, devolver a lista de entradas e para resolver um nome. A resolução de um nome consiste em pesquisar de forma linear na lista de entradas e retornar o respectivo identificador de *tnode*.

Como forma de verificar a possibilidade de conflitos, este objecto mantém uma lista dos nomes alterados durante o seu tempo de vida. Estas lista de alterações consiste no conjunto de nomes que não existiam ou que deixaram de existir desde o momento em que o objecto foi criado.

De forma a obter um novo identificador único do *tnode* para uma directoria, é criada na directoria temporária, um nova directoria com um nome gerado de forma única. Como tal a aplicação das alterações realizadas implementada por este objecto limita-se mover a directoria para a localização final, visto que todos os outros casos são tratados pelo *log* presente na transacção que regista a criação de directorias.

4.3.7 *Tlink*

O módulo responsável pelo tratamento de *links* é o *tlink*. Um objecto *tlink* representa um *link* simbólico. Este guarda a localização do destino do *link*. Este tipo de objecto é muito simples tal como os seus elementos, que estão descritos na Tabela 4.5.

Campo	Descrição
target	caminho para o destino apontado por este link simbólico
tnode	apontador para o *tnode* que contem a meta informação deste *link* simbólico

Tabela 4.5: Campos do objecto *tlink*

Como já foi referido, um *link* simbólico é imutável, como tal quando é criando um novo objecto que representa este tipo é também criando, na directoria temporária, um *link* simbólico com estas mesma características, com a finalidade de receber um novo identificador único. No momento de aplicação das alterações é realizado um mover deste link da directoria temporária para a sua localização final.

4.3.8 *ConcorrencyControl*

Este módulo implementa os mecanismo de controlo de concorrência que permitem a detecção de conflitos entre transacções. Este módulo é responsável por gerir as estampilhas temporais para os ficheiros dos sistema. É realizada a distinção entre dois tipos de ficheiros, entre ficheiros de dados e directorias. São mantidos dois mecanismos principais, uma tabela de dispersão para blocos de ficheiros e outra tabela de dispersão para entradas de directorias.

A primeira, relativa a ficheiros, é uma tabela dispersão com dimensão fixa previamente definida. Utilizando um conjunto de funções de gestão permite-se, com base no identificador único do *tnode* e no respectivo índice do bloco, aceder ao valor da estampilha para este par (ficheiro, bloco).

Relativamente às directorias existe outra tabela com o mesmo principio de funcionamento. Neste caso, o par de acesso é (identificador da directoria, nome entrada), o que permite determinar a existência de conflitos nas alterações de nomes de uma directoria.

Em relação ao *links* simbólicos não é necessária nenhuma medida de controlo de conflito, pois estes são elementos imutáveis num sistema de ficheiros, ou seja, não é possível alterar um link simbólico, é sempre necessário remove-lo e criar um novo.

Seguidamente é feita uma breve descrição dos serviços disponibilizados por este módulo.

update_file_timestamp dado um identificador de *tnode* e um índice do bloco, actualiza o valor da estampilha.

update_entry_timestamp dado um identificador de *tnode* respeitante a uma directoria e um nome da entrada, actualiza o valor da estampilha.

get_file_timestamp dado um identificador de *tnode* e um índice do bloco, retorna o valor da estampilha.

get_entry_timestamp dado um identificador de *tnode* respeitante a uma directoria e um nome da entrada, retorna o valor da estampilha.

Os valores de estampilhas aqui mantidos são tabelas de dispersão de tamanho fixo. Este facto vai permitir a ocorrência de acesso a valores de estapilha errados, devido a sobreposição gerada pela função de dispersão. Isto quer dizer que para uma chave diferente vai ser acedido um mesmo valor de estampilha, esta situação vai gerar situações de falso conflitos entre transacções.

4.3.9 *Repository*

O módulo *Repository* implementa o componente onde são preservadas as versões finais dos ficheiros. Este módulo recorre a uma directoria no sistema de ficheiros base da maquina para realizar a gravação dos ficheiros.

Disponibiliza um conjunto de serviços similares aos já disponibilizados por um sistema de ficheiros, contudo acresce ainda a gestão especial de ficheiros temporários. Este gere de forma diferenciada os ficheiros que ficam na directoria temporária e os localizados na directoria base.

5

Avaliação do TFSoF

Depois de no capitulo anterior ter sido descrita a implementação da solução, vamos agora neste capitulo vamos avaliar a implementação realizada. Inicialmente iremos testar o correcto funcionamento do TFSoF, validando as suas características, seguindo-se depois uma segunda etapa de avaliação de desempenho. O TFSoF foi testado num sistema computacional com as características descritas na Tabela 5.1.

Maquina de Testes	
Designação	Sun Fire X4600
Sistema operativo	Debian 5.0.3
Kernel	Linux 2.6.26.2-amd64 SMP
FUSE	2.7.4
CPU	4 x Dual-Core AMD Opteron 8220 @ 2.86 Ghz
Familia CPU	x86-64
L1 Cache	64 KB / Core
L2 Cache	1024 KB
Memoria	32 GB
Disco	LSI Logic Volume 136 GB
Sistema de ficheiros	93 GB @ Ext3

Tabela 5.1: Características técnicas na maquina onde foram realizados os testes

5.1 Validação funcional

Neste ponto vamos avaliar o correcto comportamento do protótipo desenvolvido. A descrição de cada teste cobre quatro aspectos: os objectivos do teste, a metodologia seguida para a realização do mesmo, os resultados obtidos e a analise desses resultados.

Verificação das operações de escrita e leitura sobre ficheiro

Esta verificação tem como objectivo comprovar que o conteúdo de um ficheiro se mantém consistente após uma sequência de operações de leitura e escrita.

Neste teste foi efectuada um cópia de um ficheiro de texto, com 35 KiB, do sistema de ficheiros da máquina para o ponto de montagem do TFSoF. De seguida foi feita a comparação entre o ficheiro original e o cópia, recorrendo ao utilitário *diff*. Esta comparação não reportou qualquer diferença entre os dois ficheiros. Posteriormente, o ficheiro de texto presente no TFSoF foi alterado de forma a substituir todas as ocorrências da letra 'A' por '#', seguida de nova comparação, que agora reportou todas as diferenças.

Numa segunda fase foi realizada a cópia de um ficheiro comprimido com 92 MiB para o directoria de montagem do TFSoF. Depois fui computado o *checksum md5* deste ficheiro, tanto na localização original, como o directoria de montagem do TFSoF. Como resultado obtivemos o mesmo valor para as duas localizações.

Estes resultados permitem comprovar que a manipulação do conteúdo dos ficheiros está a ser realizada de forma correcta.

Verificação dos atributos de um ficheiro

Esta verificação tem como objectivo comprovar que a alteração dos atributos de um ficheiro estão a ser implementadas de forma correcta.

Primeiramente foram alterados os atributos de *id* do dono, *id* do grupo e comprovado que essas alterações se apresentavam no sistema de ficheiros (através de uma listagem da directoria). Depois disso foram alteradas as permissões, comprovado que estas apareciam modificadas na listagem da directoria. Para além disso foi comprovado que com as permissões definidas de forma a impedir o acesso ao ficheiro, a operação de acesso a este devolvia o erro correspondente.

Numa segunda fase foi alterado o tamanho de um ficheiro, com 35 KiB, para uma dimensão de 20 KiB. Esta alteração foi comprovada tanto, pela listagem da directoria, como pela leitura do ficheiro.

Estes resultados permitem comprovar que a manipulação dos atributos dos ficheiros está a ser realizada de forma correcta.

Verificação do isolamento em ficheiros

Esta verificação tem como objectivo comprovar que a cada processo tem a sua versão do conteúdo de um ficheiro que seja por si manipulado.

Neste teste foi desenvolvido um programa que acedia a um ficheiro predeterminado e alterava um byte numa posição aleatória. Esta alteração era também feita para um valor aleatório. Este programa realizava a leitura dessa posição, com a finalidade de comprovar o valor que tinha escrito. Esta alteração era realizada de forma cíclica um elevado número de vezes. Caso o valor lido fosse diferente do valor escrito a aplicação terminava e reportava o erro.

Foram postas em execução várias instâncias concorrentes deste programa e em nenhuma delas reportou a ocorrência de erro. Isto permite chegar a conclusão de que cada instância tinha a sua própria versão do conteúdo do ficheiro a que acediam.

Verificação de alterações atómicas

Esta verificação tem como objectivo comprovar que as alterações realizadas por um processo ao conteúdo de um ficheiro são aplicadas de forma atómica.

Foi desenvolvido um programa que abria um determinado ficheiro, realizava uma modificação a todo o seu conteúdo e depois esperava por input do utilizador para fechar o ficheiro. Mais especificamente, o programa vai alterar o conteúdo de um ficheiro repleto com 1000 caracteres 'A' por caracteres 'B' e esperar pela ordem do utilizador para fechar o ficheiro. Assim o programa foi posto em execução, verificado o conteúdo do ficheiro (com o utilitário *cat*) e comprovada que apenas continha 'A's. De seguida foi dado input à aplicação e verificado novamente o conteúdo do ficheiro, que estava agora continha somente 'B's.

O mesmo procedimento foi novamente realizado, mas agora para 5 ficheiros idênticos, em que o programa fechava todos os ficheiros excepto um e só depois do input do utilizador fechava o último. Verificou-se que os conteúdos dos 5 ficheiros só eram alterados depois de dado o input à aplicação.

Foi também desenvolvido outro programa que lança 10 *threads*, em que cada uma vai realizar 100 vezes o seguinte procedimento: abre 5 ficheiros pre-determinados de forma sequencial, escreve no início dos ficheiros o seu identificador e fecha os 5 ficheiros. Depois de executado este programa verificou-se que o conteúdo dos 5 ficheiros era idêntico.

Os resultados observados permitem comprovar que a alteração do conteúdo dos ficheiros é realizado somente depois do último ficheiro ser fechado (final da transacção).

Verificação de alteração nas directorias

Esta verificação tem como objectivo comprovar que as alterações realizadas nas entradas de uma directoria são realizadas de forma correcta.

Primeiramente foi extraído o conteúdo de um arquivo comprimido do código fonte do TFSoF (que contém alguns ficheiros organizados numa estrutura de directorias) para o ponto de montagem do TFSoF. Esta operação decorreu sem erros, sendo depois a directoria criada comparada com a directoria original, utilizando o utilitário *diff*. Depois disso foi removida a directoria criada e todo o seu conteúdo (de forma recursiva), operação que também decorreu sem erros.

Seguidamente foi desenvolvido um programa que realizava as seguintes operações numa dada directoria: escolhe se vai eliminar ou adicionar uma entrada na directoria; caso vá eliminar, lista a directoria, escolhe uma entrada, elimina essa entrada e no final lista novamente a directoria para comprovar que a entrada não está presente; caso vá criar uma nova entrada, gera um nome aleatória e realiza um *hard link* para um ficheiro pré-determinado, de seguida lista a directoria e comprava que a nova entrada está presente nesta listagem. Estas operações estão delimitadas por uma abertura e um fecho de um ficheiro para serem vistas como presentes numa transacção. A execução deste programa não reportou qualquer erro.

Os resultados observados permitem comprovar que a manipulação das entradas numa directoria está a ser realizada correctamente.

5.2 Validação de desempenho

Neste secção vai ser descrita a avaliação de desempenho realizada ao sistema desenvolvido. Para tal recorreu-se a uma ferramenta de *benchmark* de sistema de ficheiros intitulada IOzone [IOz09]. Este *benchmark* mede a capacidade do um sistema de ficheiros em termos do débito que consegue alcançar num determinada conjunto de testes.

Nestes testes faz-se variar a dimensão do ficheiro, bem como o dimensão do *buffer* onde são guardados os dados ao nível da aplicação. Os resultados apresentados de seguida tiveram como parâmetros os seguintes valores:

- Dimensão da *buffer* da aplicação variável entre 4 *KBytes* e 16 *MBytes*
- Dimensão do ficheiro variável entre 64 *KBytes* e 52 *MBytes*

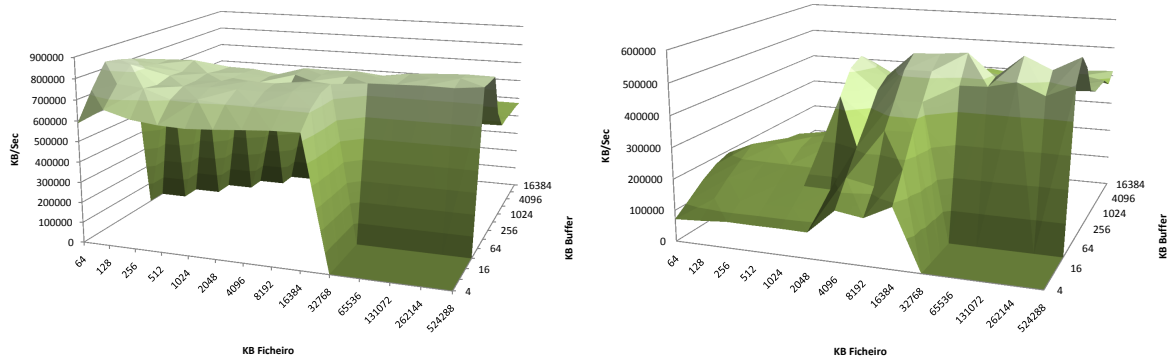
O *benckmark* IOzone foi executado e foram obtidos valores para os seguintes sistemas de ficheiros:

- Transacional File Sistem over FUSE (TFSoF);
- Sistema de ficheiros de base da máquina, o Ext3;
- Sistema de ficheiros implementado com o recuso ao FUSE, que simplesmente faz passar todas as operações para o sistema base da maquina, sendo assim intitulado de FUSE-Nulo;
- Transacional File Sistem (TFS);

De referir que os resultados apresentados em forma de gráfico apresentam variação na escala vertical correspondente ao débito, e também que os resultados conseguidos relativamente ao TFS foram limitados a certa certos valores da dimensão do *buffer*, bem como da dimensão do ficheiro. O TSF não suporta ficheiros grandes, estando o seu limite próximo dos 64 KiB, valor onde começaram os demais testes. Embora tenham sido obtidos com o IOzone os resultados para todos os testes, de seguida vamos só discutir os de leitura e escrita.

Testes de leitura

Os gráficos na Figura 5.1 apresentam os resultados obtidos no teste de leitura. De referir que o sistema base da máquina (Ext3) apresenta grande debito para dimensões pequenas tanto de *buffer* como de ficheiro, bem como apresenta em média um desempenho superior a todos os outros. Os sistemas de ficheiros implementados com recurso



(a) Ext3

(b) FUSE-Nulo

(c) TFS

(d) TFSoF

Figura 5.1: Resultados do teste de leitura sobre os quatro sistema de ficheiros

ao FUSE apresentam o mesmo padrão de resultados, baixo desempenho para baixos valores de *buffer* e ficheiros, mas para os restantes valores aproxima-se do Ext3.

Comparando o desempenho do TFSoF relativamente ao sistema de ficheiros nativo, para o qual foi construído um gráfico apresentado na Figura 5.2. É possível notar-se que existe uma perda de desempenho significativa para valores baixos das dimensões, tanto de ficheiro como do *buffer*, mas que para valores mais elevados deste mesmos parâmetros o desempenho melhora significativamente. Em média o desempenho do TFSoF é 47% quando comparado com o Ext3.

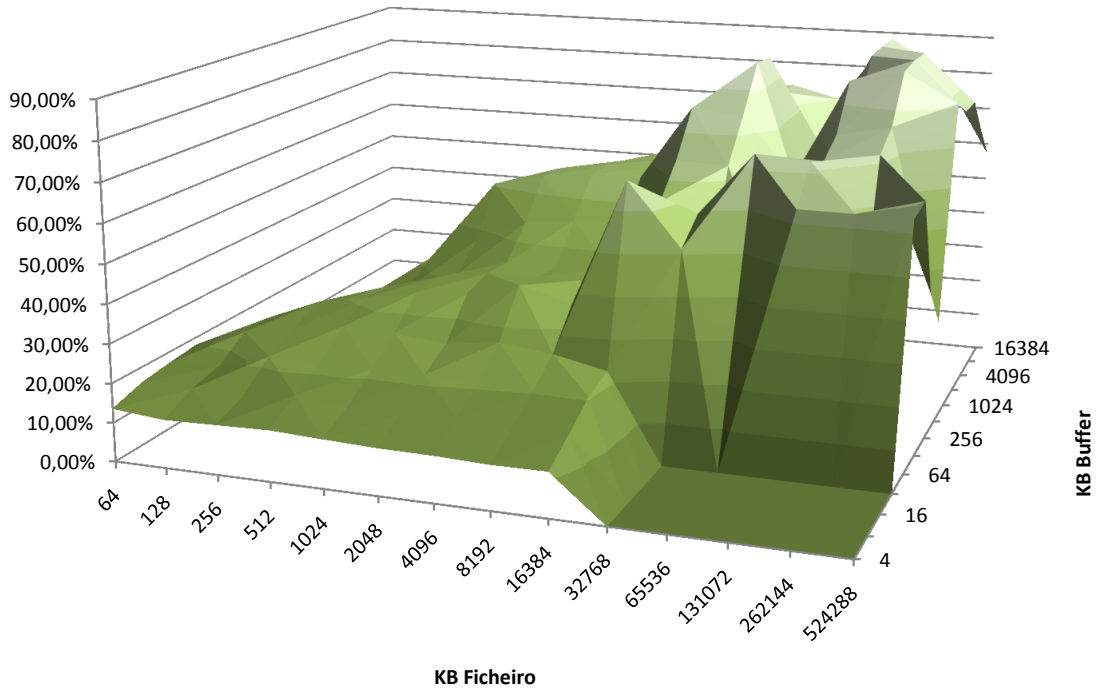


Figura 5.2: Comparação entre os resultados do teste de leitura para o TFSoF e o Ext3

Para entender este comportamento é necessário perceber qual o custo nas operações de leitura que estão ligadas a utilização do FUSE. Para responder a essa questão é apresentado um gráfico na Figura 5.3, onde se pode ver que existe um similaridade de comportamento e performance entre o FUSE-Nulo e o TFSoF. A média do desempenho obtido nesta comparação é de 57%, quando comparamos o FUSE-Nulo com o Ext3.

Relativamente ao TFS quando comparado com o TFSoF este apresenta alguns valores de débito mais elevados (de notar que a escala horizontal (KB ficheiro) é diferente dos demais). Isto muito provavelmente deve-se ao facto de a gestão dos blocos no TFS ser realizada em memória e portanto é espectável um melhor desempenho.

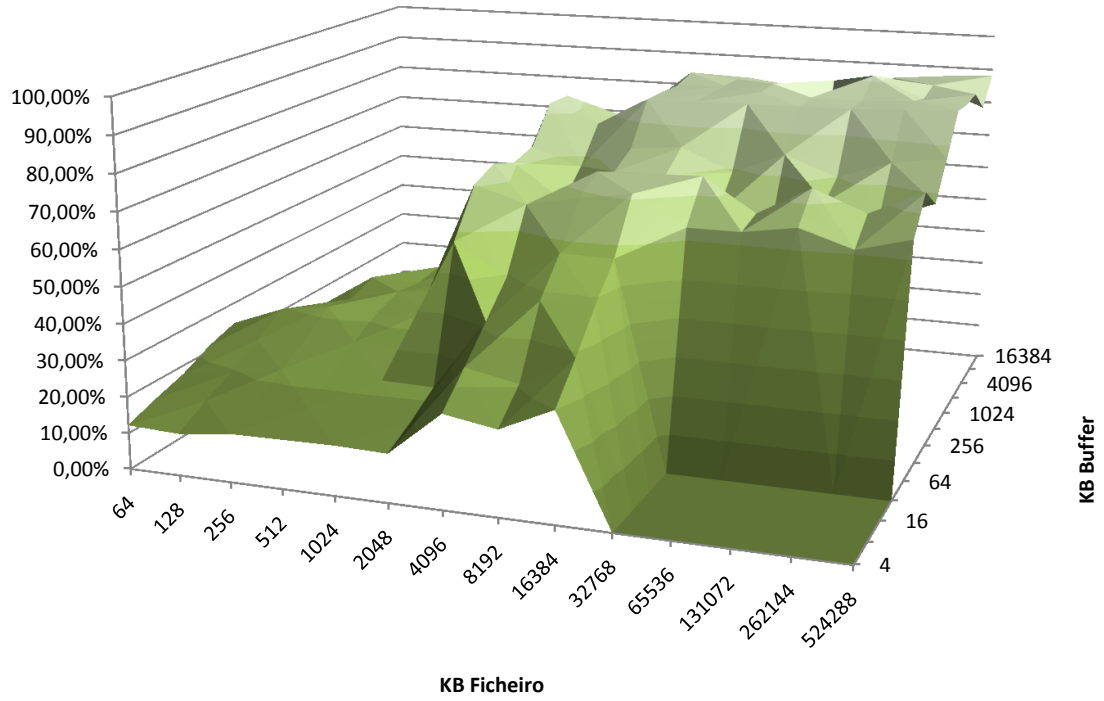


Figura 5.3: Comparação entre os resultados do teste de leitura para o FUSE-Nulo e o Ext3

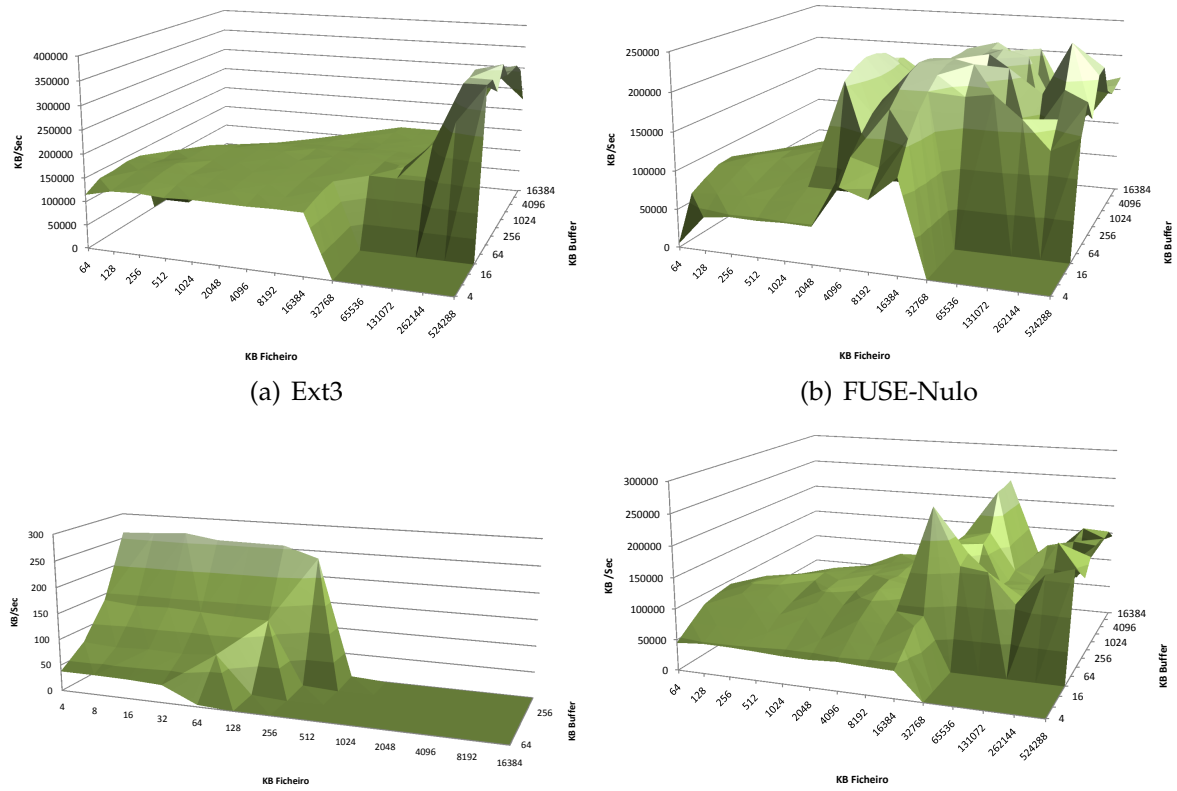
Testes de escrita

Os gráficos na Figura 5.4 apresentam os resultados obtidos no teste de escrita. De referir que o sistema base da máquina (Ext3) resultados regulares para quase maioria dos valores de parâmetros, tirando que para valores elevados obtém-se um pico de debito. Os sistema de ficheiros implementados sobre o FUSE apresentam um comportamento irregular, sendo que chegam a apresentar melhor desempenho que o Ext3.

Comparando o desempenho do TFSoF relativamente ao sistema de ficheiros nativo. Para isso foi construído um gráfico apresentado na Figura 5.5, onde se nota que existe uma perda de desempenho para valores baixos das dimensões, tanto de ficheiro como do *buffer*, mas que para valores mais elevados deste mesmos parâmetros o desempenho melhora significativamente, sendo até superior ao do Ext3 em alguns casos. Em média o desempenho do TFSoF é 72% quando comparado com o Ext3.

Mais uma vez, é necessário perceber qual o custo no desempenho que se deve a utilização do FUSE. Para responder a essa questão é apresentado um gráfico na Figura 5.6, onde se pode observar que existem valores para os quais o FUSE-Nulo obtém melhor resposta que o Ext3. A média do desempenho obtido nesta comparação é de 94%, quando comparamos o FUSE-Nulo com o Ext3, nas operações de escrita.

Relativamente ao TFS quando comparado com o TFSoF neste teste, o TFSoF apresenta resultados muito superiores ao TFS.



(a) Ext3 (b) FUSE-Nulo

(c) TFS (d) TFSoF

Figura 5.4: Resultados do teste de escrita sobre os quatro sistema de ficheiros

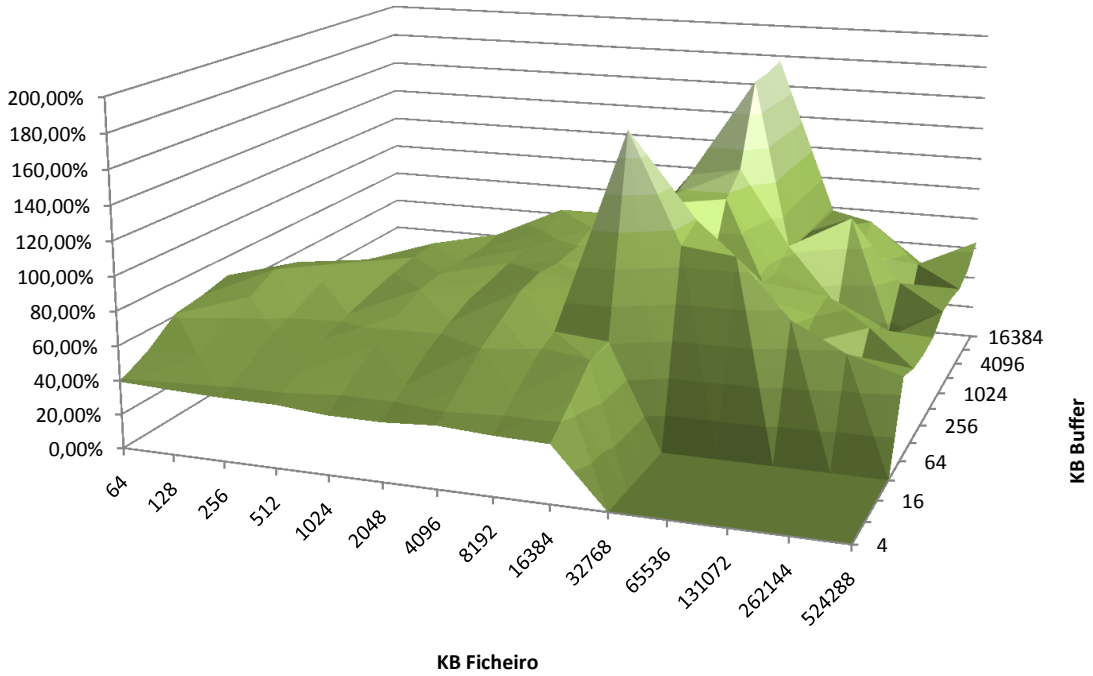


Figura 5.5: Comparação entre os resultados do teste de escrita para o TFSoF e o Ext3

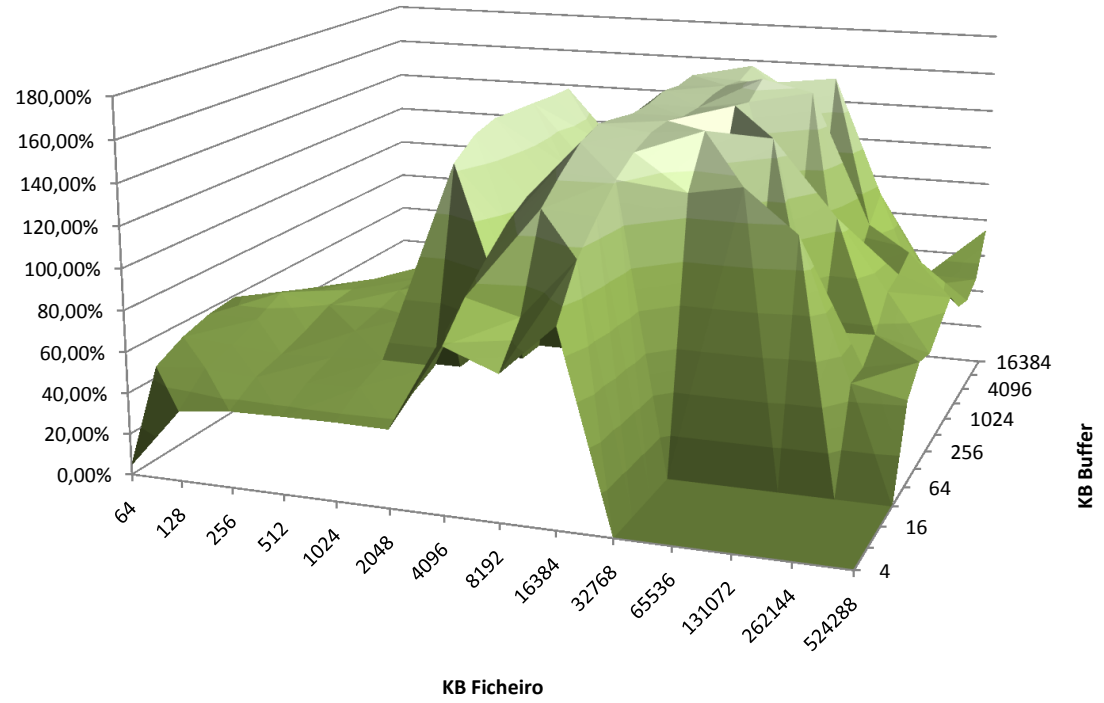


Figura 5.6: Comparação entre os resultados do teste de escrita para o FUSE-Nulo e o Ext3

Discussão

Na avaliação de desempenho de leituras o TFSoF apresenta um desempenho médio de 47% quando comparado com o Ext3, este inclui toda gestão de contexto transaccional, bem como a utilização do FUSE, que já por si só leva a um perda de desempenho. Quanto ao TFS, este apresenta em alguns casos melhores resultados que o TFSoF, embora devido as limitações do TSF, apenas haja resultados para os valores mais baixos dos parâmetros. Este melhor desempenho deve-se ao facto de o TFS gerir os blocos dos ficheiros em memória e como tal conseguir melhores tempos de acesso.

Quanto à avaliação de desempenho em escrita, o TFSoF apresenta um desempenho médio de 72% quando comparado com o Ext3, contudo o *overhead* médio do FUSE neste caso é de 6%, sendo algumas vezes superior ao Ext3. Este facto provavelmente deve-se a algum tipo de optimização que o FUSE possa realizar internamente em relação aos blocos transferidos. Quanto ao TFS, este apresenta resultados significante baixos, não sendo imediato qualquer motivo que possa conduzir a este comportamento.

6

Conclusões

Neste capitulo são apresentadas, por um lado, as considerações finais sobre o trabalho realizado e os seus respectivos resultados, e por outro, o possível trabalho futuro que possa surgir ou constituir um melhoria a esta dissertação.

6.1 Considerações finais

O principal objectivo desta dissertação foi criar um sistema de ficheiros com suporte para a noção de transacção e desta forma transportar a responsabilidade da gestão de acessos concorrentes das aplicações para o sistema de ficheiros.

Actualmente existe cada vez mais a necessidade de lidar com a concorrência a que as aplicações estão sujeitas. Tal facto leva a uma maior necessidade de utilização de mecanismos que permitem lidar com os possíveis problemas de concorrência, contudo a utilização deste mecanismos nem sempre é fácil. Neste sentido a utilização de transacções surge como um modelo que simplifica a tarefa com a concorrência.

Foram estudados os conceitos ligados as transacções, bem como as noções de sistemas de ficheiros. Foi também realizada uma recolha de sistemas de ficheiros que de alguma forma suportam a noção de transacção.

Foi realizado o desenho de um sistema de ficheiros que oferece a noção de transacção às aplicações, permitindo assim que estas possam beneficiar das propriedades ACID no acesso ao sistema de ficheiros. A sua utilização pode ser realizada de duas formas distintas: de forma explicita, onde a aplicação define directamente os limites do contexto transaccional; ou de forma implícita, onde é o sistema a determinar, com

base nas chamadas da aplicação, quais os limites da transacção.

Com base no desenho alcançado foi implementado um protótipo do sistema, apelidado de TFSoF, que recorrendo a utilização do *framework* FUSE, realiza a implementação de um novo sistema de ficheiros. Desta forma permite-se que as aplicações utilizem a noção de transacção, sem que para isso tenham de ser alterados os serviços padrão do sistema operativo. Este facto permite que a adopção do sistema seja mais fácil e rápida.

Os resultados obtidos na validação, embora demonstrem um fraco desempenho, permitem demonstrar o correcto funcionamento do sistema desenvolvido, permitindo assim afirmar que a solução proposta nesta dissertação é validada.

6.2 Trabalho futuro

Embora os objectivos tenham sido, a nosso ver, atingidos nessa dissertação, temos consciência de que certos aspectos ainda podem ser melhorados. Nesta secção vamos apresentar alguns pontos de interesse que podem ser futuros desenvolvimentos para o trabalho até aqui realizado.

Melhorar o desempenho

Uma das questões que ressalta a vista é o problema do baixo desempenho das operações realizadas. Este é um dos aspectos que tem necessidade de ser melhorado. A forma de o conseguir é fazer uma melhor gestão das cópias privadas de blocos, pois este é um dos pontos e onde é mais utilizado onde são utilizadas mais operações de escrita. Um das soluções possíveis será realizar a alocação antecipada de espaço em disco onde são mantidas essas cópias, outra das outras possíveis melhorias será ainda a melhor gestão dos índices mantidos em memória para gestão desses mesmo blocos.

Suporte para transacções explicitas

Uma das lacunas na implementação realizada prende-se com o facto de esta não suportar a delimitação de transacções de forma explicita. Embora tenha sido considerado como uma das opções de delimitação, esta não foi implementada.

Para realizar a implementação de tal funcionalidade seria necessário uma forma de comunicação directa entre a aplicação e a implementação do sistema de ficheiros. Uma das ideias consideradas foi a chamada de um função, por exemplo um *open*, sobre um ficheiro especial, ou com parâmetros especiais de forma a delimitar o inicio ou fim das operações transaccionais. Tal não consegue ser feito de forma trivial, pois a partida, a aplicação não sabe qual é o ponto de montagem da sistema. Para resolver essa questão

podia-se criar um ficheiro com localização global predefinida onde se encontrava a localização do ponto de montagem do sistema de ficheiros transaccional.

Outra ideia considerada foi a de implementar no sistema uma porta ou *device* que recebia os pedidos de início e fim, o que implicava o fornecimento de uma biblioteca à aplicação que implementasse a parte cliente deste serviço.

Suporte para sub-transacções

No ponto actual o sistema só suporta a noção de sub-transacções abertas, em que não existe isolamento entre as diversas sub-transacções. Uma das formas de enriquecer a solução proposta seria implementar outro modelos e com isso oferecer mais possibilidades às aplicações.

Bibliografia

[AMS+07] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 159–174, New York, NY, USA, 2007. ACM.

[Cun07] Gonçalo Cunha. Consistent state software transactional memory. Master's thesis, Faculdade de Ciência e Tecnologia - Universidade Nova de Lisboa, 2007.

[Dia08] Ricardo Dias. Cooperative memory and database transactions. Master's thesis, Faculdade de Ciência e Tecnologia - Universidade Nova de Lisboa, 2008.

[DLC08] Ricardo Dias, J. M. S. Lourenço, and G. Cunha. Developing libraries using software transactional memory. *Computer Science and Information Systems*, 5(2):103–117, 12 2008.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[GFG98] João Garcia, Paulo Ferreira, and Paulo Guedes. The PerDiS FS: a transactional file system for a distributed persistent store. In *EW 8: Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pages 189–194, New York, NY, USA, 1998. ACM.

[GMSP00] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Trans. Comput. Syst.*, 18(2):127–153, 2000.

[Gra81] Jim Gray. The transaction concept: virtues and limitations (invited paper). In *VLDB '1981: Proceedings of the seventh international conference on Very Large Data Bases*, pages 144–154. VLDB Endowment, 1981.

[GT05] Eran Gal and Sivan Toledo. A transactional flash file system for microcontrollers. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 7–7, Berkeley, CA, USA, 2005. USENIX Association.

[GTK09] GTK+ Team. The GTK+ Project, July 2009. `http://www.gtk.org/`.

[HM93] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[HR83] Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Comput. Surv.*, 15(4):287–317, 1983.

[HS08] Ralf Hoffmann and Miklos Szeredi. AVFS: A virtual filesystem, December 2008. `http://avf.sourceforge.net/`.

[HSP+08] Csaba Henk, Miklos Szeredi, Dobrica Pavlinusic, Richard Dawe, and Sebastien Delafond. Filesystem in userspace (FUSE), December 2008. `http://fuse.sourceforge.net/`.

[IBM09] IBM. Anatomy of the linux file system, January 2009. `http://www.ibm.com/developerworks/linux/library/l-linux-filesystem/`.

[IOz09] IOzone. Iozone file system benchmark, May 2009. `http://www.iozone.org/`.

[KCH+09] Sanjeev Kumar, Michael Chu, Christopher Hughes, Partha Kundu, and Anthony Nguyen. Hybrid Transactional Memory - Presentation, January 2009. `http://cobweb.ecn.purdue.edu/~smidkiff/ppopp/presentations/kumar_chu.ppt`.

[KM86] S. R. Kleiman and Sun Microsystems. Vnodes: An architecture for multiple file system types. pages 238–247, 1986.

[Lom77] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Proceedings of an ACM conference on Language design for reliable software*, pages 128–137, 1977.

[Mar08] Artur Martins. Transactional file system. Master's thesis, Faculdade de Ciência e Tecnologia - Universidade Nova de Lisboa, 2008.

[Mic08] Microsoft. Microsoft SQL Server, December 2008. `http://www.microsoft.com/sqlserver`.

[Mic09] Microsoft. About Transactional NTFS, January 2009. `http://msdn.microsoft.com/en-us/library/aa363764(VS.85).aspx`.

[MJLF84] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, 1984.

[MMW07] Paul E. McKenney, Maged M. Michael, and Jonathan Walpole. Why the grass may not be greener on the other side: a comparison of locking vs. transactional memory. In *PLOS '07: Proceedings of the 4th workshop on Programming languages and operating systems*, pages 1–5, New York, NY, USA, 2007. ACM.

[MSD08] MSDN: Microsoft Developer Network. Transaction-Safe FAT File System, December 2008. `http://msdn.microsoft.com/en-us/library/aa911939.aspx`.

[MTV02] Nick Murphy, Mark Tonkelowitz, and Mike Vernal. The design and implementation of the database file system, 2002.

[NTF09] NTFS-3G Technology. NTFS-3G Stable Read/Write Driver, January 2009. `http://www.ntfs-3g.com/`.

[OBS99] Michael A. Olson, Keith Bostic, and Margo I. Seltzer. Berkeley db. In *USENIX Annual Technical Conference, FREENIX Track*, pages 183–191. USENIX, 1999.

[Ols93] Michael A. Olson. The design and implementation of the inversion file system. In *Proceedings of the Usenix Winter 1993 Technical Conference*, pages 205–217, 1993.

[Ora08] Oracle Corporation. Oracle Database, December 2008. `http://www.oracle.com/database/index.html`.

[POS92] IEEE standards interpretations for IEEE standard portable operating system interface for computer environments (IEEE std 1003.1-1988). *IEEE Std 1003.1-1988/INT, 1992 Edition*, pages –, July 1992.

[Pos08] PostgreSQL Global Development Group. PostgreSQL, December 2008. http://www.postgresql.org/.

[Res08] Intel Research. Teraflops research chip, December 2008. http://techresearch.intel.com/articles/Tera-Scale/1449.htm.

[RG02] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill Science/Engineering/Math, August 2002.

[Ric09] Ricardo Correia. ZFS Filesystem for FUSE/Linux, January 2009. http://www.wizy.org/wiki/ZFS_on_FUSE.

[SATH+07] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Vijay Menon, Tatiana Shpeisman, Mohan Rajagopalan, Anwar Ghuloum, Eric Sprangle, Anwar Rohillah, and Doug Carmean. Runtime environment for tera-scale platforms. *Intel Technology Journal*, 11(Q2), 2007.

[SC98] David A. Solomon and Helen Custer. *Inside Windows NT*. Microsoft Press, Redmond, WA, USA, 1998.

[SGG04] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, Dec 2004.

[SKS05] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill Science/Engineering/Math, 5 edition, May 2005.

[Ste09] Steve Best. JFS overview, January 2009. http://www.ibm.com/developerworks/library/l-jfs.html.

[Sun08] Sun Microsystems. MySQL, December 2008. http://www.mysql.com/.

[Sun09] Sun Microsystems. OpenSolaris Community: ZFS, July 2009. http://opensolaris.org/os/community/zfs/.

[Sut05] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobb's Journal*, 30(3), March 2005.

[TvS06] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. Prentice Hall, October 2006.

[Twe98] Stephen C. Tweedie. Journaling the linux ext2fs filesystem. 1998.

[TZJW08] Avishay Traeger, Erez Zadok, Nikolai Joukov, and Charles P. Wright. A nine year study of file system and storage benchmarking. *Trans. Storage*, 4(2):1–56, 2008.

[Wik09] Wikipedia. Filesystem in userspace — Wikipedia, the free encyclopedia, January 2009. http://en.wikipedia.org/wiki/Filesystem_in_Userspace.

[WSSZ07] Charles P. Wright, Richard Spillane, Gopalan Sivathanu, and Erez Zadok. Extending acid semantics to the file system. *Trans. Storage*, 3(2):4, 2007.