UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

LEONARDO GARCIA DE MELLO

Validação experimental de sistemas de arquivos baseados em *journaling* para o sistema operacional Linux

Dissertação submetida à avaliação como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof[a] Dr[a] Taisy Silva Weber
Orientadora

Porto Alegre, dezembro de 2004

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Mello, Leonardo Garcia de

Validação experimental de sistemas de arquivos baseados em *journaling* para o sistema operacional Linux / Leonardo Garcia de Mello. – Porto Alegre: PPGC da UFRGS, 2004.
97f:il..

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul, Programa de Pós-Graduação em Computação, Porto Alegre, RS - BR, 2004. Orientadora: Taisy Silva Weber

1. Injeção de falhas. 2. Linux. 3. Sistemas de arquivos baseados em *journaling*. I. Weber, Taisy Silva. II. Título.

AGRADECIMENTOS

Como muitas pessoas participaram na elaboração deste trabalho e muitas outras o influenciaram de várias maneiras, não poderia deixar de expressar aqui a minha profunda gratidão a elas. Existem ainda aquelas pessoas que, mesmo sem perceber, me deram o impulso que faltava: com um abraço, um tapinha nas costas. À todas elas, meu amor e carinho.

Listar nomes é uma tarefa árdua e dolorosa, pois sempre há o risco de esquecer um e outro. Se você está lendo este agradecimento e o seu nome não está listado, considere-se parte dele. Agradeço, em especial,

- Aos meus pais (Luiz Carlos e Maria Eva) e irmãos (Leandro, Luciana, Vladimir e Emílson)
- Não tenho nem palavras apropriadas para dizer o quanto sou grato à Taisy, minha orientadora... que me acolheu como orientando e acreditou na conclusão deste trabalho, quando eu mais precisei disso.
- Aos meus amigos da Procuradoria Regional da República - 4ª Região, que acompanharam este processo
- Aos amigos Dé, Chinho, Leo, Rafael, Raquel, Édson, Alex, Rogério, Sirlei, Kellen, Clairton, Aline, Branden, Moser e Alexsandra. Vocês são o máximo!!
- À Deus, que não desistiu de mim - mesmo quando eu já havia desistido

Homem, considera que Eu fui o primeiro a amar-te
Você não estava ainda no mundo
O mundo nem existia, e Eu já o amava
Eu amo você desde que sou Deus
Amo você! E desde que amei a mim mesmo
Amei você também!

Santo Afonso

SUMÁRIO

LISTA DE ABREVIATURAS **8**

LISTA DE TABELAS **10**

RESUMO **11**

ABSTRACT **12**

1 INTRODUÇÃO **13**

1.1 Motivação **14**
1.2 Objetivos e resultados esperados **15**
1.3 Resultados alcançados **15**
1.4 Trabalhos relacionados **16**
1.5 Organização do texto **16**
1.6 Convenções **17**

2 INJEÇÃO DE FALHAS **18**

2.1 Introdução a injeção de falhas **18**
2.1.1 Injeção de falhas baseada em simulação 19
2.1.2 Injeção de falhas baseada em protótipos 19
2.1.3 Análise baseada em medições 20
2.2 Componentes de um ambiente para injeção de falhas **20**
2.3 Injeção de falhas por hardware **21**
2.4 Injeção de falhas por *software* **22**
2.5 Implementação de injetores de falhas por *software* **22**
2.5.1 Injeção de falhas no nível da aplicação 23
2.5.2 Injeção de falhas no nível do sistema operacional 26
2.6 Ferramentas para injeção de falhas por *software* **26**
2.6.1 ComFIRM 27
2.6.2 FIDe 28
2.6.3 FAU Machine 30
2.6.4 Orchestra 30
2.6.5 Xception 32
2.6.6 FIJI 34
2.7 Resumo **34**

3 SISTEMA DE ARQUIVOS LINUX **35**

3.1 Tipos de dispositivos 35
3.2 Sistemas de arquivos - *filesystems* 35
3.3 Metadados 36
***3.4 Linux Virtual Filesystem Switch - vfs* 37**
3.4.1 Estrutura *file_system_type* 38
3.4.2 Estrutura *super_block* 39
3.4.3 Estrutura vfsmount 39
3.4.4 Estrutura dentry 40
3.4.5 Estrutura *inode* 40
3.4.6 Estrutura file 41
3.5 Consistência do sistema de arquivos 41
3.5.1 Verificando a consistência de um sistema de arquivos 43
3.6 Resumo 44

4 SISTEMAS DE ARQUIVOS BASEADOS EM JOURNALING 45

4.1 Introdução 45
4.2 Sistemas de arquivos baseados em *journaling* 46
4.2.1 Conteúdo do *log* do *journaling* 47
4.2.2 Local de armazenamento dos logs 48
4.2.3 Verificação de consistência 49
4.2.4 Vantagens de sistemas de arquivos baseados em *journaling* 49
4.3 Motivação para escolha do XFS como alvo 50
4.4 Sistema de arquivos XFS 51
4.5 Escalabilidade do XFS 52
4.5.1 Sistemas de arquivos grandes 53
4.5.2 Arquivos grandes 53
4.5.3 Arquivos esparsos 55
4.5.4 Diretórios grandes 56
4.5.5 Alocação dinâmica de *inodes* 56
4.6 Recuperação de falhas no XFS 56
4.6.1 Estado do sistema de arquivos XFS 57
4.7 Resumo 57

5 AMBIENTE LINUX 58

5.1 Sinais 58
5.1.1 Introdução 58
5.1.2 Conceitos relacionados a sinais 59
5.1.3 Tipos de sinais 59
5.1.4 Relação de sinais permitidos pelo Linux 60
5.2 Chamadas de sistema 62
5.2.1 Introdução 62
5.2.2 Implementação das chamadas de sistema no Linux 62
5.2.3 Limitações das chamadas de sistema 63
5.2.4 Código de retorno de chamadas de sistema 63
5.3 Conceito de programa 64
5.4 Glibc 65
5.4.1 A função *write* 65
5.4.2 A função *read* 66
5.5 Recursos de depuração do Linux (ptrace) 66
5.6 Resumo 67

6 O INJETOR DE FALHAS FIJI 68

6.1 Introdução 68
6.2 Modelo de falhas 69
6.3 Metodologia de injeção 69
6.4 Plataforma de *hardware* 70
6.5 Plataforma de *software* 70
6.5.1 Sistema alvo 70
6.5.2 Monitoração e coleta de dados 70
6.5.3 Gerador de carga de trabalho 70
6.6 Arquitetura da ferramenta FIJI 72
6.7 Funcionamento do FIJI 72
6.7.1 Nível de abstração 74
6.7.2 Intrusividade 75
6.8 Detalhes de implementação 79
6.8.1 Iterações com o *kernel* 80
6.9 Resumo 81

7 TESTES E EXPERIMENTOS PARA INJEÇÃO DE FALHAS 82

7.1 Descrição do experimento manual 82
7.2 Sistema de arquivos baseado em blocos – ext2 83
7.2.1 Um único arquivo grande – ext2 83
7.2.2 Milhares de arquivos com tamanho aleatório – ext2 85
7.3 Sistema de arquivos baseado em *journaling* - XFS 85
7.3.1 Um único arquivo grande - XFS 85
7.3.2 Milhares de arquivos com tamanho aleatório - XFS 87
7.4 Análise dos resultados obtidos 88
7.5 Teste da ferramenta 89
7.6 Resumo 90

8 CONCLUSÕES 91

8.1 Introdução 91
8.2 Dificuldades encontradas 91
8.3 Melhorias na implementação atual 92
8.4 Resultados alcançados e trabalhos futuros 92

REFERÊNCIAS 94

LISTA DE ABREVIATURAS

ACID	*Atomicity, Consistency, Isolation and Durability*
ACL	*Access Control List*
AG	*Allocation Group*
COTS	*Commercial (Common)-Off-The-Shelf*
EXT2	*Second Extended File System*
IDT	*Interrupt Descriptor Table*
IDE	*Integrated Development Environment*
MTBF	*Mean Time Between Failures*
MTTF	*Mean Time To Failure*
MTTR	*Mean Time To Repair*
NFS	*Network File System*
PFI	*Protocol Fault Injection*
PIO	*Programmable input/output*
PPGC	Programa de Pós-Graduação em Computação
RAID	*Redundant Array of Independent Disks*
SPOF	*Single Point of Failure*
UFRGS	Universidade Federal do Rio Grande do Sul
VFS	*Virtual Filesystem Switch*

LISTA DE FIGURAS

Figura 1.1: Fórmula para cálculo da disponibilidade 13
Figura 2.1: Ambiente para injeção de falhas 21
Figura 2.2: Organização em camadas dos protocolos admitidos pelo Linux 27
Figura 2.3: Sintaxe das regras utilizadas pelo FIDe 29
Figura 2.4: Arquitetura do FIDe 29
Figura 2.5: Pilha do protocolo com camada de PFI 31
Figura 2.6: Estrutura do Xception 33
Figura 3.1: VFS com vários sistemas de arquivos no Linux 37
Figura 3.2: Estrutura que define um *file_system_type* 38
Figura 3.3: Estrutura para endereçamento em um *inode* 40
Figura 3.4: Exemplos de utilitários *fsck* para uma máquina Linux 43
Figura 4.1: Execução do comando xfs_info 49
Figura 4.2: Exemplo de uma B-TREE+ para o diretório /etc 52
Figura 4.3: Representação para a tripla que descreve um extent 55
Figura 5.1: Definição de um sinal 59
Figura 5.2: Declaração da função *write* 65
Figura 5.3: Declaração da função *read* 66
Figura 6.1: Arquitetura do FIJI 72
Figura 6.2: Opções disponíveis para uso com o FIJI 73
Figura 6.3: Iteração entre processo, Glibc e chamadas de sistema 74
Figura 6.4: Listagem do programa para testar intrusão – caso 1 76
Figura 6.5: Listagem do programa para gerar intrusão – caso 2 77
Figura 6.6: Listagem do programa para gerar intrusão – caso 3 78
Figura 6.7: Elementos obtidos por PTRACE_GETREGS 79
Figura 6.8: Busca da próxima chamada de sistema 80
Figura 6.9: Injeção de falha pelo FIJI 81
Figura 7.1: Ambiente para injeção manual de falhas 83
Figura 7.2: Definição de T_1 como sendo a diferença entre dois tempos 86
Figura 7.3: Definição de T_2 como sendo a diferença entre dois tempos 88
Figura 7.4: Listagem do programa para teste do FIJI 89

LISTA DE TABELAS

Tabela 3.1: Exemplos de sistemas de arquivos disponíveis para o Linux 36
Tabela 4.1: Sistemas de arquivos baseados em *journaling* para o Linux 47
Tabela 5.1: Lista de sinais Linux e ações padrão 60
Tabela 6.1: Intrusão no caso 1 76
Tabela 6.2: Intrusão no caso 2 77
Tabela 6.3: Intrusão no caso 3 78
Tabela 6.4: Intrusão do FIJI 79
Tabela 7.1: Tempo necessário para tornar uma partição ext2 consistente 84
Tabela 7.2: Tempo necessário para montar uma partição xfs 86

RESUMO

Alta disponibilidade (muitas vezes referenciada como *HA*, de *High Availability*) é uma característica de sistemas computacionais que são projetados para evitar ao máximo as interrupções, planejadas ou não, na prestação de serviços. Em alta disponibilidade, o ideal é haver poucas falhas e, mesmo quando estas acontecerem, que o seu tempo médio de reparo (ou *MTTR*, de *Mean Time To Repair*) seja tão pequeno quanto possível.

Sistemas operacionais têm um papel importante em alta disponibilidade, sendo preferível o uso daqueles que possuam sistemas de arquivos seguros e relativamente independentes de ações por agentes humanos para a recuperação. Uma das abordagens para auxiliar a obter-se uma alta disponibilidade em sistemas de arquivos é a do tipo *journaling*, ou *meta-data logging*. Existe uma série de sistemas de arquivos para o sistema operacional Linux baseando-se nela, tais como ext3, JFS, ReiserFS e XFS.

Este trabalho tem por objetivo propor uma metodologia de validação experimental para avaliar a eficiência do mecanismo para recuperação de sistemas de arquivos baseados em *journaling*, na ocorrência de falhas. Para isso, a técnica de validação empregada é a da injeção de falhas e o sistema sob teste é uma implementação do XFS.

Foram utilizados os recursos de depuração do sistema operacional Linux (que permitem a utilização de métodos para interceptação e manipulação de chamadas de sistema) para a implementação de um injetor de falhas específico para sistemas de arquivos baseados em *journaling*, o qual foi chamado de FIJI (*Fault Injector for Journaling fIlesystems)*. Manipular os parâmetros de chamadas de sistema (ou *system calls*) através do FIJI equivale a alterar as requisições feitas ao sistema operacional.

A eficiência do mecanismo de *journaling* é medida injetando-se falhas e medindo-se o MTTR e a cobertura de falhas. Basicamente, o que procura-se fazer através do injetor de falhas FIJI é ignorar os *logs* do *journaling* e manipular uma quantidade de informações diferente daquela que foi solicitada originalmente.

Palavras-chave: tolerância a falhas, injeção de falhas, Linux, sistema de arquivos, XFS

Experimental validation of Linux journaling filesystems

ABSTRACT

High Availability (sometimes referenced by its initials, HA) is a characteristic of computer systems projected to avoid interruptions, planned or not, on services being offered. In high availability, the ideal is to have no faults but, when they occur, which that their Mean Time To Repair (or MTTR) be as little as possible.

Operating systems are important to high availability, and those that have secure filesystems relatively independent of human actions for recovery should be preferred. An approach that helps to gain high availability for filesystems is journaling, or meta-data logging. There is a series of journaling filesystems for Linux, including ext3, JFS, ReiserFS and XFS.

This works aims to propose a way to experimentally validate the Linux journaling filesystems recovery efficiency in occurrence of faults. To do so, the techniques of Fault Injection are used for validation and the system under test is an XFS implementation.

The debugging resources of Linux operating system, which allows the utilization of methods for interception and manipulation of system calls, are used to implement a fault injector specific for journaling filesystems - called FIJI (Fault Injector for Journaling fIlesystems). Manipulate parameters of system calls through FIJI is the same thing as change requests made to operating system.

The journaling efficiency is measured by metering the Mean Time To Repair and Faults Coverage. Basically, what is done through FIJI fault injector is to drop the log of journal and inject faults to manipulate an amount of data different from that was originally requested.

Keywords: fault tolerance, fault injection, Linux, filesystems, XFS

1 INTRODUÇÃO

O termo conhecido como “alta disponibilidade” (muitas vezes também referenciado como *HA*, de *High Availability*) representa uma característica de sistemas computacionais que são projetados para evitar ao máximo as interrupções na prestação de serviços. A idéia de alta disponibilidade consiste em procurar manter um serviço [LYU 96], mesmo durante aquelas situações em que um ou mais componentes de um sistema possam ter se tornado indisponíveis. Esta indisponibilidade de um componente do sistema, por sua vez, pode ter sido causada tanto por algo planejado (tal como uma parada para realização de manutenção preventiva) quanto por algo inesperado (tal como o defeito em um servidor, por exemplo).

A maneira de alcançar alta disponibilidade consiste em empregar redundância, visando eliminar a existência dos pontos únicos de falha (ou *SPOF*, de *single point of failure*). Por exemplo, se uma organização possui apenas um *link* de acesso à Internet, então este *link* é um ponto único de falha. Para esta situação, o ideal seria contratar mais um *link* de acesso à Internet e fazer com que os roteadores de borda deste sistema autônomo estejam utilizando os protocolos de roteamento apropriados.

De um modo geral, a classificação para sistemas com alta disponibilidade é feita de acordo com a quantidade de "9s" em sua disponibilidade [LAP 92]. Esse cálculo de disponibilidade, por sua vez, pode ser feito de acordo com a fórmula da Figura 1.1 [TRI 82].

$$\text{Disponibilidade } (\alpha) = MTTF / (MTTF + MTTR)$$

Figura 1.1: Fórmula para cálculo da disponibilidade

Na Figura 1.1, a disponibilidade de um sistema é o atributo que representa a probabilidade de o mesmo estar acessível para prestar um determinado serviço quando este for requisitado. Durante as solicitações que forem feitas para um sistema ao longo de sua vida, ele poderá encontrar-se nos estados "funcionando" e "em reparo” [LAP 92].

O estado "funcionando" para um componente indica que ele está operacional, enquanto que o estado "em reparo" significa que este falhou e ainda não foi substituído. Havendo defeitos em um componente, o estado de todo o sistema muda de "funcionando" para "em reparo". Mas quando é feito o conserto, ele volta para o estado "funcionando". Sendo assim, pode-se dizer que o sistema possui um tempo até apresentar falha e um tempo de reparo. A sua vida é uma sucessão entre períodos de falha e de reparo.

Em alta disponibilidade, são preferíveis sistemas apresentando o tempo total de indisponibilidade como sendo pequenas interrupções na prestação de um serviço ao longo de todo um período - e sempre que possível, de maneira imperceptível aos usuários. Para a grande maioria dos casos, tem-se que o comportamento ideal é o de haver poucas falhas e, mesmo quando estas acontecerem, que o seu tempo de reparo seja tão pequeno quanto possível [WEY 2001].

Em sistemas com ou sem alta disponibilidade, as técnicas para injeção de falhas permitem acelerar a ocorrência das falhas para testá-los quanto à sua dependabilidade. Com isso, ao invés de esperar pela ocorrência espontânea de falhas, pode-se introduzi-las intencionalmente controlando-se o tipo, a localização, o disparo e a duração. Injeção de falhas pode ser implementada por *hardware*, *software* ou simulação [HSU 97]. Neste trabalho são enfocadas as técnicas para injeção de falhas por *software*.

O Linux dispõe de sistemas de arquivos baseados em *journaling* para auxiliar a obter alta disponibilidade, tais como ext3 [TWE 2003], JFS [JFS 2003], ReiserFS [REI 2003] e XFS [XFS 2003]. Esses sistemas de arquivos reduzem o tempo de verificação da consistência das informações de controle dos arquivos após a ocorrência de falhas, reduzindo assim o tempo de recuperação e reparo.

Este trabalho propõe uma validação das propriedades relacionadas com tolerância a falhas para o sistema de arquivos XFS, através da injeção de falhas por *software*. Uma verificação experimental através do monitoramento do sistema até que uma falha real ocorresse seria impraticável, visto que a ocorrência de falhas não é determinística e nem previsível [HSU 97]. Entretanto, técnicas para injeção de falhas permitem que aplique-se falhas específicas sobre um sistema de arquivos de interesse, facilitando a comprovação de que ele realmente tolera o modelo de falhas que propõe-se a lidar. Para este caso, o modelo de falhas encontra-se descrito no capítulo 6.

1.1 Motivação

A grande motivação deste trabalho é que o tempo necessário para execução do utilitário *fsck* geralmente representa grande parte do tempo necessário para restabelecer um sistema [PIE 2002]. Em alta disponibilidade, o ideal é fazer com que todos os componentes (incluindo os sistemas de arquivos) tenham tempos de reparo mínimos.

A grosso modo, todas as operações realizadas sobre um sistema de arquivos podem ser divididas em duas grandes categorias: [SEL 92]

- operações envolvendo dados - atuam diretamente sobre blocos de dados em disco, onde informações de usuário estão sendo mantidas. Exemplo disso são instruções para a leitura e escrita de arquivos;
- operações envolvendo metadados - modificam as estruturas internas de organização do sistema de arquivos, tais como as operações para criar, remover ou renomear arquivos e diretórios.

Durante as operações envolvendo dados e metadados, é preciso que a representação do sistema de arquivos nos discos seja mantida consistente – isso mesmo após a ocorrência de falhas. Em sistemas de arquivos como o ext2, construídos com alocação baseada em blocos, o sistema de arquivos pode ficar inconsistente após a ocorrência de falhas porque dados e metadados são gravados em disco de forma assíncrona – por razões de desempenho [TAN 97][BAR 2000]. Sempre que algo assim acontece, é preciso executar algum utilitário de verificação - como o *fsck* [PIE 2002].

O *fsck* realiza uma série de passos ao longo de todo o sistema de arquivos de um dispositivo para validar as suas entradas e assegurar-se de que os blocos alocados em disco estão todos sendo referenciados corretamente. Mas infelizmente, para unidades de tamanho muito grande, a execução do *fsck* pode consumir um tempo bastante elevado [PIE 2002]. Em uma máquina com muitos *gigabytes* em arquivos, por exemplo, a execução do *fsck* pode demorar até 10 horas ou mais [BAR 2000].

Assim, levando-se em conta essas dificuldades, surgiram as propostas de sistemas de arquivos para alta disponibilidade. Por meio deles, é possível diminuir a chance de inconsistências virem a ser introduzidas nos sistemas de arquivos e, mesmo quando isso por ventura aconteça, reduzir o tempo de reparo. Atualmente as principais técnicas empregadas para auxiliar a obter alta disponibilidade em sistemas de arquivos são o *journaling* e o *Soft Updates* [SEL 2000].

Enquanto o *Soft Updates* só está disponível em sistemas BSD, o *journaling* é utilizado pela grande maioria dos outros sistemas operacionais. A técnica de *journaling* baseia-se na redundância para aumentar a confiabilidade dos dados e metadados, mas sem aumentar significativamente os custos de *hardware* [TWE 98]. Ela já é adotada por sistemas de arquivos em sistemas operacionais para plataformas diversas, tais como Solaris, AIX, Digital UNIX, HP-Ux, Irix e Windows NT [SEL 2000].

Apesar de os sistemas de arquivos baseados em *journaling* terem sido adotados quase como um padrão pela indústria de *software*, atualmente não se encontram publicações sobre avaliação de medidas da sua disponibilidade.

1.2 Objetivos e resultados esperados

Este trabalho tem por objetivo definir uma metodologia de validação experimental para avaliar a eficiência do mecanismo de *journaling* para sistemas de arquivos na ocorrência de falhas. Para isso, utiliza-se a técnica de injeção de falhas para validação, tendo como sistema alvo uma implementação do XFS.

A partir de um estudo dos mecanismos para tolerância a falhas do *journaling*, foi escolhida uma das implementações de sistemas de arquivos disponíveis atualmente para o ambiente Linux. A escolha deveria limitar-se entre ext3, JFS, ReiserFS e XFS, e basear-se em critérios tais como existência de documentação, disponibilidade de aplicativos ou mesmo facilidade para manutenção no código-fonte. O XFS preenche tais critérios [XFS 2003].

Através de injeção de falhas no sistema de arquivos XFS, primeiramente é feita uma medida da cobertura e do tempo de reparo. Espera-se, com isto, identificar gargalos na dependabilidade oferecida pelo *journaling*. Finalmente, espera-se desenvolver uma metodologia para injeção de falhas em sistemas de arquivos baseados em *journaling* através de uma ferramenta de *software*.

1.3 Resultados alcançados

A ferramenta FIJI, desenvolvido nesta dissertação, foi utilizada para injetar falhas no sistema de arquivos XFS. Levando-se em conta que o estado de um sistema de arquivos XFS é formado pelas informações armazenadas em disco, pelo conteúdo dos *cache/buffers* e pelo *log* do *journaling*; foi possível, através do FIJI, ignorar o *log* do *journaling* e impedir que informações armazenadas em disco fossem manipuladas.

Neste trabalho já se encontram resultados significativos, assegurando que a ferramenta está operacional e que a abordagem utilizada é digna de estudos mais aprofundados. Foram realizados testes usando aplicações simples, comprovando que as requisições feitas pelas aplicações são efetivamente alteradas.

1.4 Trabalhos relacionados

Diversos trabalhos relacionados a sistemas de arquivos e a injeção de falhas em plataforma Linux já foram apresentados. Entre estes, vale a pena destacar:

- *Ballista* – o projeto Ballista está relacionado com o uso de COTS (categoria de *software* na qual o Linux enquadra-se), e forneceu critérios importantes sobre como proceder a análise de robustez em sistemas POSIX (padrão com o qual o Linux é compatível) [KOO 99];
- *Faumachine* – é um simulador de ambiente Linux, em que são criadas máquinas virtuais. Na camada de *hardware* de uma *faumachine*, podem ser injetadas falhas por meio do *scripts* através do *Expect* [Seção 2.6.3]; e
- *FIDe* – é baseado na depuração de programas pela sua execução até uma chamada de sistema. O FIDe baseia-se na chamada de sistema *ptrace* para injetar falhas, baseando-se em um *script* para injetar falhas [Seção 2.6.2].

1.5 Organização do texto

O texto desta dissertação está organizado para apresentar o assunto com profundidade crescente, porém, deixando de lado os conceitos mais básicos de tolerância a falhas - visto que o seu público-alvo já os deve conhecer.

No capítulo 2 são brevemente revisadas as técnicas para injeção de falhas, e são apresentadas algumas ferramentas para injeção de falhas por *software* que foram consideradas para esta dissertação. Já no capítulo 3, é apresentada a organização básica para sistemas de arquivos em ambiente Linux.

O capítulo 4 apresenta como é a organização para sistemas de arquivos baseados em *journaling*, e trata especificamente sobre a implementação do XFS. O capítulo 5 descreve como funcionam os sinais e as chamadas de sistema no ambiente Linux, e revisa os mecanismos para depuração disponíveis no Linux. A execução de processos e suas iterações são estudadas para que seja possível entender a chamada de sistema *ptrace* – base deste projeto.

O capítulo 6 descreve a ferramenta para injeção de falhas FIJI: a metodologia para injeção de falhas, sua arquitetura e implementação encontram-se descritos em detalhes. Define o modelo de falhas e descreve o ambiente de *hardware* e *software* considerado para a realização dos experimentos de injeção de falhas.

O capítulo 7 apresenta os resultados de experimentos manuais com os sistemas de arquivos ext2 e XFS, e também os testes para verificar se a ferramenta FIJI realmente está injetando erros.

O capítulo 8 encerra o trabalho, apresentando conclusões e indicando possibilidades para pesquisas futuras.

1.6 Convenções

Para uma melhor compreensão do texto, adotou-se a seguinte convenção:

- Palavras e expressões em inglês aparecem em *itálico*, e
- Comandos aparecem em **negrito.**

2 INJEÇÃO DE FALHAS

Este capítulo apresenta uma visão global sobre injeção de falhas, e as diferentes abordagens para injeção de falhas por *software*. Neste ponto, o leitor familiarizado com os conceitos básicos na área não encontrará nenhuma contribuição nova sobre o tema.

Também encontram-se relacionadas algumas características das ferramentas para injeção de falhas por *software* que foram consideradas para esta dissertação.

2.1 Introdução a injeção de falhas

Sistemas computacionais são compostos por *hardware* e *software* que podem vir a falhar, eventualmente. Essas falhas, por sua vez, podem levar um sistema a um estado em que esteja apresentando um defeito - ou seja, o serviço oferecido pelo mesmo não esteja mais de acordo com aquilo o que foi especificado.

As falhas geram erros, que podem levar a defeitos, e normalmente são ocasionadas por um mau funcionamento de *hardware* ou algum componente de *software* [LAP 92]. Estes defeitos, quando em sistemas utilizados para o controle de atividades críticas, podem acarretar grandes perdas econômicas ou mesmo de vidas humanas.

Atualmente existe uma grande dependência quanto ao correto funcionamento de sistemas computacionais. A utilização destes em aplicações de missão crítica como o controle de tráfego aéreo, bancos 24 horas, equipamentos médicos, controle de sistemas para direcionamento de mísseis e outras tantas atividades motivam estudos no sentido de aumentar a dependabilidade (ou segurança de funcionamento) destes sistemas.

Na verdade, mesmo sistemas que não coloquem em risco vidas humanas ou vultosas somas em dinheiro podem ser beneficiados por um aumento em sua dependabilidade. *Clusters* para processamento de alto desempenho, sistemas operacionais e ambientes de desenvolvimento com maior dependabilidade podem aumentar a produtividade.

Visando este aumento de dependabilidade, podem ser utilizadas técnicas de tolerância a falhas nestes sistemas. Estas técnicas podem ser implementadas por *hardware*, *software* ou ambos [HSU 97]. As técnicas são escolhidas, também, em função do modelo de falhas adotado para o sistema. Mas o simples fato de utilizar estas técnicas, porém, não garante um aumento imediato na dependabilidade de um sistema – a utilização de mecanismos não apropriados pode até mesmo diminuí-la.

Mesmo bons programadores podem cometer erros, portanto, não há como garantir que de uma especificação correta sempre obtém-se uma implementação correta. Mas assim como é possível testar e comprovar formalmente uma especificação, o mesmo também pode ser feito com uma implementação. Porém, levando-se em conta a atual complexidade dos sistemas computacionais, é praticamente impossível antecipar e

prever, em tempo razoável, todas as inúmeras possibilidades de comportamento de uma implementação quando da ocorrência de falhas.

A validação da dependabilidade envolve o estudo de falhas, erros e defeitos. A natureza destrutiva de um colapso e a longa latência de erros dificultam a identificação das causas de defeitos em um ambiente. É particularmente difícil recriar o cenário de um defeito que tenha acontecido em um sistema amplo e complexo [HSU 97].

Além disso, existem muitos sistemas que não podem ser testados em um ambiente de produção. Testar um sistema de piloto automático durante um vôo doméstico, por exemplo, seria inadmissível. Além disso, a natureza não-determinística e imprevisível das falhas pode fazer com que uma dada falha não ocorra durante todo o período de testes – por maior que este seja. Logo, a injeção de falhas é uma das possibilidades para esta validação.

Para identificar e entender defeitos em potencial, utiliza-se uma abordagem baseada em experimentação para estudar a dependabilidade de um sistema. Uma abordagem deste tipo pode ser aplicada não somente nas fases de concepção e projeto, mas também ao longo das etapas de prototipação e operação.

Para adotar-se uma abordagem experimental, deve-se primeiramente entender a arquitetura, a estrutura e o comportamento de um sistema. Especificamente, deve-se conhecer a sua tolerância a falhas e a defeitos, incluindo seus mecanismos internos de detecção e recuperação de erros. Também precisa-se de instrumentos e ferramentas específicas para injetar falhas/erros e monitorar os seus efeitos.

2.1.1 Injeção de falhas baseada em simulação

Pode-se utilizar injeção de falhas baseada em simulação para validar a dependabilidade de um sistema durante as fases de concepção e projeto. Neste ponto, o sistema pode ser visto como uma série de abstrações de alto nível. Com base nestas abstrações, cria-se um modelo para o sistema – geralmente simplificado. Modelos analíticos ou gerados em sistemas de simulação podem ser utilizados [HSU 97].

A injeção de falhas é baseada em alguma distribuição estatística considerada conveniente para o sistema sob avaliação. Neste ponto, assume-se que detalhes de implementação são desconhecidos e que algumas definições poderão mudar nas fases de prototipação e implementação. Entretanto, as simplificações feitas no modelo utilizado podem comprometer os resultados obtidos.

2.1.2 Injeção de falhas baseada em protótipos

Nas fases seguintes, pode-se testar um protótipo do sistema – o que fornece resultados mais próximos da realidade. Na injeção de falhas baseada em protótipos, as falhas são injetadas no sistema para [HSU 97]:

1. Identificar gargalos na dependabilidade,
2. Estudar o comportamento do sistema na presença de falhas,
3. Determinar a cobertura dos mecanismos de detecção de erros e recuperação, e
4. Validar a eficiência dos mecanismos de tolerância a falhas e mensurar a perda de desempenho.

A injeção de falhas baseada em protótipos pode ser realizada por *hardware*, por *software*, ou em arranjos híbridos. Realizada no sistema já operacional, pode fornecer informações a respeito da ocorrência de defeitos.

Entretanto, a injeção de falhas baseada em protótipos é indicada para estudar apenas as falhas que forem emuladas – logo, não há como prever medidas como tempo médio entre falhas (MTBF) e disponibilidade. Também não é possível medir o MTTR diretamente.

2.1.3 Análise baseada em medições

Com o sistema já pronto pode-se ainda, ao invés de injetar falhas, realizar medições do sistema durante a sua operação. Assim, pode-se estudar o comportamento do sistema por meio da análise baseada em medições [HSU 97].

Coletam-se dados do funcionamento do sistema, enquanto ele manipula cargas de trabalho reais, e realiza-se uma análise destes dados segundo um modelo definido. Estes dados contém muita informação a respeito da ocorrência natural de erros e defeitos. Além disso, também podem conter informações a respeito de tentativas de recuperação. A análise destes dados pode prover entendimento a respeito das características de erros e defeitos.

2.2 Componentes de um ambiente para injeção de falhas

A injeção de falhas (nome genérico dado ao conjunto de técnicas utilizadas para acelerar a ocorrência de falhas, erros e defeitos [LAP 92] em um sistema) é largamente utilizada na validação dos mecanismos para tolerância a falhas. A injeção de falhas permite-nos validar os mecanismos de tolerância a falhas na presença de entradas particulares para as quais eles foram desenvolvidos para tratar: as falhas.

Pode-se caracterizar um ambiente para injeção de falhas segundo a organização proposta por Hsueh, e que segue na Figura 2.1 [HSU 97]. De acordo com ela, um ambiente para injeção de falhas tipicamente é formado por um sistema alvo, um injetor de falhas, uma biblioteca para o injetor de falhas, um gerador de carga de trabalho, uma biblioteca para o gerador de carga de trabalho, um controle, um monitor, um coletor de dados e um analisador de dados [HSU 97].

O componente chamado controle é quem coordena a realização dos experimentos. Fisicamente, o componente controle é um programa que pode estar sendo executado tanto no sistema alvo como em uma máquina em separado.

O controle faz com que o injetor de falhas insira falhas no sistema alvo, ao mesmo tempo em que o injetor de carga de trabalho execute comandos. O monitor acompanha a execução dos comandos, e inicia a coleta de dados sempre que isto for necessário. O coletor de dados recebe as informações de modo *on-line*, mas o componente para análise de dados pode ser *off-line*.

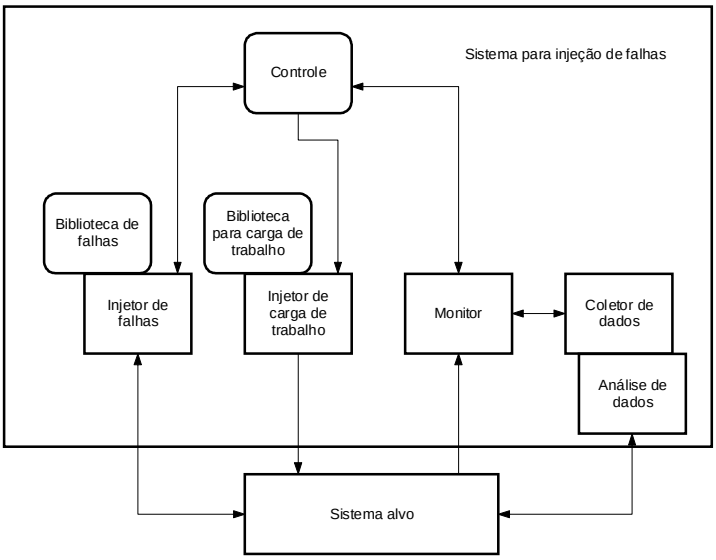

Figura 2.1: Ambiente para injeção de falhas

2.3 Injeção de falhas por hardware

Nesta abordagem, utiliza-se algum *hardware* adicional para injetar falhas no sistema alvo [HSU 97]. Falhas de *hardware* que ocorrem durante a operação dos sistemas podem ser classificadas pela sua duração em [LAP 92]:

- **Permanentes**: causadas por falhas irreversíveis de componentes, por exaustão de seu uso, falhas de projeto e fabricação ou ainda pelo seu mau uso. Estas falhas somente podem ser reparadas pela troca do componente ou o seu conserto;
- **Transientes**: são causadas por condições ambientais como flutuação de voltagem, distúrbios eletromagnéticos ou radiação. Normalmente estas falhas não danificam os componentes envolvidos, mas podem levar o sistema a um estado errôneo ou inesperado. Estas falhas são muito difíceis de detectar, e ocorrem com muito mais freqüência do que as falhas permanentes; e
- **Intermitentes**: ocorrem devido a uma instabilidade de *hardware* ou variação de seu estado. Podem ser consertadas pela troca do *hardware* ou ainda um reprojeto do mesmo.

Falhas de *hardware* temporárias ou permanentes, que afetem um computador isolado, podem propagar-se pela rede em um sistema distribuído, afetando vários outros nodos. Nesses casos, para evitar deter-se nos detalhes de um único nodo, usa-se uma abstração de maior nível que classifica as falhas em sistemas distribuídos como falhas de comunicação. Nessa visão, pode ser adotado o modelo de Christian: colapso (ou *crash*), omissão, temporização e bizantinas [CHR 91].

Tem-se o costume de atribuir a causa das falhas dos sistemas às falhas de *software* ou às falhas de *hardware* permanentes, pela dificuldade de identificar as falhas de *hardware* transientes e intermitentes. Considerando-se que as falhas podem ter as mais diversas origens, e também o seu caráter aleatório e freqüentemente não repetitível, a injeção de falhas constitui-se em uma importante forma de avaliar e validar os mecanismos de tolerância a falhas em sistemas de computadores [HSU 97].

2.4 Injeção de falhas por *software*

É uma técnica em que as falhas são logicamente introduzidas, através de *software* específico, no sistema alvo. O seu objetivo é o de emular as conseqüências de eventuais falhas físicas [HSU 97]. Esta abordagem também poderia ser chamada de injeção de erros, visto que é só o que se pode fazer.

A injeção de falhas por *software* consiste em criar código que simule a ocorrência das falhas necessárias à validação de um sistema. Este código deve alterar o estado do sistema, levando-o a perceber uma determinada falha.

As falhas de *software* ocorrem, normalmente, por erros de especificação, de projeto ou de codificação. Muitas dessas falhas podem ficar latentes e só ocorrerem durante a operação do sistema, especialmente quando sob pesadas ou não usuais cargas de trabalho, ou sob determinados intervalos de tempo. Os *bugs* de natureza não-determinística são os mais difíceis de eliminar por verificação, validação ou teste e, normalmente, os sistemas podem contê-los durante todo o seu ciclo de vida.

A injeção de falhas por *software* apresenta as seguintes vantagens [HSU 97]:

- Não necessita de equipamentos especiais de *hardware*;
- Baixo custo;
- Pode ser aplicada na fase de testes, pois não necessita que o *hardware* em que irá atuar esteja disponível;
- Possibilita maiores facilidades no controle e observação do sistema durante os testes;
- Não há risco de danificar os componentes sob teste; e
- Maior portabilidade, pois normalmente não precisam ser específicas a um determinado sistema alvo.

Esta técnica é adequada para a validação de mecanismos de tolerância a falhas implementada por *software*, por possuir a capacidade de injetar falhas específicas a um determinado sistema alvo. Mas em contrapartida, esta técnica normalmente possui grandes desvantagens:

- injetor de falhas pode causar um grande impacto sobre o sistema alvo, que pode alterar seus resultados devido ao *overhead* causado pela inserção do código do injetor de falhas, e
- ocasiona dificuldade para implementar alguns tipos de falhas devido às limitações da maioria dos modelos

Apesar das desvantagens, suas vantagens - aliadas à flexibilidade para a escolha do modelo de falhas - definiram ser esta como sendo a técnica a ser utilizada nesta dissertação.

2.5 Implementação de injetores de falhas por *software*

Injetores de falhas não podem estar presentes sem causar alguma alteração da carga ou das características temporais dos sistemas [LEI 2000]. Como uma classificação genérica, pode-se dizer que um injetor de falhas pode estar, dependendo de seu objetivo, em dois níveis: ou no nível da aplicação, ou no nível do sistema operacional.

Se o alvo da injeção de falhas é uma aplicação, o injetor de falhas pode ser inserido em vários locais: na própria aplicação (necessita da disponibilidade do código-fonte ou objeto), entre a aplicação e o sistema operacional (em bibliotecas, *wrappers*, ou mesmo via depuradores) ou no próprio sistema operacional (permitindo a execução da aplicação de forma inalterada).

Mas se o alvo for algum componente do sistema operacional, o injetor de falhas deve ser embutido no mesmo – visto que é bastante difícil colocar uma camada entre o sistema operacional e o *hardware*.

2.5.1 Injeção de falhas no nível da aplicação

A inserção do injetor de falhas no nível da aplicação visa a introdução de falhas ou na própria aplicação em teste, ou em código que execute no mesmo nível. Injetores que implementam esta abordagem podem estar no código da aplicação, em código concorrente com a aplicação, em código usado pela aplicação ou no meta-nível.

2.5.1.1 Injeção no código da aplicação

Considerando a aplicação a ser validada e o injetor de falhas um único programa, existe a necessidade de contar com o código-fonte da aplicação para serem realizadas alterações. Isto porque se o injetor de falhas fará parte da aplicação durante a execução dos testes, são necessárias alterações no código-fonte da aplicação para que a mesma possa atuar em conjunto com o injetor. A aplicação sob teste deve incluir, em seu código, chamadas às rotinas do injetor de falhas.

A cada chamada ao injetor, a aplicação sob teste deve desviar a sua operação normal, executar as ações do injetor e então retornar à sua execução. O posicionamento correto das chamadas no código-fonte da aplicação sob teste exige uma análise cuidadosa do código e uma boa documentação da aplicação sob teste. Para injeção de falhas de comunicação, por exemplo, todas as chamadas a rotinas como *send, receive, broadast, multicast, deliver* e outras funcionalidades semelhantes devem ser substituídas por rotinas do injetor de falhas. Alternativamente, as próprias rotinas de comunicação do protocolo podem ser alteradas [LEI 2000].

Esta abordagem de implementação restringe a utilização do injetor para aquela aplicação específica sob teste, que sofreu alteração em seu código-fonte. A portabilidade para outras aplicações fica comprometida. Cada nova aplicação a ser validada deve ter seu código-fonte minuciosamente analisado para determinação dos melhores pontos para posicionamento das chamadas às rotinas de injeção de falhas.

Outra desvantagem corresponde à integridade da aplicação. Deve-se garantir que o procedimento de validação não irá comprometer a execução da aplicação em si. Como o injetor é executado no mesmo espaço de endereçamento que a aplicação, com os mesmos privilégios, ele tem acesso a todos os recursos da aplicação e pode, inadvertidamente, alterar o comportamento da aplicação - mascarando as medidas de confiabilidade que poderiam ser obtidas pelo teste.

Essa abordagem, entretanto, pode ser bastante útil quando o injetor é desenvolvido simultaneamente com a aplicação. Nesse caso, o desenvolvedor tem perfeito domínio sobre o código-fonte e pode tirar proveito da ausência de proteção do espaço de endereçamento da aplicação em relação ao injetor.

Ainda no nível da aplicação, existe a possibilidade de inserir em um programa código adicional que crie uma *thread* extra que executará o injetor. Este é o ponto máximo de intrusão que pode-se chegar, visto que altera o código-fonte, altera o ambiente de execução, compartilha espaços de memória e rouba ciclos de CPU do sistema em teste. Problemas inesperados com o injetor podem atrapalhar a aplicação, e pode ser necessário mudar a característica de implementação dos algoritmos do sistema sob teste para acomodar a inspeção e possível alteração pelo injetor.

Para tirar o máximo proveito da implementação desta abordagem, entretanto, o sistema operacional deve permitir a execução de *threads* [BAR 2000]. Assim, quando a *thread* da aplicação em teste abandonar a CPU por alguma razão, a *thread* do injetor poderá executar e liberar a CPU antes que a aplicação volte à execução. O impacto sobre a temporização do sistema nesse caso é mínima, considerando que o injetor execute apenas umas poucas e rápidas tarefas cada vez que assuma a CPU. Se *threads* são permitidas apenas pela linguagem de programação, quando a aplicação abandonar a CPU levará junto o injetor e a vantagem da redução do tempo de execução anular-se-á.

2.5.1.2 Injeção por processos concorrentes à aplicação

Esta abordagem consiste na criação de um processo injetor, que vai de alguma forma alterar o ambiente de execução do processo simulando a ocorrência de falhas. Uma possibilidade é a injeção de falhas via chamadas de sistema [GON 2001].

Esta técnica é especialmente interessante por não necessitar da disponibilidade do código-fonte do sistema em teste. Esta característica é vital para organizações independentes de teste e validação, que recebem sistemas prontos, fechados, cujo código-fonte normalmente é coberto por patentes e registros comerciais.

Neste caso, o processo correspondente à aplicação sob teste e o processo injetor de falhas executam concorrentemente. Somente haverá interação entre os dois processos na inserção de uma falha e na posterior coleta de resultados da inserção. Entretanto, tais procedimentos não implicam em alterações de código dos processos [SIE 93].

Para interagir com as aplicações e sem alterar o seu código, o processo injetor de falhas deve receber privilégios especiais para alterar determinadas áreas de dados do processo da aplicação. Para tanto, o injetor deve saber determinar quais áreas alocadas para a aplicação, a cada momento, que correspondem à manipulação de dados.

Determinar as áreas e obter privilégios de acesso a outros processos não é uma tarefa trivial para processos no nível da aplicação, e está intimamente relacionada aos recursos que um determinado sistema operacional pode oferecer aos processos. Se entretanto, for prevista no desenvolvimento da aplicação a validação por um injetor, as áreas de manipulação de dados podem ser declaradas como de acesso comum a mais de um processo - facilitando a obtenção dos privilégios necessários por parte do injetor.

Outra vantagem em relação à abordagem anterior refere-se à portabilidade. Sendo a aplicação um processo em separado do injetor de falhas é possível, com pouca ou nenhuma alteração, utilizar o mesmo injetor de falhas para aplicações com características semelhantes.

A desvantagem com relação à integridade da aplicação, apresentada pela abordagem anterior, é consideravelmente menor nesta abordagem, mas não totalmente ausente. Isto deve-se ao fato de que o injetor de falhas não interfere diretamente no código da aplicação, ou seja, a execução dos módulos injetor de falhas e aplicação em dois processos distintos evita a interferência de um no código do outro. A única

interação que ocorre nesta abordagem refere-se à introdução das falhas, por parte do injetor, na área de manipulação de dados durante a execução.

Não existe alteração manual do código-fonte e nem interferência dinâmica na área de código durante a execução – isto se o esquema de proteção do sistema operacional for adequado. Entretanto, nesta abordagem o custo de chaveamento de contexto para a execução alternada dos dois processos pode ser considerável.

Se o objetivo é minimizar a carga do injetor no sistema e evitar comprometer o seu comportamento temporal, a aplicação dessa solução deve considerar cuidadosamente os tempos envolvidos no chaveamento de contexto.

2.5.1.3 Injeção em código utilizado pela aplicação

Uma abordagem de implementação interessante é a utilização de bibliotecas dinâmicas modificadas, cujo uso pode ser ativado ou desativado facilmente. Nos sistemas padrão Unix, por exemplo, a variável LD_LIBRARY_PATH pode conter um diretório onde encontram-se bibliotecas a serem carregadas antes de verificar os diretórios padronizados do sistema operacional [BAR 2000].

Isto permite que seja executado um mesmo programa, sem alterações, porém que sofrerá a ação de uma biblioteca dinâmica com funções modificadas e que podem muito bem injetar qualquer tipo de erro no processo ao qual estejam ligadas. Esta técnica parece-se muito com as técnicas que envolvem alteração no código da aplicação – porém, não necessita do código-fonte.

É aplicável somente a programas que não tenham sido *linkados* estaticamente, para que usem a biblioteca dinâmica alterada [LAD 98]. Novamente um problema no injetor pode atrapalhar a aplicação. Na verdade, esta abordagem mistura algumas vantagens e desvantagens tanto da inserção no código da aplicação quanto da inserção em processos concorrentes à aplicação.

2.5.1.4 Injeção no metanível

Pode-se utilizar técnicas de programação reflexiva para a implementação do injetor de falhas [ROS 98]. Reflexão é uma maneira fácil de separar funcionalidades de implementação do sistema alvo. A reflexão introduz um novo modelo de arquitetura, no qual existem dois níveis: o metanível e o nível funcional.

O nível funcional ou nível base [MAE 87] pode ser usado para implementar os objetos do sistema alvo, enquanto o metanível permite que programadores observem e manipulem estruturas de dados e/ou ações realizadas no nível funcional. Desta forma, chamadas a métodos podem ser interceptadas e então parâmetros de entrada ou saída podem ser monitorados e modificados. Um perfil da execução no nível funcional pode ser obtido, e falhas podem ser injetadas com uma ativação bastante flexível.

Uma das vantagens desta abordagem é a transparência, uma vez que para o procedimento de injeção não há interrupção explícita do código da aplicação ou execução desta em modo de depuração - como ocorre usualmente em mecanismos para injeção de falhas por *software*. Outra vantagem é o reuso, já que os objetos responsáveis tanto pela injeção de falhas como pelo monitoramento podem ser incorporados a outras aplicações. Além disso, injeção de falhas no metanível não necessita da execução do programa em modo privilegiado, tampouco é preciso *hardware* dedicado.

Injetores de falhas no metanível prestam-se bem a sistemas orientados a objetos e podem ser usados para validar protocolos de comunicação em sistemas distribuídos. Os metaobjetos do metanível podem interceptar a comunicação entre objetos distribuídos e dessa forma selecionar e manipular as mensagens desejadas durante a injeção de falhas, por exemplo [JAC 2004].

2.5.2 Injeção de falhas no nível do sistema operacional

Um processo só conhece o mundo externo através das chamadas de sistema que o sistema operacional lhe proporciona [TAN 97]. Se um processo implementa um protocolo de comunicação confiável, por exemplo, ele pede através de chamadas de sistema que o sistema operacional entregue estas mensagens. Para este processo, uma falha real na rede e uma falha injetada pelo sistema operacional são indistingüíveis. Assim também acontece para falhas em memória e em registradores, por exemplo.

Esta abordagem possui a vantagem de permitir que a aplicação sob teste execute sem qualquer interferência do injetor de falhas. Esta característica aumenta o reuso do injetor de falhas. A portabilidade da ferramenta está vinculada à portabilidade do sistema operacional usado como plataforma. Entretanto, por encontrar-se no nível do sistema operacional e utilizar-se dele para atuar, o injetor de falhas pode interferir em sua integridade. Além disso, há necessidade de alterações do próprio sistema operacional - que deve permitir a presença de intrusos.

Aqui é importante salientar que a forma com que a falha é injetada pode acabar por dar um significado diferente à mesma. Suponhamos uma mensagem gerada por um processo e que deve ser descartada pelo sistema operacional, para simular uma falha de perda de pacote na rede. Caso a falha real acontecesse, a mensagem passaria por todos os protocolos de rede utilizados, atualizando números de série de *flags*, por exemplo, e então seria enviada pela placa de rede, que colocaria em suas estatísticas o número de mensagens enviadas, de *bytes*, de pacotes com erro, colisões, etc. A mensagem seria perdida na rede, porém, já teria influenciado todo o código por onde passou.

Sendo assim, se fosse escolhido descartar a mensagem no ponto mais próximo do processo – digamos, na chamada de sistema *socket_call* - esta não influenciaria todo o código do protocolo TCP e, portanto, os números de série estariam errados para a próxima mensagem. Do ponto de vista do protocolo, não houve mensagem. Isto não serve como uma emulação de falha da rede, pois a próxima mensagem será entregue normalmente e tudo prosseguirá como se aquela mensagem descartada nunca tivesse sido gerada. Isto é uma falha da aplicação, não da rede [LEI 2000].

2.6 Ferramentas para injeção de falhas por *software*

Foi realizado um estudo sobre algumas das várias ferramentas para injeção de falhas disponíveis na atualidade. Aquelas ferramentas que podem ser consideradas como sendo as principais encontram-se destacadas no texto que segue.

O próprio grupo de Tolerância a Falhas da UFRGS já desenvolveu uma série de ferramentas para injeção de falhas em aspectos diversos de sistemas. Entretanto, nenhum dos injetores desenvolvidos pelo grupo de Tolerância a Falhas pôde ser utilizado para este trabalho. Foram realizados vários experimentos com o FIDe (uma das ferramentas mais promissoras) mas sem sucesso, por motivos que serão relatados posteriormente.

Atualmente não se encontram publicações sobre validação usando injeção de falhas para sistemas de arquivos baseadas em *journaling*. Isso é algo que desperta o interesse em realizar injeção de falhas em uma das implementações baseadas nesta abordagem para o ambiente Linux.

2.6.1 ComFIRM

A ferramenta ComFIRM (*Communication Fault Injection through OS Resources Modification*) [LEI 2000], desenvolvida no PPGC da UFRGS, é um injetor de falhas de comunicações bastante abrangente e flexível, que situa-se dentro do *kernel* do sistema operacional Linux. O ComFIRM está localizado no ponto mais baixo do tratamento de mensagens pelo subsistema de rede, na camada independente de dispositivos – como mostra a Figura 2.2 [LEI 2000]:

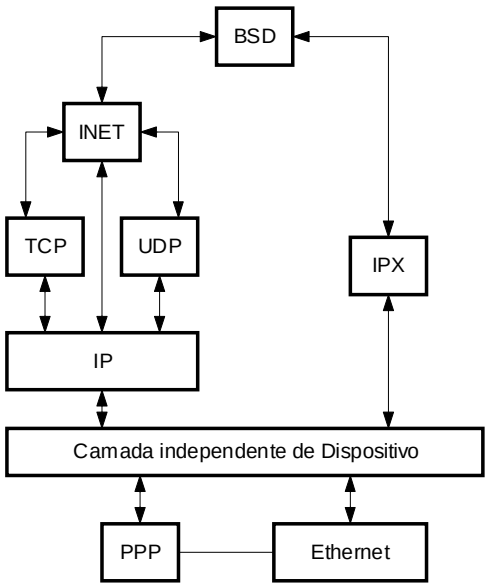

Figura 2.2: Organização em camadas dos protocolos admitidos pelo Linux

Para utilização do ComFIRM, é necessária uma pequena modificação no *kernel* do sistema operacional - onde introduz-se as chamadas da ferramenta. Isto não constitui um problema, haja visto que todas a distribuições Linux trazem seu código-fonte. Basicamente, a implementação do ComFIRM alterou as rotinas do *kernel* que transmitem (**dev_queue_xmit**) e que recebem pacotes do subsistema de rede (**netif_rx**). As alterações nestas rotinas apenas verificam se a ferramenta está ativada e se existem regras de transmissão ou recepção, chamando as rotinas originais diretamente caso alguma condição não seja satisfeita. Caso as condições sejam satisfeitas, a ferramenta é ativada e as rotinas de injeção de falhas são chamadas.

A ferramenta foi projetada para permitir a seleção e manipulação de pacotes de rede, por meio de arquivos que são utilizados na definição de regras para a injeção de falhas - bem como para o seu controle e uso. Estes são quatro arquivos “virtuais”, que localizam-se no diretório **/proc/net**. São eles:

- **ComFIRM_Log**: somente de leitura, em que a ferramenta disponibiliza as informações sobre seu funcionamento;

- **ComFIRM_Control**: onde são gravados os comandos para inicializar, terminar e reinicializar a ferramenta;
- **ComFIRM_TX_RULES**: contém as regras para injeção de falhas na transmissão dos pacotes; e
- **ComFirm_RX_RULES**: contém as regras para injeção de falhas na recepção dos pacotes.

As regras para injeção de falhas são formadas pela concatenação de instruções, que são interpretadas pela ferramenta. É necessário que estas instruções sejam codificadas em *bytes*, já que a ferramenta situa-se dentro do *kernel* do sistema operacional e a sua interpretação deve ser a mais rápida possível.

Existe um número considerável de instruções que, basicamente, estão agrupadas em dois grandes grupos:

- instruções de testes, que verificam se é o momento oportuno para a injeção de falhas; e
- instruções de ação, que realizam a injeção de falhas propriamente dita.

A ferramenta permite o retardo, descarte, duplicação ou a alteração do conteúdo dos pacotes transmitidos ou recebidos pelo subsistema de rede.

Por restringir-se apenas a falhas de comunicação, o ComFIRM não é adequado para a validação de sistemas de arquivos baseados em *journaling*. Entretanto, o interesse com relação a ele deve-se ao fato de o trabalho também haver utilizado o sistema operacional Linux como plataforma.

2.6.2 FIDe

O FIDe (*Fault Injection via Debugging*) [GON 2001], ferramenta desenvolvida no PPGC da UFRGS, é baseado na depuração de programas pela sua execução até uma chamada de sistema (*syscall)*. O FIDe leva em consideração que praticamente todo o acesso a recursos do sistema operacional e o acesso ao *hardware,* no Linux, dá-se via chamadas de sistema.

O FIDe utiliza a chamada de sistema *ptrace*, que permite executar um processo de três diferentes maneiras: passo-a-passo, usando *breakpoints* ou executá-lo até a próxima chamada de sistema. As primeiras duas maneiras necessitam a modificação do código-fonte dos programas, que na maioria das vezes não está disponível. Baseado nestas premissas, utiliza-se da terceira maneira.

Esta ferramenta trabalha com um arquivo de configuração com as regras que serão usadas durante a execução da aplicação alvo, definindo um cenário de falhas. Um cenário de falhas pode conter muitas regras, as quais são definidas pela sintaxe que segue na Figura 2.3 [GON 2001]:

```
main_rule
        rule_when
                rule [reg | memo | user | param]
                rule [reg | memo | user | param]
                ...
        rule_when
                rule [reg | memo | usr | param]
                ...
```

Figura 2.3: Sintaxe das regras utilizadas pelo FIDe

A regra **main_rule** define qual *syscall* a ferramenta deve interceptar. Quando esta regra é satisfeita, ou seja, quando ocorre a chamada de sistema definida pela regra, o FIDe avalia a regra **rule_when** e determina se já é possível, pela avaliação dos temporizadores ou contadores, a injeção de falhas. Múltiplas regras **rule_when** podem ser definidas dentro de uma **main_rule**.

A regra **rule_when** pode ser baseada em tempo, fluxo ou pela ocorrência de determinados valores nos registradores de memória. Este tipo de regra também determina se a injeção ocorrerá na chamada ou no retorno da *syscall*.

Se uma regra **rule_when** é satisfeita e a ferramenta está pronta para injetar falhas, existem quatro possibilidades de ações que podem ser tomadas:

1. **rule_reg:** trocar o valor dos registradores;
2. **rule_memo:** trocar o conteúdo da memória;
3. **rule_param:** trocar o conteúdo da memória apontado pelos registradores mais um deslocamento; e
4. **rule_user:** trocar dados do *user struct* de um processo, tais como as informações do próprio processo, do seu ambiente ou da sua execução.

As regras de trocas podem ser por incremento, decremento, atribuição e os operadores matemáticos mais (+) e menos (-). A Figura 2.4 mostra a arquitetura do FIDe [GON 2001].

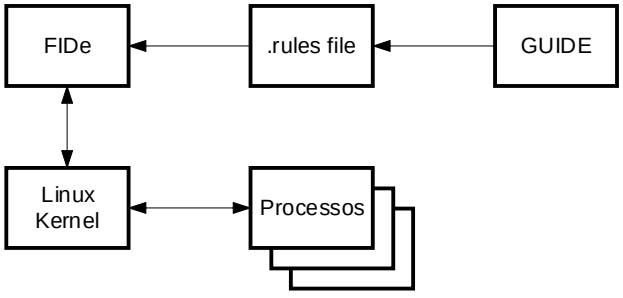

Figura 2.4: Arquitetura do FIDe

Inicialmente cria-se o arquivo de texto com as regras **main_rules,** que contém o conjunto de falhas a serem injetadas. Posteriormente ativa-se a ferramenta da seguinte forma: **fide –d –f <processo que deseja-se injetar falhas>**, onde o parâmetro **–d** faz a ferramenta gravar um arquivo com o *log* de suas ações e o **–f** força a ferramenta a

também gravar no seu *log* todas a aberturas de arquivos realizadas pelo processo no qual deseja-se injetar falhas.

Para este trabalho, inicialmente considerou-se utilizar o FIDe para fazer a injeção de falhas no sistema de arquivos baseado em *journaling*, mas isto não foi possível. Isso aconteceu porque o FIDe foi desenvolvido para funcionar com o *kernel* 2.2 do Linux, e o XFS exige no mínimo o *kernel* 2.4.

Ao invés de adaptar toda a ferramenta FIDe para o novo *kernel* e depois verificar se ela era adequada para a realização de experimentos com sistemas de arquivos, pareceu-nos ser mais conveniente concentrar esforços em uma estratégia específica para validação de sistemas de arquivos baseados em *journaling*.

2.6.3 FAU Machine

É um simulador de ambiente Linux, desenvolvido pelo Departamento de Ciência da Computação da Universidade Erlangen-Nürnberg. O FAUMachine é uma máquina virtual [SIE 2002], similar em muitos aspectos àquelas que são criadas pelos *softwares VMWare* e *Virtual PC*. A máquina virtual FAUMachine é executada como um processo normal de usuário (sem a necessidade de receber privilégios de superusuário, ou o carregamento de módulos de *kernel*), utilizando o *hardware* de uma máquina Linux.

A máquina virtual FAUMachine é carregada usando um *bootloader* adaptado, e executa um *kernel* do Linux ligeiramente modificado. A camada de *hardware* de uma máquina virtual FAUMachine é a da máquina Linux em que o programa está sendo executado. Por meio da FAUMachine, é possível criar uma máquina virtual contendo CPU, disco, CD-ROM e memória pré-alocados.

Nesta máquina virtual, pode ser feita a instalação de uma distribuição Linux pré-configurada – sendo possível escolher entre Debian, Suse ou RedHat. A instalação é feita usando-se os programas de instalação que vêm junto com estas distribuições. Os arquivos binários compilados para o *hardware* i386 podem ser executados na FAUMachine. É permitido o acesso à rede, fazendo com que a FAUMachine possa conectar-se de modo transparente à rede local na qual a máquina hospedeira esteja.

A máquina virtual FAUMachine é usada como plataforma de experimento no *European Dbench Project*, que pesquisa o *benchmarking* de dependabilidade para sistemas de informação [DEP 2003]. Para este propósito, a FAUMachine é distribuída junto com o controlador de experimentos *Expect*. O *Expect* pode realizar um experimento completo a partir de *scripts*, injetando falhas no *hardware* virtual da FAUMachine [HOX 2002].

Para este trabalho tentou-se utilizar a ferramenta FAU Machine, mas isso não foi possível. Isto aconteceu porque, como foi dito, cada máquina virtual apenas pode trabalhar com um conjunto pré-estabelecido de distribuições Linux (Debian, RedHat ou Suse), e ela interpretou o disco de instalação do XFS (que é fornecido pela SGI) como sendo uma nova e desconhecida distribuição Linux.

2.6.4 Orchestra

ORCHESTRA [DAW 96a] [DAW 96b] consiste num ambiente de injeção de falhas para teste de tolerância a falhas em sistemas de tempo-real distribuídos. Ele constitui um *framework* baseado em *scripts* para: sondagem e injeção de falhas, e avaliação/validação de tolerância a falhas em protocolos distribuídos. Foi originalmente

desenvolvido para o sistema operacional de tempo real Mach, e posteriormente foi portado para os sistemas operacionais SunOS e Solaris, da *Sun Microsystems*.

ORCHESTRA visa primordialmente ser uma ferramenta para teste de aplicações distribuídas e protocolos de comunicações, portável para diferentes plataformas injetando falhas na pilha do protocolo. Um aspecto importante de sua arquitetura é a clara separação entre o mecanismo de injeção de falhas no protocolo alvo e o código dependente da plataforma.

O módulo de injeção de falhas é dividido entre as partes que dependem e independem do protocolo. A parte independente do protocolo consiste da parte que realmente injeta falhas e suas estruturas de dados. Também inclui a *interface* do usuário para geração dos *scripts* de injeção de falhas. Estes *scripts* podem ser especificados através da linguagem Tcl ou via diagramas de transição de estado.

A parte dependente do protocolo consiste de código "grudado" na pilha do protocolo de comunicação. Este código implementa a *interface* entre o mecanismo de injeção de falhas e as camadas do protocolo, possuindo também rotinas especiais para troca de mensagens do mecanismo.

A injeção de falhas é feita no nível das mensagens do protocolo. As mensagens que são enviadas e recebidas pelo protocolo alvo são interceptadas, manipuladas ou, ainda, são acrescidas mensagens aleatórias injetadas no sistema (Figura 2.5).

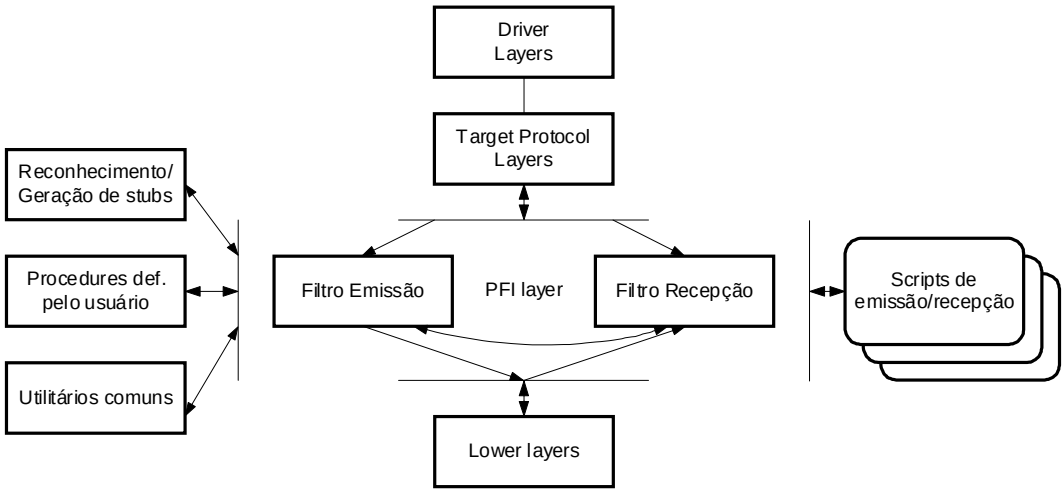

Figura 2.5: Pilha do protocolo com camada de PFI

A Figura 2.5 mostra que a camada de injeção de falhas (ou *PFI Layer*) é inserida entre duas camadas da pilha do protocolo de comunicação. A PFI pode interceptar mensagens que passam e manipulá-las. Por exemplo, a camada de PFI pode inserir mensagens específicas, atrasar uma mensagem por um intervalo de tempo pré-determinado, retransmitir uma mensagem, modificá-la no conteúdo ou, ainda, excluí-la.

Basicamente, a camada de PFI roda dois *scripts:* o **filtro de emissão**, que é executado toda vez que uma mensagem é emitida pelo protocolo, e o **filtro de recepção,** que é ativado sempre que uma mensagem é recebida. Estes *scripts* realizam três tipos de operações sobre as mensagens que trafegam pela pilha do protocolo:

1. Filtragem de Mensagens - para interceptar e examinar uma mensagem;
2. Manipulação de Mensagens - para reter, retardar, reordenar, duplicar ou modificar uma mensagem; e

3. Injeção de Mensagens - para que novas mensagens sejam introduzidas no sistema.

O pacote de **reconhecimento/geração de *stubs*** é invocado para determinar o tipo de mensagem sempre que ela é interceptada pelo PFI. Uma *interface* entre o ***script* de emissão/recepção** para o **reconhecimento/geração de *stubs*** existe para determinar quais tipos de pacotes serão tratados. E ainda, o ***script* de emissão/recepção** pode usar o pacote de **reconhecimento/geração de *stubs*** para gerar certos tipos de mensagens na camada de PFI.

Um aspecto importante considerado por ORCHESTRA é o de procurar não intervir muito sobre o sistema alvo, evitando ao máximo o "efeito Heisenberg" em que ferramentas de injeção de falhas podem ocasionar *overhead* do sistema, principalmente em sistemas de tempo-real. O mecanismo usado por ORCHESTRA é a alocação de recursos extras de CPU para as atividades de comunicação, já que o *overhead* seria sobre os sistemas de comunicação.

ORCHESTRA tem sido utilizado para testar mecanismos de tolerância a falhas em muitos sistemas comerciais e de pesquisa, incluindo as seis principais implementações de TCP existentes, protocolos de comunicação de grupo e sistemas de áudio-conferência em tempo-real.

Novamente, visando especificamente injetar falhas de comunicação, ORCHESTRA não poderia ser utilizado para validação de sistemas de arquivos baseados em *journaling*. Entretanto, ORCHESTRA não pode deixar de ser considerado por ser uma ferramenta clássica para injeção de falhas.

2.6.5 Xception

Xception [CAR 95b] é um *software* de injeção e monitoração de falhas. O Xception usa basicamente depuradores e monitores de desempenho existentes nos modernos processadores para injetar falhas nos sistemas alvo e monitorar o impacto das mesmas. Falhas são injetadas com uma mínima interferência na aplicação alvo - esta aplicação não sofre modificações. Não são inseridos *traps* de *software* e nem a aplicação precisa ser executada em modo *trace*.

O Xception fornece um conjunto de falhas que podem ser ativadas, incluindo falhas espaciais e temporais. Pode-se injetar falhas em qualquer processo em execução no sistema alvo (inclusive no sistema operacional) e o conjunto de falhas pode ser definido pelo usuário. O Xception inicialmente foi implementado sobre uma máquina paralela, utilizando processador PowerPC 601 (IBM/Motorola) e sistema operacional PARIX (versão paralela do Unix). Atualmente, foi portada para outros processadores (SPARC, PowerPc 604, Pentium, etc) e outros sistemas operacionais (Solaris, AIX e Windows NT).

O Xception está dividido em três módulos:

1. Um módulo injetor, que está *linkado* com o sistema alvo;
2. Uma biblioteca com funções para serem chamadas pelo usuário da aplicação, para iniciar a injeção de falhas;
3. Um módulo principal, que roda no sistema *host* e implementa a *interface* com usuário para definição de falhas, injeção automática de falhas e coleta de resultados.

A estrutura do Xception é a que está mostrada na Figura 2.6.

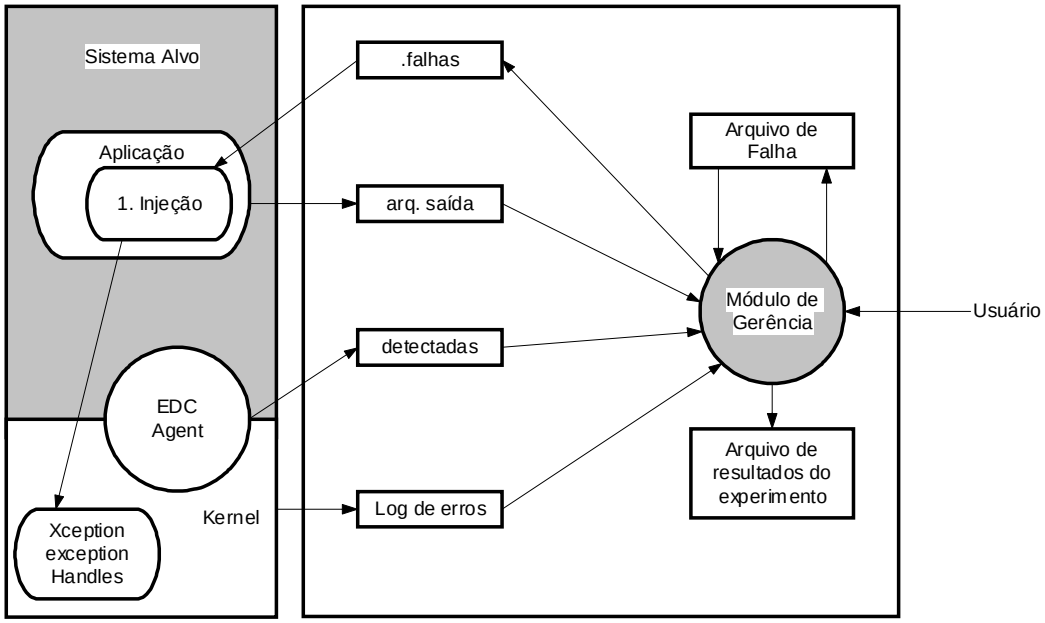

Figura 2.6: Estrutura do Xception

A injeção de falhas pode ser iniciada por qualquer processo que chame a função *StartXception (*presente na biblioteca de funções da ferramenta). A maior parte do código do Xception roda na parte *host* e a porção que roda no sistema alvo é mínima, não necessitando de modificações.

O Xception prevê um completo conjunto de falhas que podem ser ativadas (incluindo as espaciais e temporais), ou ainda, ativadas pela manipulação de dados na memória. Após a injeção de cada falha, o Xception coleta os resultados e adiciona no arquivo de saída, que pode ser analisado posteriormente.

Erros no nível de aplicação, como “acesso ilegal” ou ainda “instrução ilegal”, podem ser detectados pelo sistema embutido no mecanismo de detecção de erro. Ainda, o *kernel* gera mensagens de erros para o *host* que é a origem da ocorrência do erro. Essas mensagens, e também as do estado da injeção enviada pelo módulo Xception no nível do *kernel*, são gravadas no arquivo de *log* de erros - como é mostrado na Figura 2.6. Mais tarde, as mensagens são salvas pelo Xception no arquivo de resultados do experimento.

As mensagens do *status* da injeção são enviadas para o *host* e gravadas no arquivo de *log* imediatamente antes da falha ser injetada - isto é, antes que o processo de exceção retorne o controle para a aplicação, com dados errados. Esta mensagem contém informações de tempo, para permitir o cálculo da latência na detecção do erro pelo *host.*

A principal contribuição desta ferramenta é a possibilidade de injeção de falhas em qualquer processo em execução no sistema alvo, incluindo o sistema operacional, o que possibilita a injeção de falhas em aplicações em que o código-fonte não esteja disponível.

Entretanto, o Xception não pode ser utilizado para este trabalho porque atualmente ele ainda não está disponível para a plataforma Linux [XCE 2003]. Mas devido ao fato de ele ser uma ferramenta clássica para injeção de falhas, ele não poderia deixar de ser mencionado neste trabalho.

2.6.6 FIJI

A ferramenta FIJI (*Fault Injector for Journaling fIlesystems*), desenvolvida nesta dissertação, é baseada na depuração de programas pela sua execução até uma chamada de sistema – usando os mesmos recursos do sistema operacional Linux empregados no desenvolvimento da ferramenta FIDe (Seção 2.6.2). A ferramenta FIJI tem algumas de suas características relacionadas logo a seguir, e encontra-se descrita mais detalhadamente no capítulo 6 desta dissertação.

O FIJI é implementado como sendo um processo concorrente à aplicação alvo, usando os recursos de depuração oferecidos pelo sistema operacional Linux através da chamada de sistema *ptrace*. Conceitualmente, o FIJI localiza-se entre a aplicação alvo e o sistema operacional.

A principal diferença entre o FIJI e o FIDe é o fato de o FIDe ser mais "genérico", visto que permite definir em um arquivo de *script* quais as chamadas que devem ser interceptadas. Este arquivo de *script* contém uma descrição completa do cenário de falhas. Enquanto isso, o FIJI apenas faz a injeção de um subconjunto de falhas que são de interesse específico para sistemas de arquivos baseados em *journaling*. As falhas injetadas pelo FIJI estão de acordo com o modelo de falhas descrito no capítulo 6, e encontram-se especificadas dentro do próprio código-fonte da ferramenta.

As principais razões para o não-reaproveitamento do FIDe para experimentos com o XFS são os fatos de que o FIDe foi desenvolvido com base em recursos que são específicos ao *kernel* do sistema operacional Linux na versão 2.2, enquanto o XFS exige o uso de um *kernel* na versão 2.4 ou posterior. Para que o FIDe pudesse ser utilizado com o XFS seria necessário que fossem feitas várias adaptações – o que apresentou-se como sendo uma tarefa inviável. Sendo assim, foi preferível desenvolver uma nova ferramenta.

Por trabalhar em um nível de abstração razoavelmente mais elevado, o FIJI permite que sejam feitos experimentos para injeção de falhas de modo absolutamente independente do sistema de arquivos em uso na máquina Linux. Ou seja, potencialmente esta mesma ferramenta poderia ser utilizada para validar qualquer outro sistema de arquivos Linux.

Para a realização de experimentos de injeção de falhas com o FIJI, deve ser utilizada em conjunto alguma ferramenta que faça a geração da carga de trabalho. Ao longo desta dissertação, procurou-se gerar diferentes cargas de trabalho a fim de que fossem feitos testes com vários tipos de *workloads* (tais como um único arquivo grande, vários arquivos pequenos, e assim por diante).

2.7 Resumo

Este capítulo apresentou uma visão global sobre injeção de falhas, e relacionou algumas características das ferramentas que foram consideradas para elaboração desta dissertação. Na medida do possível, procurou-se destacar como determinados aspectos em cada uma delas influenciaram no desenvolvimento do FIJI.

3 SISTEMA DE ARQUIVOS LINUX

Este capítulo trata sobre sistemas de arquivos Linux e apresenta as principais estruturas nas quais eles são organizados. Estes fundamentos são empregados tanto pelo XFS (que é o sistema alvo desta dissertação) quanto por outros sistemas de arquivos.

O leitor familiarizado com o assunto não encontrará nenhuma contribuição nova sobre o tema.

3.1 Tipos de dispositivos

Os dispositivos de entrada e saída podem ser divididos, a grosso modo, em duas grandes categorias: dispositivos de bloco e dispositivos de caractere. Os dispositivos de bloco operam como se armazenassem as informações em um conjunto de blocos de tamanho fixo (algo entre 512 e 32.768 *bytes*), cada um com o seu próprio endereço. A propriedade essencial de um dispositivo de bloco é permitir ler ou gravar cada bloco de maneira independente de todos os demais. As unidades de disco, disquetes e CDROM são exemplos para dispositivos de bloco.

Já uma outra categoria é a dos dispositivos de caractere. Nela, as informações são tratadas como sendo apenas fluxos de caracteres - mas sem qualquer tipo de organização. Os dados não são endereçáveis, nem tampouco dispõem de qualquer operação para busca. Impressora, *interface* de rede e mouse são exemplos de dispositivos com esta organização. Em verdade, a grande maioria dos dispositivos que não sejam unidades de disco são construídas com base neste modelo [TAN 97].

Obviamente, este esquema de classificação não é perfeito, pois em um computador existem dispositivos que não podem ser enquadrados em nenhuma dessas categorias. Os relógios internos são um exemplo disto, pois não são endereçáveis como bloco e nem tampouco aceitam ou geram fluxos de caracteres: tudo o que fazem é gerar interrupções em intervalos bem definidos [TAN 97].

No sistema operacional Linux, todas as tarefas relacionadas com a manipulação de dispositivos - sejam estes orientados a bloco ou a caracter - são realizadas pelo *kernel* através de arquivos especiais. Os arquivos especiais são relacionados no diretório /dev do sistema de arquivos, sem consumir espaço algum. Em verdade, as estruturas relacionadas em /dev funcionam apenas como *links* para os *drivers* de dispositivo apropriados [BAR 2000].

3.2 Sistemas de arquivos - *filesystems*

A finalidade dos sistemas de arquivos (ou *filesystem*s) é a de oferecer um nível de abstração para que os processos consigam lidar com os dados contidos em dispositivos.

Sua função é fornecer uma *interface* entre as aplicações e as estruturas que mantém os dados nos meios de armazenamento (tais como *inodes*, arquivos, diretórios e *links*). Sistemas de arquivos não mantém nenhuma informação sobre os dados - eles apenas servem como um mecanismo para representar dispositivos de *hardware* e, eventualmente, comunicação entre processos.

Sistemas de arquivos são implementados como sendo um conjunto de objetos, acessíveis através de uma série de métodos que manipulam determinadas estruturas que são compartilhadas com o *kernel*. Quando um processo tenta abrir um arquivo, o *kernel* deve encarregar-se de redirecionar as instruções para o *driver* de dispositivo adequado.

Segundo Moshe Bar [BAR 2000], um sistema de arquivos no Linux deve ser capaz de permitir as seguintes funcionalidades junto a um dispositivo de bloco ao qual possa eventualmente estar associado:

- Criar e apagar arquivos;
- Abrir arquivos para leitura e escrita;
- Procura dentro de arquivos (apesar de o Linux não lidar com arquivos estruturados como se fossem registros);
- Fechar arquivos;
- Criar diretórios, contendo grupos de arquivos;
- Listar o conteúdo de diretórios; e
- Remover arquivos de diretórios.

Cada partição em um dispositivo de armazenamento pode ter o seu próprio sistema de arquivos, sendo que para o sistema operacional Linux existe uma ampla variedade deles. Alguns exemplos de sistemas de arquivos disponíveis atualmente para o ambiente Linux são os da Tabela 3.1 [BAR 2000]:

Tabela 3.1: Exemplos de sistemas de arquivos disponíveis para o Linux

Sistema de arquivos	Descrição
Ext2	Nativo para o sistema operacional Linux
Ufs	Nativo para o sistema operacional BSD
FAT-32	Nativo para os sistemas operacionais Win9x
NTFS	Nativo para o sistema operacional Windows 2000/NT
NFS	Compartilhamento de arquivos em redes Unix
Smbfs	Compartilhamento de arquivos em redes Microsoft
Ncpfs	Compartilhamento de arquivos em redes Netware

3.3 Metadados

Metadados são estruturas que devem ser mantidas pelos sistemas de arquivos, sem a necessidade de estarem acessíveis através de suas *interface*s. Para um arquivo, por exemplo, metadados podem ser ponteiros indicando quais blocos de dados em uma unidade de disco devem ser unidos para formá-lo. Outros exemplo de metadados são *inodes* e mapas com indicações de blocos livres.

Metadados de um arquivo incluem [TRA 2003]:

- datas (de criação, último acesso e modificação);
- posse;
- permissões de acesso;

- informações de segurança, tais como listas de controle de acesso (ou ACLs);
- tamanho do arquivo; e
- local ou locais de armazenamento no disco.

As informações contidas nos metadados podem ser utilizadas em tarefas relacionadas com a organização do próprio sistema de arquivos, tais como:

- Procurar por todos os blocos de dados para arquivos e diretórios;
- Manter o controle de espaço disponível; e
- Procurar por *inodes* livres.

3.4 Linux Virtual Filesystem Switch - vfs

Entre o *kernel* do sistema operacional Linux e os sistemas de arquivos, existe uma camada de nome *Virtual Filesystem Switch* (ou VFS). Ela é similar à *interface* vnode/vfs encontrada nas variantes do Unix SVR-4, e originalmente é baseada nas implementações que foram feitas para os sistemas BSD e Sun [BAR 2000].

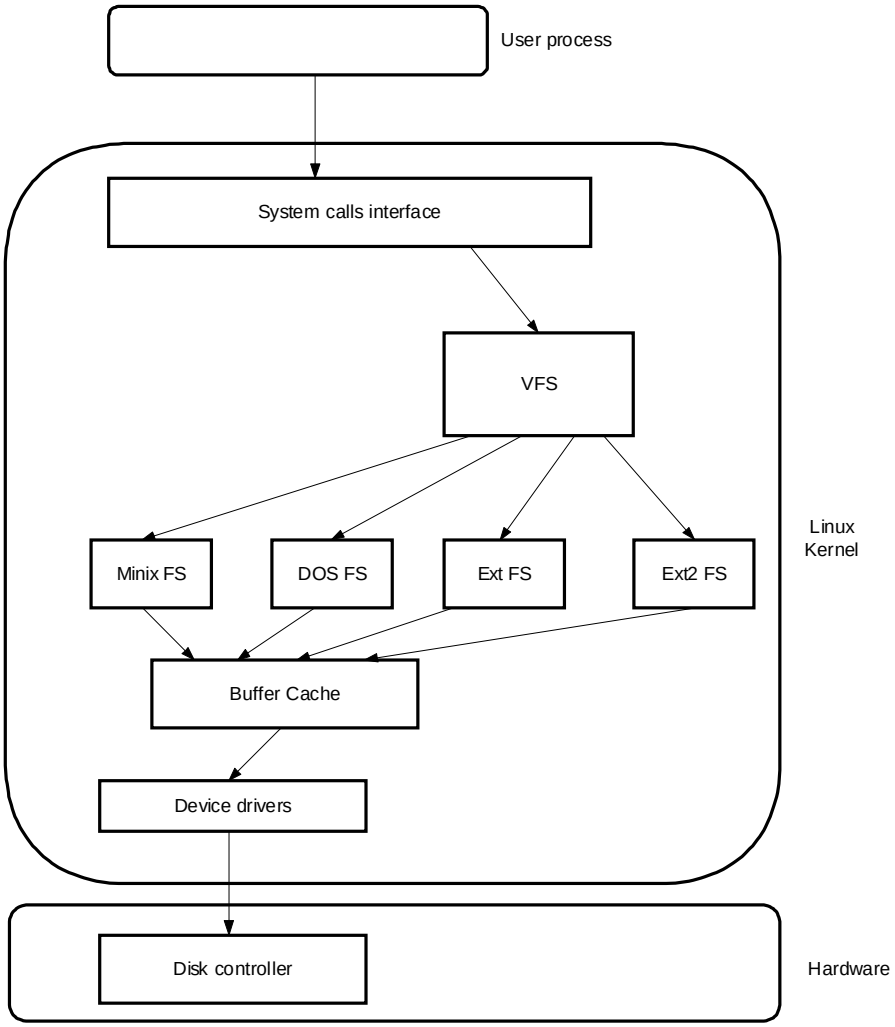

Figura 3.1: VFS com vários sistemas de arquivos no Linux

Por meio desta *interface*, o *kernel* do sistema operacional Linux consegue acessar as informações contidas em dispositivos de armazenamento como sendo uma série de endereços "lineares" para *inodes*. Uma representação da maneira como o VFS interage com vários sistemas de arquivos no sistema operacional Linux é a da Figura 3.1 [CAR 95]:

Os sistemas de arquivos baseados em *journaling* situam-se no mesmo nível do que os sistemas de arquivos Minix e Ext2 na Figura 3.1. O VFS encarrega-se de fazer os ajustes necessários nas estruturas de dados para que sistemas de arquivos diferentes (tais como NFS, ext2, ReiserFS ou mesmo procfs) sejam todos visualizadas como sendo um mesmo conjunto de objetos, acessíveis por métodos bem definidos [BAR 2000].

Sempre que é necessário executar alguma chamada de sistema relacionada com sistemas de arquivos, o VFS fica encarregado de realizar o tratamento adequado. O VFS tem a função de atender as chamadas à essas rotinas, manipulando certas estruturas de dados para que as ações apropriadas venham a ser executadas [CAR 95].

Para conseguir este resultado, a especificação do VFS define uma funcionalidade mínima que deve ser implementada em todo sistema de arquivos Linux - a fim de tornar possível a criação de uma *interface* entre eles e o *kernel* do sistema operacional. Segundo Stephen Tweedie [CAR 95], o conjunto de operações oferecido pelas funções de um sistema de arquivos deve ser capaz de lidar com os objetos *filesystem*, *inode* e *open file*.

3.4.1 Estrutura *file_system_type*

Primeiramente, é preciso que o VFS tenha conhecimento de quais sistemas de arquivos são admitidos pelo sistema operacional Linux na máquina em questão. Para isso, o VFS utiliza uma tabela que é definida durante a configuração do *kernel*. Cada entrada nesta tabela descreve um sistema de arquivos diferente, contendo o seu nome e o ponteiro para uma função que deve ser chamada durante as operações de montagem para sistemas de arquivos deste tipo [MOS 2000].

O trecho de código na Figura 3.2 foi extraído do código-fonte para o *kernel* do Linux, e apresenta uma definição da estrutura que representa um sistema de arquivos na tabela que é criada durante a configuração do *kernel*. Ela encontra-se no arquivo /usr/src/linux/include/linux/fs.h [MOS 2000]:

```
struct file_system_type {
        const char *name;
        int fs_flags;
        struct super_block *(*read_super) (struct super_block *, void *, int);
        struct module *owner;
        struct vfsmount *kern_mnt;
        struct file_system_type * next;
};
```

Figura 3.2: Estrutura que define um *file_system_type*

Na Figura 3.2, o item *name* representa o nome do sistema de arquivos – tal como ext2, nfs ou xfs, por exemplo. Uma lista completa contendo os nomes dos sistemas de arquivos admitidos por uma máquina Linux pode ser obtida no arquivo

/proc/filesystems. Adicionalmente, outros nomes podem vir a ser adicionados a nessa lista se sistemas de arquivos forem carregados em memória na forma de módulos.

O item *fs_flags* descreve o tipo de sistema de arquivos que esta estrutura representa. Para este item, o *flag* mais comumente utilizado é o FS_REQUIRES_DEV - que informa ao Linux que sistema de arquivos precisa ser montado como um dispositivo de bloco. Sistemas de arquivos em rede (tal como o NFS, por exemplo) não possuem este *flag* FS_REQUIRES_DEV habilitado [MOS 2000].

Eventualmente, o item *fs_flags* também pode ter o valor FS_SINGLE - para sistemas de arquivos que podem ter apenas um superbloco. Além disso, o item *fs_flags* também pode conter o valor FS_NOMOUNT (para sistemas de arquivos que não devem ser montados por meio do comando *mount*, tal como acontece com o *pipefs)*.

E por fim, o item *read super function* indica qual a função que deve ser chamada para montar o sistema de arquivos deste tipo. A indicação de um nome para esta função é obrigatória.

3.4.2 Estrutura *super_block*

Cada partição em um dispositivo de armazenamento possui uma estrutura chamada de *superbloco*. O superbloco contém uma descrição do tamanho e do formato do sistema de arquivos, sendo necessário para a sua gerência, uso e manutenção.

Normalmente, apenas o superbloco do grupo 0 de um dispositivo é utilizado. Porém, por questões de recuperação em caso de falhas, cada grupo de blocos pode manter uma cópia do superbloco – tal como acontece no ext2, por exemplo [CAR 95].

Entre as informações que são mantidas pelo superbloco, estão [MOS 2000]:

- um identificador do sistema de arquivos (número mágico);
- quantidade de vezes que este sistema de arquivos já foi montado;
- número do grupo de bloco que contém essa cópia do superbloco;
- tamanho do bloco;
- a quantidade de blocos por grupo;
- número de blocos livres no sistema;
- número de *inodes* livres; e
- um apontador para o primeiro *inode* do sistema de arquivos (o diretório "/").

As operações que podem ser realizadas com o superbloco no Linux são bastante diferentes daquelas que são definidas para o superbloco nos sistemas Unix SVR-4, pois incluem funções para leitura e escrita de *inodes* [MOS 2000].

3.4.3 Estrutura vfsmount

Como resultado de uma operação de montagem, sempre haverá o retorno de um descritor. Com isso, o VFS poderá acessar as rotinas do sistema de arquivos em questão - através deste descritor para o sistema de arquivos montado [CAR 95].

O descritor para sistema de arquivos montado pode conter vários tipos de informação: desde dados que são comuns para todos os tipos de sistemas de arquivos, até ponteiros para as funções e dados que sejam privativos a este sistema de arquivos. É

justamente através desses vários ponteiros, contidos no descritor para o sistema de arquivos montado, que o VFS pode acessar às rotinas deste sistema de arquivos.

No Linux, informações sobre os sistemas de arquivos montados são mantidas em duas estruturas separadas: o superbloco (descrito anteriormente) e o descritor para sistema de arquivos montado. A razão para isso é que o Linux permite a montagem de um mesmo sistema de arquivos (dispositivo de bloco) sob vários pontos de montagem, o que significa que um mesmo superbloco pode corresponder a múltiplos descritores para sistema de arquivos montados.

3.4.4 Estrutura dentry

O VFS possui um cache de diretórios, chamado de *dcache*, que é formado por um conjunto de estruturas *dentry*. O *dcache* é um *cache* tanto de diretórios quanto de *inodes* contidos dentro destes diretórios. Cabe salientar que a *dcache* não armazena o conteúdo dos *inodes* de um diretório – ela apenas faz um mapeamento entre nomes de arquivos e número de *inode* [MOS 2000].

A função do *dcache* é manter em memória os diretórios que são buscados com maior freqüência, para acelerar as operações de manipulação desses diretórios. Quando o VFS necessita encontrar um arquivo em um diretório, ele primeiramente faz uma consulta no *dcache*. Se ele encontra a informação procurada, ela é utilizada. Mas se a entrada de diretório não existe no *dcache*, é feita uma chamada para as rotinas de leitura no dispositivo de armazenamento. Assim que este novo *inode* for acessado, ele será adicionado ao *dcache* [BAR 2000].

3.4.5 Estrutura *inode*

Todo arquivo e diretório em um sistema de arquivos Linux é representado por um, e apenas um, *inode*. Um *inode* é um registro que armazena a maior parte das informações sobre um arquivo específico em disco. O nome *inode* deriva de "*index node*" (nó de índice) [BAR 2000]. A estrutura de um inode segue na Figura 3.3:

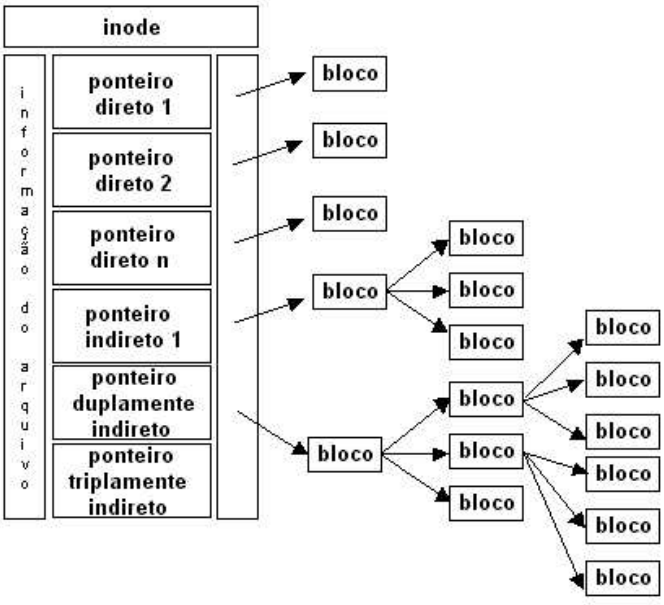

Figura 3.3: Estrutura para endereçamento em um *inode*

Um *inode* contém todas as informações sobre um arquivo, exceto o seu nome – o qual está armazenado no diretório, junto com o número do *inode*. O *inode* contém os identificadores de usuário e grupo do arquivo, as datas de última modificação e último acesso, um contador do número de *hard links* (entradas de diretório) ao arquivo, e o tipo de arquivo (arquivo simples, diretório, *link* simbólico, dispositivo de caracteres, dispositivo de blocos ou *socket*).

Na Figura 3.3, o *inode* contém 15 ponteiros para os blocos que mantêm os dados do arquivo. Os primeiros 12 desses ponteiros apontam para blocos diretos: ou seja, eles contêm endereços de blocos contendo dados do arquivo. Assim, os dados de arquivos pequenos (não mais do que 12 blocos) podem ser referenciados imediatamente, porque uma cópia do *inode* é mantida na memória principal enquanto o arquivo estiver aberto.

O próximos três ponteiros do *inode* apontam para blocos indiretos. O primeiro ponteiro de bloco indireto é o endereço de um bloco indireto simples. É um bloco de índice, que não contém dados, mas os endereços dos blocos que efetivamente contêm dados. O segundo é um ponteiro de bloco indireto duplo, o endereço de um bloco que contém os endereços dos blocos que contêm ponteiros para os blocos de dados reais. Por fim, o ponteiro final contém o endereço de um bloco indireto triplo.

3.4.6 Estrutura file

No Linux, considera-se arquivo tudo aquilo com o qual seja possível executar as operações de leitura e escrita. A estrutura *file* contém ponteiros para funções a serem executadas em arquivos abertos (tais como "*read*" e "*write*", por exemplo).

Existem diversos níveis entre o descritor para um arquivo aberto e o seu *inode* correspondente. Quando um processo executa a chamada de sistema *open* para um arquivo, o *kernel* do sistema operacional Linux retorna um número inteiro não-negativo que será usado para todos os acessos subseqüentes a este arquivo. Este número inteiro é um índice para um *array* de estruturas do tipo *file.*

Cada estrutura *file* corresponde a uma entrada para uma estrutura *dentry*. E cada estrutura *dentry*, por sua vez, aponta para um *inode.* A explicação detalhada para os campos que compõem descritores de arquivos abertos é extremamente complexa, e pode ser encontrada na obra de Moshe Bar [BAR2000] e Jim Mostek [MOS 2000].

3.5 Consistência do sistema de arquivos

Todo aplicativo deve ser capaz de armazenar e recuperar arquivos em disco, e a gerência destes arquivos é responsabilidade do sistema operacional. Através do sistema de arquivos, o sistema operacional deve definir o modo como os arquivos serão estruturados, nomeados, acessados, utilizados, protegidos e implementados [TAN 97].

Sob o ponto de vista do usuário, um sistema de arquivos é uma coleção hierárquica de arquivos e diretórios. Mas para o sistema operacional Linux, o sistema de arquivos consiste de:

- *Inodes* que contém informações sobre arquivos e diretórios (também conhecidos como metadados), e
- blocos de dados, que contém as entradas dos diretórios e os conteúdos dos arquivos.

Caso múltiplos *inodes* estivessem referenciando um mesmo bloco de dados, haveria inconsistências. Para evitar isso, as estruturas de dados que definem a organização de um sistema de arquivos devem ser corretas no momento em que o sistema de arquivos for utilizado. Os sistemas de arquivos cujas estruturas de dados internas estão corretas são chamados de consistentes.

É responsabilidade do sistema que armazena um sistema de arquivos verificar a consistência do sistema de arquivos antes de torná-lo disponível através de suas *interfaces*. As características básicas de um sistema de arquivos consistente são:

- O bit s_dirt no superbloco é ajustado para indicar que ele foi desmontado com sucesso no momento em que a máquina foi desligada; e
- Todos os metadados para este sistema de arquivos estão corretos.

Testar a consistência de um sistema de arquivos deve ser simples, se estas duas condições puderem ser verificadas rapidamente. Mas a tarefa de verificar se todos os metadados para um sistema de arquivos estão corretos é algo difícil e consiste em verificar [HAG 2004]:

- Se cada unidade de alocação (seja ela um bloco, ou um *extent*) ou pertence somente a um único arquivo ou diretório, ou está marcada como não-utilizada. A lista de quais unidades estão alocados e livres normalmente é armazenada em um *bitmap* para o sistema de arquivos, onde cada bit representa uma unidade de alocação. Sistemas de arquivos com alocação baseada em *extents* também precisam manter informações sobre tamanho e intervalo dos *extents*, tanto para os utilizados quanto para os livres;
- Nenhum arquivo ou diretório contém uma unidade de alocação marcada como "não-utilizada" no *bitmap* do sistema de arquivos;
- Cada arquivo ou diretório no sistema de arquivos é referenciado em algum outro diretório deste sistema de arquivos. Sob o ponto de vista do usuário, isto significa que sempre existe um caminho para cada arquivo ou diretório do sistema de arquivos; e
- Cada arquivo tem somente o número de diretórios superiores, conforme o indicado no seu *inode*. Embora cada arquivo exista somente em um único lugar físico no disco, vários diretórios podem conter referências ao *inode* que representa este arquivo – e essas referências são chamadas de *hard links*. O arquivo deve poder ser acessado por qualquer uma destas entradas de diretório, e o fato de alguma delas vir a ser excluída significa que o campo *link count* para este *inode* deve ser decrementado. Um arquivo somente deve ser removido do disco se o valor *link count* tornar-se igual a zero – ou seja, quando não houver mais nenhuma entrada de diretório para ele.

Verificar esses relacionamentos pode exigir um tempo demasiado, se for necessário testar cada uma dessas condições. A diferença fundamental entre um sistema de arquivos tradicional (tal como o ext2) e um sistema de arquivos baseado em *journaling* é a necessidade de realizar esses testes.

3.5.1 Verificando a consistência de um sistema de arquivos

A tarefa de verificar a consistência de um sistema de arquivos Unix é feita por um utilitário chamado *fsck* (*Filesystem Consistency Check)* [PIE 2002]. Este utilitário *fsck*, por sua vez, é capaz de executar várias outras versões de utilitários *fsck* – uma para cada sistema de arquivos diferente, e que compreende a organização para sistemas de arquivos deste tipo. No Linux, o padrão é que todas essas versões de *fsck* estejam no diretório /sbin e tenham nomes no formato *fsck.<tipo-de-sistema-de-arquivos>*.

A Figura 3.4 é uma lista com exemplos de diferentes versões do utilitário *fsck*, encontradas em uma máquina Linux típica. Ela foi obtido em um equipamento usando a distribuição Red Hat Linux 7.2:

```
[root@cygnus sbin]# ls -la *fsck*
-rwxr-xr-x    3 root     root        40508 Jul  6  2001 dosfsck
-rwxr-xr-x    3 root     root       568872 Feb 26  2002 e2fsck
-rwxr-xr-x    1 root     root        22252 Feb 26  2002 fsck
-rwxr-xr-x    3 root     root       568872 Feb 26  2002 fsck.ext2
-rwxr-xr-x    3 root     root       568872 Feb 26  2002 fsck.ext3
-rwxr-xr-x    1 root     root        16588 Jun 24  2002 fsck.minix
-rwxr-xr-x    3 root     root        40508 Jul  6  2001 fsck.msdos
lrwxr-xr-x    1 root     root           10 Mar 19  2002 fsck.reiserfs -> reiserfsck
-rwxr-xr-x    3 root     root        40508 Jul  6  2001 fsck.vfat
-rwxr-xr-x    1 root     root         2584 Nov  7  2001 fsck.xfs
-rwxr-xr-x    1 root     root       186716 Jul 21  2001 reiserfsck
[root@cygnus sbin]#
```

Figura 3.4: Exemplos de utilitários *fsck* para uma máquina Linux

No momento em que uma máquina Linux é reiniciada, o utilitário *fsck* obtém uma lista de todos os sistemas de arquivos que devem ser montados e os seus tipos através do arquivo /etc/fstab. O sexto e último campo de cada linha deste arquivo /etc/fstab indica se o *fsck* deve ou não fazer teste de consistência para este sistema de arquivos. Um valor zero neste campo indica que o sistema de arquivos não deve ser verificado. Qualquer outro valor numérico indica a partir de qual etapa do *fsck* que deve ser feita a verificação para este sistema de arquivos.

Como a execução do *fsck* pode ser demorada em demasia para unidades de disco com grande capacidade, foram feitas otimizações neste utilitário para que fosse possível executá-lo em várias instâncias, de modo concorrente. Entretanto, isto apenas é possível se os sistemas de arquivos referenciados em /etc/fstab estiverem localizados em discos diferentes – caso contrário, a execução do *fsck* é feita em seqüência.

O utilitário *fsck* primeiramente faz o teste para verificar se o superbloco do sistema de arquivos indica que o mesmo foi desmontado com sucesso. Caso este bit do superbloco esteja habilitado, o *fsck* não faz nada e segue para o próximo sistema de arquivos listado em /etc/fstab. Mas caso este bit não esteja habilitado, o utilitário *fsck* executa a tarefa de verificar o sistema de arquivos através da versão apropriada de *fsck* para este tipo de sistema de arquivos.

3.6 Resumo

Este capítulo apresentou as principais estruturas de dados nos quais os sistemas de arquivos Linux são baseados. Uma visão sobre o funcionamento do mesmo, bem como de suas iterações com o VFS é necessário para que se tenha um entendimento sobre o XFS (sistema alvo deste experimento para injeção de falhas).

4 SISTEMAS DE ARQUIVOS BASEADOS EM JOURNALING

Este capítulo apresenta como é a organização para sistemas de arquivos baseados em *journaling*. E mais especificamente, ele descreve como é a organização do sistema de arquivos XFS (o qual é o sistema alvo da injeção de falhas nesta dissertação).

4.1 Introdução

O conceito de *journaling* foi introduzido na área de banco de dados com o propósito de assegurar consistência e integridade na ocorrência de falhas durante transações. Sistemas com *journaling* armazenam cada uma das operações realizadas sobre os seus registros para conseguirem retornar a um último estado consistente, na ocorrência de falhas [GRA 93].

Sistemas de arquivos baseados em *journaling*, por sua vez, usam o mesmo princípio: fazem um controle sobre as mudanças realizadas ou nos metadados (para JFS, ReiserFS e XFS), ou nos dados e metadados (para ext3) associados a um sistema de arquivos. A idéia consiste em tratar de forma diferente os dados e metadados, usando uma área dedicada em disco para manter um histórico das mudanças [PIE 2002].

O registro das alterações entre dados e metadados deve ser feito em disco de forma síncrona, sob pena de virem a ser geradas inconsistências. Entretanto, normalmente estas sincronizações não são feitas com freqüência por razões de desempenho. Sistemas de arquivos baseados em *journaling* implementam uma política chamada de *write-ahead logging*, fazendo com que os registros no *log* sejam armazenados em disco antes de as operações serem efetivamente realizadas [PIE 2002].

Este tipo de sistema de arquivos pode ser empregado para aplicações que lidem com arquivos pequenos e façam uso freqüente da chamada de sistema *sync*, tal como acontece nos servidores de NFS [SEL 92]. Sistemas de arquivos baseados em *journaling* tem um desempenho excelente para realizar operações de escrita com arquivos de tamanho pequeno porque as alterações nos metadados são transferidas para o disco através de uma única operação, que acrescenta várias transações em uma mesma área em disco usada para *log* [SEL 2000].

Uma das grandes vantagens da abordagem baseada em *journaling* é de que ela pode ser facilmente implementada em sistemas de arquivos já existentes, quando estes são baseados na abordagem baseada em blocos. Exemplo disso é o resultado que pôde ser obtido com o sistema de arquivos ext2, baseado na abordagem baseada em blocos; e que pôde ser facilmente reimplementado como sendo o ext3, um sistema de arquivos baseado em *journaling* [TWE 2003].

Quanto aos aspectos de alta disponibilidade, os tempo de recuperação para sistemas de arquivos baseados em *journaling* são bastante reduzidos. A tarefa de verificação do sistema de arquivos consiste apenas em inspecionar as transações pendentes do *log*, ao invés de percorrer todos os blocos de um sistema de arquivos para buscar inconsistências. Com isso, na ocorrência de defeitos, o sistema de arquivos pode ser levado a um estado consistente pela aplicação das transações pendentes no *log* - ao invés de ser necessário inspecionar toda uma unidade de disco com o *fsck* [PIE 2002].

A técnica de *journaling* para sistemas de arquivos com alta disponibilidade baseia-se na redundância para aumentar significativamente a confiabilidade dos dados e metadados, mas sem aumentar significativamente os custos de *hardware*. Ela já é adotada por sistemas de arquivos em sistemas operacionais para plataformas diversas, tais como Solaris, AIX, Digital UNIX, HP-Ux, Irix e Windows NT [SEL 2000]. Para o sistema operacional Linux, existem os sistemas de arquivos ext3, JFS, ReiserFS e XFS.

4.2 Sistemas de arquivos baseados em *journaling*

Sistemas de arquivos baseados em *journaling* mantém um registro sobre todas as mudanças que são feitas no sistema de arquivos em uma parte especial do disco chamada de *log*, ou *journal*. Nesta abordagem, as mudanças sempre são armazenadas no *log* antes de serem efetivamente aplicadas ao sistema de arquivos em questão [PIE 2002]. O *log* é implementado como sendo um *buffer* circular, ocupando geralmente menos de 1% da capacidade da unidade de disco.

Durante a operação normal do sistema, uma *thread* em *idle* encarrega-se de processar as transações registradas no *log* escrevendo os dados para os sistemas de arquivos. Esta *thread* também coloca um *flag* em cada transação completada, para indicar que foi feito um *commit* da mesma. Fazer o *commit* de transações do *log* consiste em estabelecer um novo *checkpoint.* Além disso, a cada vez que a chamada de sistema *sync* é executada, um novo *checkpoint* é estabelecido [TAN 97].

Se um computador vier a sofrer alguma falha enquanto estiver realizando mudanças no sistema de arquivos, o sistema operacional poderá utilizar as informações no *log* para tornar o sistema de arquivos consistente através de um *log replay* - ou seja, aplicação das transações pendentes relacionadas no log. Isto geralmente é feito durante as operações de montagem do sistema de arquivos.

Conforme descrito na seção 3.4.7, logo após o colapso de um computador a integridade dos sistemas de arquivos locais deve ser verificada por meio da realização de exaustivos testes com o *fsck*. Mudanças no sistema de arquivos podem ter sido escritas apenas parcialmente em disco, e o sistema operacional não tem meios para determinar se todas as operações de escrita foram realizadas com sucesso – a não ser por uma verificação completa da integridade do sistema de arquivos.

Este tipo de teste não seria necessário após um *shutdown* ou *restart* normal do sistema, porque estes procedimentos fazem com que seja feito um *flush* de todas as operações de escrita pendentes e o sistema de arquivos é marcado como *clean* no bit s_dirt do superbloco antes de ser desmontado. Sistemas de arquivos marcados como *clean* não são verificados quando uma máquina é reiniciada.

Mas diferentemente dos sistemas de arquivos padrão, os sistemas de arquivos baseados em *journaling* podem ser tornados consistentes pelo *replay* de quaisquer ações do *log* que não estejam marcadas como tendo sido salvas em disco. Essas ações registradas no *log*, por sua vez, podem variar sendo desde o registro de algumas mudanças em metadados até um registro completo de todas as mudanças feitas em um arquivo.

A medida que as unidades de disco e sistemas de arquivos tornam-se cada vez maiores, o tempo necessário para verificar a sua integridade usando *fsck* também cresce; ficando ainda mais evidente a diferença entre os tempos de verificação das abordagens baseadas em bloco e em *journaling*.

Na Tabela 4.1, encontram-se relacionados os sistemas de arquivos baseados em *journaling* para o Linux, a versão de *kernel* a partir da qual eles estão disponíveis, a data em que foram criados, e a data em que tornaram-se disponíveis para a plataforma Linux:

Tabela 4.1: Sistemas de arquivos baseados em *journaling* para o Linux

Sistema de arquivos	Disponível no kernel	Data de criação	Disponível para Linux
ext3	2.4.15	1999	1999
JFS	2.4.20	1999	2000
ReiserFS	2.4.1	1996	1996
XFS	2.5.x	1993	1999

4.2.1 Conteúdo do *log* do *journaling*

Existem duas diferentes abordagens que podem ser utilizadas por um sistema de arquivos baseado em *journaling*, cada qual com as suas vantagens e desvantagens:

- *log* contém apenas o registro das mudanças realizadas sobre metadados do sistema de arquivos, associados com cada operação de escrita; e
- *log* contém o registro das mudanças realizadas sobre dados e metadados do sistema de arquivos, associados com cada operação de escrita.

O denominador comum entre as duas abordagens – de fazer o *log* das mudanças nos metadados – é o que assegura a integridade do sistema de arquivos baseado em *journaling* [PIE 2002]. Mesmo após o colapso do sistema, a estrutura de arquivos, diretórios, e o sistema de arquivos podem ser tornados consistentes pela reexecução de quaisquer operações pendentes que encontrem-se descritas completamente no *log*. As entradas do *log* são tratadas como sendo transações ACID [GRA 93], o que vale dizer que o início e o fim de cada mudança é registrado. Portanto, o conjunto de mudanças feitas em um sistema de arquivos deve ou ser realizado totalmente, ou então descartado.

Por exemplo, assuma-se que tenha sido salva uma nova versão de um arquivo que esteja sendo editado. Isto é algo que faria com que os eventos a seguir acontecessem no sistema de arquivos:

- Novas unidades seriam alocadas para armazenar os novos dados (isto sempre acontece primeiro);
- Essas novas unidades alocadas seriam marcados como sendo “em uso” no *bitmap* do sistema de arquivos;
- O conteúdo do novo arquivo seria escrito para essas novas unidades de alocação;
- *O inode* ou bloco indireto identificando a cadeia de unidades de alocação associada com este arquivo seria atualizado, para incluir as novas unidades de alocação em disco; e
- Os *timestamps* para último acesso e última escrita seriam atualizados.

Além de registrar todos esses eventos, o *log* deve conter informações para indicar que todos eles estão correlacionados. Se o computador sofrer um colapso neste momento, o desejável é que ou todos os eventos sejam efetivados, ou que nenhum deles seja. Isso porque apenas marcar blocos como sendo "em uso", mas não utilizá-los, além de ser incorreto seria um desperdício de espaço em disco.

As ações associadas com cada mudança no sistema de arquivos devem ser atômicas – ou todas elas ocorrem, ou nenhuma delas ocorre. Continuando com este exemplo, um sistema de arquivos baseado em *journaling* que somente faça o registro das mudanças nos metadados contém um registro completo de todas as alterações – com exceção do conteúdo real destas novas unidades de alocação. Para este caso, fazer o *replay* de somente as mudanças atômicas nos metadados poderia assegurar a consistência do sistema de arquivos, mas o arquivo modificado conteria informações sem sentido no seu final – porque as novas informações que deveriam ter sido escritas não foram registradas em *log*.

Sendo assim, para haver uma completa segurança, os logs devem conter não só o registro das mudanças, mas também o conteúdo original dos arquivos - antes de as mudanças serem feitas. Isto torna possível que seja feito um *undo*, habilitando o sistema de arquivos a desfazer mudanças relativas a operações que não foram completadas.

Mas infelizmente, a medida que mais informações são armazenadas no *log*, também aumenta a quantidade de tempo necessária para realizar as escritas – especialmente porque as escritas no *log* devem ser feitas de forma síncrona com as alterações nos dados. Sob o ponto de vista do usuário, isto reduz a chance de que mudanças feitas por ele não estejam visíveis quando o sistema for restabelecido após uma falha. Apesar disso, cada sistema de arquivos possui uma abordagem diferente sobre o que deve ou não constar no log: dados modificados em arquivos, diretórios de dados, metadados, ou mudanças nos metadados.

4.2.2 Local de armazenamento dos logs

A medida que diferentes sistemas de arquivos baseados em *journaling* armazenam diferentes tipos de informações nos seus *logs*, eles também armazenam esses *logs* em locais diferentes [HAG 2004]. Esta seção apresenta uma visão geral sobre vantagens e desvantagens de cada uma das abordagens que são possíveis.

A abordagem mais simples é a de armazenar o *log* na forma de um arquivo real, dentro do próprio sistema de arquivos em que estão sendo feitas mudanças. Esta é a abordagem utilizada pelo ext3 [TWE 98], visto que o objetivo principal desde sistema de arquivos é adicionar as vantagens do *journaling* a um sistema de arquivos já existente (o ext2), enquanto continua mantendo compatibilidade com este.

Armazenar o *journal* como um arquivo dentro do sistema de arquivos causa problemas de desempenho. Primeiramente, porque as escritas no *log* devem ser feitas através das chamadas de sistema padrão para o sistema de arquivos. Além disso, se o sistema de arquivos que contém o *log* vier a ser danificado de alguma forma, é possível perder o *log*. Ainda outro problema com a abordagem de armazenar o *journal* como um arquivo dentro do sistema de arquivos é que a tarefa de manter o controle torna-se mais lenta, visto que o registro de informações no *log* está concorrendo com os movimentos de disco para o registro de alterações nas unidades de alocação do sistema de arquivos.

Uma outra abordagem é a de armazenar o *log* em uma parte especial do sistema de arquivos, não acessível aos programas de usuário. Isto permite que sejam utilizadas chamadas de sistema especializadas para escrita no *log*, otimizadas, aumentando o

desempenho. Além disto, também diminuem substancialmente as chances de que o *log* possa ser perdido – pois ele continua associado ao sistema de arquivos, mas está armazenado em uma região especial ao invés de ser um arquivo. Esta abordagem é permitida pelo XFS.

A última abordagem consiste em armazenar o *log* em uma unidade de disco diferente daquela em que está o sistema de arquivos. Isso elimina o problema de uma corrupção da unidade que seja extensiva aos *logs* e também elimina a concorrência pelos movimentos da cabeça do disco entre o registro de informações nos *logs* e o registro nas alterações nas unidades de alocação. O XFS permite que esta abordagem seja utilizada.

Para determinar qual a abordagem que está sendo empregada para um sistema de arquivos, é possível utilizar o utilitário xfs_info (Figura 4.1)

```
bfw:/usr/src/packages # xfs_info /
meta-data       =/              isize=256    agcount=19, agsize=262144 blks
                =               sectsz=512
data            =               bsize=4096   blocks=4725756, imaxpct=25
                =               sunit=0      swidth=0 blks, unwritten=1
naming          =version 2      bsize=4096
log             =internal       bsize=4096   blocks=2560, version=1
                =               sectsz=512   sunit=0 blks
realtime        =none           extsz=65536  blocks=0, rtextents=0
```

Figura 4.1: Execução do comando xfs_info

4.2.3 Verificação de consistência

Embora normalmente o utilitário *fsck* assegure integridade para sistemas de arquivos, isto não é verdadeiro para sistemas de arquivos baseados em *journaling*. Para os sistemas de arquivos desta abordagem, o programa *fsck* somente faz uma busca no *log* e reexecuta as transações que estejam registradas de forma completa e que ainda não tenham sido aplicadas em disco.

Sistemas de arquivos baseados em *journaling* são automaticamente atualizados no momento em que são montados. Durante a montagem, o trecho de código para o sistema de arquivos existente no *kernel* encarrega-se de fazer o *replay* das transações pendentes.

Conforme foi dito nas seções anteriores, durante o processo de *boot* o sistema operacional Linux assume que deve ser feita uma verificação de consistência para os sistemas de arquivos da máquina. Para isso, ele faz uma busca no arquivo /etc/fstab [HAG 2004]. Logo, recomenda-se que para cada sistema de arquivo baseado em *journaling* o valor da sexta coluna seja marcado como sendo zero.

4.2.4 Vantagens de sistemas de arquivos baseados em *journaling*

Embora muitas das vantagens dos sistemas de arquivos baseados em *journaling* tenham sido discutidas anteriormente, esta seção resume as principais melhorias em termos de desempenho e confiabilidade obtidos com o uso destes. São elas [HAG 2004] [TRA 2003] [TWE 2003]:

- Tempo de *restart* mais rápido após a ocorrência de falhas do tipo colapso, porque o computador não tem a necessidade de examinar cada sistema de arquivos do computador para garantir a consistência das informações;
- Maior flexibilidade, pois sistemas de arquivos baseados em *journaling* geralmente criam e alocam *inodes* de forma dinâmica – ao invés de pré-alocarem um número específico de *inodes*, durante a formatação do sistema de arquivos. Isto remove uma limitação referente ao número de arquivos e diretórios que podem ser criados em cada partição. O fato de não ser necessário gerenciar listas de *inodes* que ainda não tenham sido alocados também reduz o *overhead* associado às manipulações em metadados;
- Acesso rápido à arquivos e diretórios. Todos os sistemas de arquivos baseados em *journaling* utilizam algoritmos mais sofisticados para armazenamento e acesso de arquivos e diretórios do que os sistemas de arquivos tradicionais – como o ext2. Os sistemas de arquivos JFS, ReiserFS e XFS utilizam estruturas de dados avançadas, tais como *B+Trees* (para JFS e XFS) ou *B*Trees* (para ReiserFS) para acelerar a busca e o armazenamento de *inodes*; e
- Escritas no *log* podem ser otimizadas. Os *logs* utilizados pelos sistemas de arquivos baseados em *journaling* podem ser escritos de uma maneira mais rápida até mesmo que o sistema de arquivos: as operações normalmente consistem de anexar registros, ao invés de inseri-los. Normalmente os *logs* são pré-alocados, de tamanho fixo, circulares, e utilizam rotinas otimizadas para leitura e escrita.

4.3 Motivação para escolha do XFS como alvo

A escolha sobre qual sistema de arquivos deveria ser utilizado para a injeção de falhas esteve limitada às opções disponíveis no início deste trabalho, que foram ext3, JFS, ReiserFS e XFS. Com base em estudos e na realização de experimentos, optou-se pelo sistema de arquivos XFS [XFS 2003].

O ext3 tem a grande vantagem de permitir a migração de sistemas de arquivos já existentes, em ext2. Desta maneira, é possível que usuários do ext2 consigam obter as vantagens do *journaling* sem a necessidade de reinicializarem os seus discos através de comandos como *fdisk* e *mkfs*. Também é possível converter em sentido contrário, retirando-se o *journaling* para transformar um sistema de arquivos ext3 novamente em ext2 [TWE 2003].

Entretanto, para permitir esta conversibilidade o sistema de arquivos ext3 é implementado como sendo uma camada sobre o ext2. Talvez por este motivo, o ext3 apresente o pior desempenho entre todos sistemas de arquivos para vários tipos de operações [ZEM 2003] [BRY 2002].

Apesar de ele integrar o kernel do Linux a partir da versão 2.4.18, o ext3 é bastante recente e o uso dele encontrava-se destacado como "experimental" no início do desenvolvimento do FIJI. Como na escrita desta dissertação foi utilizado o kernel 2.4 (plataforma aonde o ext3 tem pouca confiabilidade) o ext3 foi desconsiderado.

O sistema de arquivos JFS também foi desconsiderado, por apresentar um desempenho inadequado para vários tipos de operações [ZEM 2003]. Logo, apesar de ele integrar o *kernel* do Linux a partir da versão 2.5.6, o JFS não foi considerado.

O interesse com relação ao ReiserFS deve-se ao fato de que ele foi a primeira implementação de *journaling filesystem* a fazer parte do *kernel* do sistema operacional

Linux [REI 2003]. Isto ajudou a sua disseminação e, por causa disso, atualmente o uso do ReiserFS é admitido na maior parte das distribuições Linux.

O ReiserFS apresenta uma séria desvantagem do ReiserFS em relação ao sistema de arquivos ext2. Ele ainda não prove o uso de quotas para usuários e grupos. Apesar de haver *patches* para corrigir o problema, eles não são recomendados pelos autores do ReiserFS. Isto representa uma grande dificuldade para a administração do sistema e, por este motivo, o ReiserFS foi desconsiderado.

O XFS possui uma grande confiabilidade, pois foi criado ainda em 1994 pela SGI para substituir o EFS [XFS 2003]. Ele é o mais antigo entre os sistemas de arquivos considerados para este trabalho, e a ênfase do projeto do XFS foi na capacidade de trabalhar com arquivos bastante grandes - da ordem de "*terabytes*".

O XFS pode trabalhar usando um tamanho de bloco variando entre 512 *bytes* e 64 k*bytes*, permitindo o uso de sistemas de arquivos distribuídos (incluindo NFS versão 3), ACLs no padrão POSIX 1003.e, e quotas para usuários e grupos. Enfim, justifica-se a escolha do sistema de arquivos XFS.

4.4 Sistema de arquivos XFS

No início dos anos 90, a empresa *Silicon Graphics* (ou *SGI*) precisou de um sistema de arquivos com alto desempenho e escalabilidade para satisfazer as necessidades de seus clientes [MOS 2000]. Desta maneira, o sistema de arquivos XFS foi projetado para ser otimizado com relação aos seguintes aspectos:

- tempo de recuperação em caso de falhas;
- tamanho máximo de todo um sistema de arquivos;
- tamanho máximo de cada arquivo;
- uso de grandes arquivos esparsos;
- número de arquivos; e
- tamanho de diretório.

Graças ao uso intensivo de *B+Trees* no lugar de algoritmos lineares, o XFS é capaz de fornecer escalabilidade ao mesmo tempo em que apresenta um desempenho comparável ao *throughput* do próprio *hardware*.

Na Figura 4.2, consta um exemplo para representação na forma de árvores-B+. Para isso foi considerado que o diretório /etc é a raiz e os nós contém o atributo nome de arquivo. Para esta árvore-B+, são realizadas operações para localizar um arquivo chamado resolv.conf. Essa operação de busca é realizada ao longo de 3 etapas, que são as seguintes:

1. A partir da raiz desta árvore, é feita uma busca seqüencial pelo nome "resolv.conf". Como a única entrada no primeiro nível é "mtab" e não existe mais nenhum outro nome de arquivo, é feito um deslocamento para o próximo nó interno do segundo nível à direita;
2. A partir deste nó "securetty", é feita uma nova busca pelo nome "resolv.conf". Como o valor "securetty" é maior do que "resolv.conf" segundo a ordem lexicográfica, é feito um deslocamento para o nó do terceiro nível à esquerda;
3. Como este nó interno "mtab" está no terceiro e último nível, ele é do tipo folha. Logo, vai ser preciso fazer uma busca seqüencial a partir dele junto à todos os outros nós folha do terceiro nível até que o nome "resolv.conf" seja encontrado.

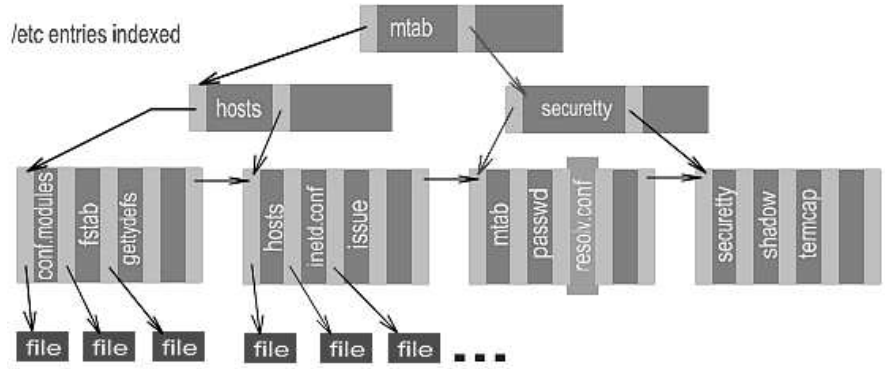

Figura 4.2: Exemplo de uma B-TREE+ para o diretório /etc

Também é possível uma recuperação extremamente rápida, graças ao *logging* assíncrono dos metadados para o sistema de arquivos [TRA 2003]. A seguir, são apresentadas as principais características do XFS.

4.5 Escalabilidade do XFS

A escalabilidade do sistema de arquivos é definida como a habilidade para manipular sistemas de arquivos muito grandes, arquivos grandes, árvores de diretórios grandes e grande número de arquivos a medida que mantém o mesmo desempenho de I/O. A escalabilidade de um sistema de arquivos depende, em grande parte, de como ele armazena informações sobre os arquivos. Por exemplo, se o tamanho do arquivo é armazenado como um número de 32 bits, então nenhum arquivo no sistema de arquivos pode de forma útil exceder 2^{32} *bytes* (4 GB).

A escalabilidade também depende dos métodos usados para organizar e acessar dados dentro do sistema de arquivos. Se as entradas de diretórios estão armazenadas como uma lista simples de nomes de arquivos sem nenhuma ordem, então para procurar um arquivo em particular, cada entrada deve ser pesquisada uma a uma até que a entrada desejada seja encontrada. Isto funciona bem para diretórios pequenos, mas não muito bem para diretórios grandes.

Os sistemas de arquivos antigos não são capazes de lidar com grandes capacidades de armazenamento. Eles foram projetados com certos tamanhos de arquivos, de diretórios e de partição. As estruturas dos sistemas de arquivos antigos possuem um número fixo de bits para armazenar informações do sistema de arquivos, um número fixo de bits para armazenar o número do bloco lógico, etc. Como uma conseqüência deste número fixo de bits, o tamanho dos arquivos, o tamanho das partições e o número de entradas de diretórios são limitados. Enfim, estruturas antigas geralmente não possuem o número de bits necessários para gerenciar certos objetos.

Embora estruturas antigas ainda sejam algumas vezes capazes de gerenciar novos tamanhos de objetos, elas são inadequadas de gerenciá-las por razões de desempenho. A principal razão é que certas estruturas comportam-se bem com tamanhos antigos, mas com tamanhos novos levam a perdas de desempenho.

4.5.1 Sistemas de arquivos grandes

Não apenas as unidades de disco individuais tem ficado maiores, mas a maioria dos sistemas operacionais permitem a criação de volumes cada vez maiores (unindo partições de múltiplas unidades). Dispositivos RAID também estão freqüentemente disponíveis, cada *array* RAID aparecendo como sendo um único dispositivo de grande capacidade. As necessidades de armazenamento de dados em muitas organizações estão praticamente dobrando a cada ano, tornando sistemas de arquivos maiores uma necessidade [TRA 2003].

O requerimento mínimo para permitir que um sistema de arquivos escale-se até 4 GB é o uso de tamanhos até 32 bits. Além disso, para ser eficiente, um sistema de arquivos deve também fornecer os algoritmos e organização interna apropriados para atender as demandas criadas pela quantidade muito maior de I/O que é provável em um sistema de arquivos grande.

O XFS inclui um gerente de volume integrado, o XLV, que é capaz de concatenar, espelhar, ou gerenciar até 128 volumes. Cada um desses volumes, por sua vez, pode consistir de até 100 partições de disco ou *arrays* RAID. Volumes separados que podem escalonar até centenas de *terabytes* de capacidade [TRA 2003].

O XFS é um sistema de arquivos em 64 bits. Os contadores globais no sistema são de 64 bits, assim como os endereços usados para cada bloco de disco e como o número exclusivo determinado para cada arquivo (o número do *inode*). Mas para evitar que todas as estruturas de dados no sistema de arquivos sejam de 64 bits, o sistema de arquivos é particionado em regiões chamadas Grupos de Alocação (ou AG, de Allocation *Groups*). Cada AG gerencia seu próprio espaço livre e *inodes* [MOS 2000].

Entretanto, o objetivo principal dos grupos de alocação é fornecer escalabilidade e paralelismo dentro do sistema de arquivos. Este particionamento também limita o tamanho das estruturas necessárias para rastrear esta informação e permite que os ponteiros internos sejam de 32 bits. AGs tipicamente variam de 0,5 a 4 GB. Arquivos e diretórios não estão limitados a alocar espaço num único AG [SWE 96].

O espaço livre e os *inodes* dentro de cada AG são gerenciados independentemente e em paralelo. Portanto, múltiplos processos podem alocar espaço livre por todo o sistema de arquivos simultaneamente.

4.5.2 Arquivos grandes

A maioria dos sistemas de arquivos tradicionais lida com arquivos de até 2^{31} (2 Gb) ou 2^{32} *bytes* (4 GB) de tamanho. Em outras palavras, eles não usam mais do que 32 bits para armazenar o tamanho do arquivo. Então, para poder lidar com arquivos maiores um sistema de arquivos deve usar mais bits para representar o tamanho do arquivo. Além disso, um sistema de arquivos deve estar apto para alocar e rastrear o espaço em disco usado pelo arquivo, mesmo quando o arquivo é muito grande - e fazê-lo eficientemente [TRA 2003].

O desempenho de I/O do arquivo pode ser aumentado se os blocos estiverem alocados contiguamente. Portanto, o método pelo qual o espaço em disco é alocado e rastreado é crítico. Há dois métodos genéricos de alocação de disco usado pelos sistemas de arquivos:

- **Alocação de Bloco:** blocos são alocados um de cada vez e um ponteiro é mantido para cada bloco no arquivo; e
- **Alocação de *Extents*:** grande número de blocos contíguos – chamados de *extents* – são alocados para o arquivo e rastreados como sendo uma unidade. Um ponteiro precisa apenas ser mantido no início do *extent*. Devido a um único ponteiro ser usado para rastrear um grande número de blocos, o controle para arquivos grandes é muito mais eficiente.

O método pelo qual o espaço livre dentro do sistema de arquivos é rastreado e gerenciado também torna–se importante, porque ele impacta diretamente na habilidade de rapidamente localizar e alocar blocos livres ou *extents* de tamanho apropriado. A maioria dos sistemas de arquivos utiliza estruturas lineares *bitmap* para mapear espaço livre versus alocado, onde cada *bit* no *bitmap* representa um bloco no sistema de arquivos. Entretanto, é extremamente ineficiente procurar através de um *bitmap* para encontrar grande blocos de espaço livre - particularmente quando o sistema de arquivos é muito grande ou encontra-se fragmentado.

É também vantajoso controlar o tamanho do bloco usado pelo sistema de arquivos. Esta é a unidade de tamanho mínimo que pode ser alocada dentro do sistema de arquivos. É importante distinguir aqui entre tamanho do bloco físico usado pelo *hardware* de disco (tipicamente fixo em 512 *bytes*), e o tamanho do bloco usado pelo sistema de arquivos – geralmente chamado de tamanho de bloco lógico.

Se um administrador do sistema sabe que o sistema de arquivos vai ser usado para armazenar arquivos grandes faria sentido usar o tamanho de bloco lógico máximo, e com isso reduzir a fragmentação externa. Fragmentação externa é o termo usado para descrever a condição quando arquivos estão espalhados em pequenos pedaços por todo o sistema de arquivos. No pior caso, em algumas implementações, o espaço em disco pode estar livre - mas não utilizável.

De modo oposto, se o sistema é usado por pequenos arquivos um bloco de tamanho pequeno faria sentido - e ajudaria a reduzir a fragmentação interna. Fragmentação interna é o termo usado para descrever o espaço em disco que está alocado para um arquivo mas não utilizado porque o arquivo é menor do que o espaço alocado.

O XFS é um sistema de arquivos em 64 bits. Todas as estruturas de dados estão apropriadamente definidas para lidar arquivos de até 2^{64}-1 *bytes* (9 exa*bytes*). O XFS utiliza alocação de *extents* de tamanho variado para permitir que arquivos aloquem os maiores blocos possíveis de espaço contíguo [TRA 2003].

Cada *extent* é descrito por uma tripla, composta pelos elementos:

- *Início* - endereço do bloco inicial para o *extent* dentro de um arquivo;
- *Tamanho* - tamanho do *extent*, em blocos; e
- *Offset* - deslocamento para o primeiro byte de um *extent* dentro de um arquivo.

Uma possível representação para um *extent* é a da Figura 4.3:

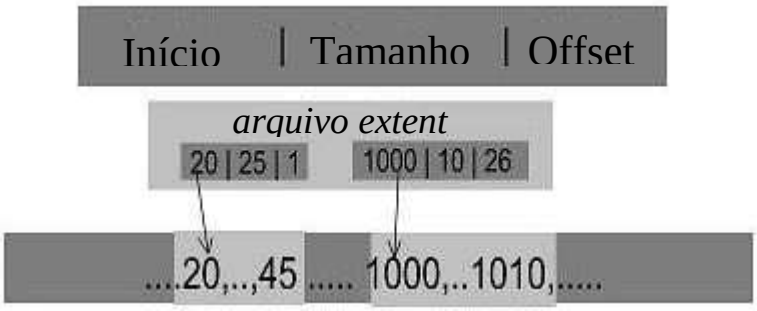

Figura 4.3: Representação para a tripla que descreve um extent

Um único *extent* pode consistir de até dois milhões de blocos contíguos ou um máximo de 4 GB de espaço de disco (o limite aplicado depende do tamanho do bloco lógico). Apesar da habilidade para alocar *extents* muito grandes, arquivos no XFS podem ainda consistir de um grande número de *extents* de tamanhos variados, dependendo do modo que o arquivo cresce ao longo do tempo.

Os *inodes* do XFS contém 9 ou mais entradas que apontam para *extents*. O tamanho do *inode* é ajustável no XFS, mas o tamanho padrão de 256 *bytes* permite 9 entradas diretas. Se um dado arquivo contém mais *extents* do que isso, as *extents* do arquivo são mapeadas por uma árvore-B+ para melhorar a velocidade com a qual qualquer bloco no arquivo pode ser localizado.

Conforme mencionado anteriormente, o espaço livre no XFS é gerenciado conforme a base do AG. Cada AG mantém duas árvores-B+ que descrevem os seus *extents* livres: uma árvore é indexada pelo bloco inicial dos *extents* livres, e a outra pelo tamanho dos *extents* livres. Dependendo do tipo de alocação, o sistema de arquivos pode rapidamente localizar tanto o *extent* mais próximo para um determinado local ou rapidamente encontrar um *extent* de um determinado tamanho. O XFS também permite que o tamanho do bloco lógico varie de 512 *bytes* a 64 KB, conforme a base do sistema de arquivos.

4.5.3 Arquivos esparsos

Algumas aplicações (tais como banco de dados, por exemplo) criam arquivos com grandes quantidades de espaços em branco dentro deles – ao menos inicialmente. Uma quantidade significativa de espaço em disco seria salva se o sistema de arquivos pudesse evitar alocar espaço em disco até que este espaço em branco fosse preenchido.

O uso de arquivos esparsos está fortemente relacionado à técnica de endereçamento de *extents* para os blocos de arquivo.Ele tira vantagem do campo *offset* dos descritores de *extents*. Além disso, toda vez que o sistema de arquivos deve procurar por blocos livres apenas para preencher o intervalo aberto por uma situação como a descrita acima, o sistema de arquivos apenas ajusta um novo *extent* com o correspondente campo *offset*. Desde então, toda a vez que uma aplicação tentar ler um dos *bytes* dentro do intervalo, um valor *null* será retornado - já que não há nenhuma informação lá. Finalmente, o intervalo deverá ser preenchido por outras aplicações que escrevam em *offsets* dentro do intervalo.

O XFS fornece um espaço de endereço esparso de 64 bits para cada arquivo. O XFS permite que arquivos tenham "buracos" que não sejam alocados. O uso de sistemas de arquivos de 64 bits significa que potencialmente há um número muito grande de blocos a serem indexados para cada arquivo. Os métodos que o XFS usa para alocar e

gerenciar *extents* torna isto eficiente, uma vez que o XFS armazena o *offset* do bloco dentro do arquivo como parte do descritor de *extent*. Então, *extents* podem ser descontínuos.

4.5.4 Diretórios grandes

Grandes *softwares* e aplicações, tais como *sendmail*, *netnews* e criação de mídia, geralmente resultam em diretórios únicos, contendo literalmente milhares de arquivos. Procurar por um nome de arquivo em tal diretório pode durar uma quantidade de tempo significativa se fosse feita uma pesquisa linear

O sistema de arquivo XFS usa árvores-B+ para organizar as entradas dentro de um diretório, levando a melhores tempos de pesquisa. Neste sistema de arquivo, as entradas para cada diretório são organizados em uma árvore-B+ - indexando as entradas por nome de diretório. Desse modo, quando um arquivo sob um dado diretório é solicitado, a árvore-B+ de diretório é percorrida para localizar o *inode* do arquivo rapidamente.

Enquanto há sistemas de arquivos que mantém uma árvore-B+ para cada diretório, enquanto outros mantém uma única árvore-B+ do sistema de arquivos para toda a árvore de diretório do sistema de arquivos. O XFS usa uma estrutura árvore-B+ no disco para os seus diretórios. Nomes de arquivos no diretório são primeiramente convertidos para valores *hash* de quatro *bytes* que são usados para indexar a árvore-B+.

A estrutura da árvore-B+ procura, cria, e remove operações em diretórios com milhões de entradas. Entretanto, listar os conteúdos de um diretório com milhões de entradas permanece impraticável devido ao tamanho do resultado de saída.

4.5.5 Alocação dinâmica de *inodes*

Para que seja possível lidar com um grande número de arquivos eficientemente, um sistema de arquivos deve alocar dinamicamente os *inodes* que serão usados para localizar os arquivos. Com um grande número de arquivos, é também razoável esperar que os acessos, criações e remoções de arquivos serão em muitos casos numerosos. Portanto, o sistema de arquivos deve também permitir que múltiplas operações em arquivos ocorram em paralelo.

No XFS, o número de arquivos num sistema de arquivos é limitado apenas pela quantidade de espaço disponível. O XFS aloca dinamicamente *inodes* a medida que vão sendo necessários.

Cada grupo de alocação gerencia os *inodes* dentro de seus confinamentos. Os *inodes* são alocados 64 por vez, e uma árvore-B+ em cada grupo de alocação guarda o local de cada grupo de *inodes* e registros de quais *inodes* estão em uso. O XFS permite que grupos de alocação funcionem em paralelo, permitindo um maior número de operações simultâneas [TRA 2003].

4.6 Recuperação de falhas no XFS

A tarefa de recuperação em caso de falhas, para o sistema de arquivos XFS, consiste basicamente em restaurá-lo para um estado consistente. Quando uma falha ocorre, uma ou mais transações podem ainda estar ativas e não terem sofrido o *commit*.

Considerando-se que as alterações no sistema de arquivos XFS devem ser tratadas como sendo transações ACID – portanto, atômicas [GRA 93] - o efeito de transações executadas parcialmente deve ser removido. Para isso, existe um *log before image* para os processos de *undo* ou *rollback*. Através dele, é possível obter o conteúdo que havia em disco antes de uma transação haver sido executada apenas parcialmente.

Além disso, uma transação pode ter sofrido *commit* e uma falha ocorrido antes de seus resultados terem sido efetivamente gravados em disco. Segundo outro princípio das transações ACID, de que elas também devem ser duráveis [GRA 93], pode ser necessário refazê-las. Para isso, existe um *log after image* para os processos de *redo* ou *replay*.

O *software* encarregado da recuperação para o sistema de arquivos XFS não é o *fsck* - mas o próprio *kernel* do sistema operacional. Durante as operações de montagem para sistemas de arquivos deste tipo, o trecho de código existente no *kernel* correspondente a ele é quem realiza as operações necessárias para a recuperação - baseando-se nos *logs*.

Apesar de existirem diferentes versões de *fsck* para vários tipos de sistemas de arquivos, as versões de *fsck* para os sistemas de arquivos baseados em *journaling* geralmente não fazem nada.

4.6.1 Estado do sistema de arquivos XFS

O estado de um sistema de arquivos XFS é o resultado da combinação de informações localizadas em três lugares diferentes: disco, memória e *log* do *journaling*. O sistema de arquivos somente estará em um estado consistente após ele ter sofrido *shutdown*, onde todos os dados residentes em memória são gravados em disco e as entradas do *log* de transações aplicadas no sistema de arquivos.

Os locais que definem o estado de um sistema de arquivos são:

1. Os discos, onde parte das informações encontra-se gravada efetivamente. As informações contidas neles podem estar incompletas, e até mesmo inconsistentes - visto que dados alterados podem ainda não ter sido gravados;
2. A memória, que contém os dados alterados recentemente e que ainda não foram gravados em disco (tais como *buffer* e *cache*, por exemplo) pela chamada de sistema *sync* [TAN 97][BAR 2000]; e
3. O *log* de transações, que contém as alterações realizadas recentemente no sistema de arquivos. O *log* de transações também pode estar incompleto, porque transações antigas são descartadas e o seu espaço é reutilizado.

4.7 Resumo

Este capítulo apresentou detalhes sobre a organização de sistemas de arquivos baseados em journaling – e sobre o XFS, em especial. Descreveu-se como é a localização e conteúdo dos logs, como é feita a verificação de consistência e os motivos que justificaram a escolha do XFS.

5 AMBIENTE LINUX

Este capítulo contém uma descrição de características básicas do ambiente Linux que são importantes para o entendimento sobre como o FIJI funciona. O leitor familiarizado com o assunto não encontrará nenhuma contribuição nova sobre o tema.

Este capítulo descreve brevemente como o Linux realiza a manipulação de sinais (ou *signals*), que são utilizados para notificar um processo ou *thread* de um evento em particular. O FIJI utiliza o sinal SIGTRAP, conforme descrito nos próximos capítulos.

E este capítulo também discute como funciona a dinâmica das chamadas de sistema, necessárias para que compreenda-se os recursos de depuração do Linux. O FIJI intercepta as chamadas de sistema *read* e *write* para não afetar o armazenamento estável durante a injeção de falhas.

5.1 Sinais

5.1.1 Introdução

Os processos de usuário são executados cada um dentro de seu próprio espaço de endereçamento virtual, e protegidos pelo sistema operacional da interferência de uns nos outros [TAN 97]. Como padrão, um processo de usuário não pode comunicar-se com outros - a menos que utilize-se de mecanismos seguros, gerenciados pelo *kernel*. Entretanto, muitas vezes processos necessitam compartilhar recursos em comum, ou mesmo sincronizar as suas ações.

Uma das possibilidades para resolver este problema é utilizar *threads* que, por definição, compartilham do mesmo contexto do processo ao qual estão vinculadas. Entretanto, nem sempre é possível empregar *threads* para resolver o problema e, quando isso acontece, faz-se necessário um método para troca de mensagens e/ou dados entre processos – e um destes mecanismos é o sistema de sinais do Linux [BAR 2000].

Sinais são a forma mais simples para comunicação interprocessos. Elas permitem que um processo seja interrompido de modo assíncrono por outro processo (ou então pelo *kernel*) para realizar a rotina de tratamento a este evento. Uma vez que o sinal é devidamente tratado, o processo que foi interrompido continua a sua execução exatamente do ponto em que foi interrompido.

Pode-se comparar o sistema de sinais do Linux com as interrupções de *hardware*, que ocorrem quando um dispositivo (tal como uma *interface* de disco, por exemplo) gera uma interrupção para o microprocessador como resultado de uma operação (tal como término de uma instrução de entrada/saída, por exemplo). Isto, por sua vez, faz com que o microprocessador execute uma rotina para tratamento de interrupção – a que possui procedimentos a serem executados para eventos deste tipo.

O Linux implementa sinais em total conformidade com o padrão POSIX. Os sinais são usados no Linux para tarefas como encerrar a execução de processos, ou dizer para algum *daemon* que ele deve ler de novo os seus arquivos de configuração. O uso de sinais é um padrão em sistemas Unix, de um modo geral; e o *kernel* do Linux utiliza-os para informar processos de uma variedade de eventos, tais como:

- A morte de um processo filho;
- Um alarme ajustado pelo processo que tenha expirado; e
- Tamanho de uma janela de terminal que tenha mudado.

5.1.2 Conceitos relacionados a sinais

Cada sinal possui um nome que é único: uma abreviação, que sempre começa por SIG, seguida por um número correspondente ao nome do sinal. Adicionalmente, para todos os possíveis sinais o sistema operacional permite definir uma ação que será executada como padrão para quando estes ocorrerem. Existem as seguintes possibilidades:

- *Exit* – força o processo a encerrar;
- *Core* – força o processo a encerrar e criar um arquivo *core*;
- *Stop* – suspende a execução do processo; e
- *Ignore* – ignora o sinal, não executa nenhuma ação.

O ação padrão, que pode ser selecionada dentro do contexto de cada processo em execução, indica a ação que deverá ser executada pelo *kernel* quando receber um sinal deste tipo. Esta definição deve ser única entre todas as *threads* de um mesmo processo.

Fazer o ajuste de um sinal é simples, isto deve ser feito por meio da chamada de sistema *signal* para informar ao *kernel* como lidar com um sinal em particular. Uma descrição de como pode ser feito o ajuste está na Figura 5.1:

```
#include <signal.h>
void * int signal (int signumber, void * handler);
```

Figura 5.1: Definição de um sinal

Na Figura 5.1, o parâmetro *signumber* informar o número correspondente ao sinal que deseja-se manipular (de acordo com a Tabela 5.1, logo adiante). O campo *handler* informa qual a ação que deve ser executada em resposta, como padrão, para quando um sinal deste tipo for recebido pelo processo. Uma vez que o sinal é enviado para o processo, o *kernel* encarrega-se de executar a rotina especificada em *handler* o mais rapidamente possível.

5.1.3 Tipos de sinais

A ocorrência de um sinal pode ser síncrona ou assíncrona a um processo ou *thread*, dependendo da origem do sinal e do motivo que causou o seu envio. Sinais

síncronos ocorrem como resultado direto da execução de um comando, na qual ocorreu um erro irrecuperável (tal como referência a um endereço de memória inválido) que exige o término imediato do processo. Neste caso, o sinal correspondente será enviado para a *thread* cuja execução causou o erro. Quando um evento deste tipo ocorre, o *kernel* da máquina Linux faz a "captura" do mesmo e acessa a correspondente rotina para tratamento de evento deste tipo. Por este motivo, sinais síncronos são muitas vezes referenciados como sendo *traps,* e fala-se que houve um *catching* do sinal.

Já os sinais assíncronos, por sua vez, tem como origem eventos externos ao contexto do processo em execução. Um exemplo de sinal assíncrono seria o envio da mensagem *kill* de um processo para outro. Os sinais assíncronos são muitas vezes referenciados como sendo *interrupts*.

Quando um processo recebe um sinal, ele pode fazer uma das seguintes opções:

- Ignorar o sinal;
- Permitir que o *kernel* execute uma parte especial do processo antes de permitir que o processo continue (isto significa fazer um *catching*, ou tratamento do sinal); e
- Permitir que o kernel execute a sua ação padrão, a qual dependerá do tipo de sinal recebido.

5.1.4 Relação de sinais permitidos pelo Linux

A Tabela 5.1 [BAR 2000] contém uma lista completa de todos os sinais permitidos pelo Linux, juntamente com uma descrição da ação que deve ser executada em resposta, como padrão. Para este trabalho, o único sinal que importa é o SIGTRAP.

Tabela 5.1: Lista de sinais Linux e ações padrão

Nome	Número	Ação padrão	Descrição
SIGHUP	1	*Exit*	*Hangup*
SIGINT	2	*Exit*	*Interrupt*
SIGQUIT	3	*Core*	*Quit*
SIGILL	4	*Core*	Instrução ilegal
SIGTRAP	5	*Core*	*Trace* ou *breakpoint trap*
SIGABRT	6	*Core*	*Abort*
SIGEMT	7	*Core*	*Emulation trap*
SIGFPE	8	*Core*	*Arithmetic exception*
SIGKILL	9	*Exit*	*Kill*
SIGBUS	10	*Core*	*Bus error*
SIGSEGV	11	*Core*	*Segmentation fault*
SIGSYS	12	*Core*	*Bad system call*
SIGPIPE	13	*Exit*	*Broken pipe*
SIGALRM	14	*Exit*	*Alarm clock*
SIGTERM	15	*Exit*	*Terminated*
SIGUSR1	16	*Exit*	Sinal 1 definido pelo usuário
SIGUSR2	17	*Exit*	Sinal 2 definido pelo usuário
SIGCHLD	18	*Ignore*	Mudou o status do processo filho
SIGPWR	19	*Ignore*	Falha de energia ou reinício
SIGWINCH	20	*Ignore*	Mudança no tamanho da janela

SIGURG	21	*Ignore*	*Urgent socket condition*
SIGPOLL	22	*Exit*	*Pollable event*
SIGSTOP	23	*Stop*	*Stop* (não pode ser ignorado)
SIGSTP	24	*Stop*	*Stop (job control)*
SIGCONT	25	*Ignore*	*Continued*
SIGTIN	26	*Stop*	*Stopped – tty input*
SIGTTOU	27	*Stop*	*Stopped – tty output*
SIGVTALRM	28	*Exit*	*Virtual timer expired*
SIGPROF	29	*Exit*	*Profiling timer expired*
SIGXCPU	30	*Core*	*Cpu time limit exceeded*
SIGXFSZ	31	*Core*	*File size limit exceeded*
SIGWAITING	32	*Ignore*	*Concurrency signal used by threads*
SIGLWP	33	*Ignore*	*Inter-LWP signal used by threads*
SIGFREEZE	34	*Ignore*	*Checkpoint suspend*
SIGTHAW	35	*Ignore*	*Checkpoint resume*
SIGCANCEL	36	*Ignore*	*Cancellation signal used by threads*
SIGLOST	37	*Ignore*	*Resource lost*
SIGRTMIN	38	*Exit*	*Highest priority real-time signal*
SIGRTMAX	45	*Exit*	*Lowest priority real-time signal*

5.1.4.1 Descrição de sinais e ação padrão

A ação que um sinal executa como padrão em resposta às suas chamadas pode ser alterada para valores diferentes da Tabela 5.1. Desta maneira, um processo pode ser organizado para fazer o tratamento de um sinal por si mesmo, ou então ignorá-lo. A única exceção à esta regra são os sinais SIGKILL e SIGSTOP, cuja a ação padrão não pode ser modificada [BAR 2000]. Esta mudança pode ser feita por meio das rotinas de biblioteca *signal* e *sigset*, e da chamada de sistema *sigaction*.

Sinais podem também ser bloqueados, fazendo com que o processo não receba mensagens de um determinado tipo. A geração de um sinal que tenha sido bloqueado fará com que o sinal permaneça pendente para este processo, até que o sinal seja explicitamente desbloqueado ou então a ação padrão torne-se *Ignore*.

5.1.4.2 Implementação de sinais

Em termos de implementação, cada sinal é representado por um *bit* em uma estrutura de dados. Mais explicitamente, quando o *kernel* envia um sinal para um processo ou *thread*, isso faz com que um *bit* seja habilitado. Uma vez que cada signal é identificado por um número único (como consta na Tabela 5.1), basta usar uma estrutura de dados com largura suficiente para representar todos os sinais da Tabela 5.1 com um *bit* para cada um. Por exemplo, ajustar o 17^{o} bit para 1 significa enviar para o processo ou *thread* o sinal SIGUSR1.

Ao considerar-se que enviar um sinal nada mais é do que ajustar um *bit* em uma tabela, deve-se levar em conta que é necessário verificar esta tabela de tempos em tempos para procurar por sinais pendentes ajustados pelo *kernel*. Uma verificação é feita na tabela por várias vezes durante a execução de um processo:

1. No retorno de uma chamada de sistema;

2. No retorno de uma *trap; e*
3. No "despertar" de um processo que estava em modo "sleep".

Em essência, a determinação da existência ou não de um sinal nada mais é do que um processo de *polling* em que é feita uma verificação de um *bit*. Uma vez que o teste é feito e determina-se que um sinal foi postado, a ação apropriada pode ser executada pelo processo que recebeu o sinal.

5.2 Chamadas de sistema

5.2.1 Introdução

O termo chamadas de sistema (ou *syscalls*) aparece inúmeras vezes no texto desta dissertação, pois ele é fundamental para o processo de programação em ambiente Linux. A uma primeira vista, as chamadas de sistema são parecidas com as funções em linguagem C. E isso não está errado, pois de fato as chamadas de sistema nada mais são do que funções em C – apenas pertencem a uma variedade especial. Para entender a diferença entre uma chamada de sistema e uma função em C, é necessário um entendimento sobre como funciona o sistema operacional Linux.

Embora existam inúmeras partes de código que componham um ambiente Linux (tais como programas utilitários, aplicativos, bibliotecas de programação, *drivers* de dispositivos, sistemas de arquivos, gerência de memória, e assim por diante), todas essas peças nada mais são do que programas, que devem ser executados em um destes seguintes contextos: ou modo *kernel*, ou modo *user* [TAN 97]. Quando um programa comum é executado, isto é feito em modo *user*. Em contrapartida, *drivers* de dispositivos e sistemas de arquivos são executados em modo *kernel*.

No modo *user*, os programas executados sofrem uma série de restrições com vistas a protegê-los uns dos outros e também para não afetarem o resto do sistema. Entretanto, código executado em modo *kernel* possui acesso total a todos os recursos da máquina e pode fazer praticamente qualquer coisa - sem restrições.

Para que um *driver* de dispositivo consiga controlar o *hardware* associado, ele deve ter acesso total ao mesmo. Assim como o *driver* de dispositivo, todo o código que é executado em modo *kernel* existe apenas para oferecer serviços ao código executado em modo *user*. As chamadas de sistema são o modo como aplicações executadas em modo *user* solicitam serviços oferecidos pelo modo *kernel*.

Os detalhes envolvendo a transição que ocorre entre modo *user* e modo *kernel* são omitidos pela *interface* da biblioteca em C. Entretanto, usar as chamadas de sistema nos programas em C impõe algumas restrições – com vistas a otimização. O sistema operacional Linux possui mais de 200 chamadas de sistema, cuja documentação completa pode ser obtida em *manpages*.

5.2.2 Implementação das chamadas de sistema no Linux

Existem dois mecanismos pelos quais o Linux implementa chamadas de sistema:

- Lcall7/lcall27 call gates; e
- Interrupção de *software* 0x80.

O XFS e programas que são nativos para o Linux implementam o segundo método (int 0x80), enquanto binários que tenham sido compilados para outras versões de Unix (tais como Solaris e Unixware, por exemplo) usam o mecanismo lcall7. A *interface* para as chamadas de sistema do Linux são vetorizadas por meio de um *stub* na biblioteca g*libc*, sendo que o único modo de fornecer parâmetro para elas é através dos registradores.

O Linux permite o máximo de 6 parâmetros para chamadas de sistemas. Eles devem ser fornecidos através dos registradores %ebx, %ecx, %edx, %esi, %edi e %ebp (este último, sendo usado apenas temporariamente). O registrador %eax contém o número que identifica a chamada de sistema solicitada [BAR 2000].

Como padrão, o Linux realiza chamadas de sistema por meio da interrupção 0x80. O vetor desta instrução é utilizado para transferir o controle de modo *user* para modo *kernel*. Este vetor é inicializado no momento em que a máquina é ligada.

Durante o processo de *boot*, o *kernel* faz um ajuste no *interrupt descriptor table (IDT)* para informar que a interrupção de *software* 0x80 é o *call gate* para executar funções em modo *kernel* [BAR 2000].

5.2.3 Limitações das chamadas de sistema

As operações em modo *kernel* são protegidas pelo sistema operacional de eventuais abusos que possam ser cometidos em modo *user*. Uma das maneiras como é feita esta proteção está no tipo de dados que pode ser transferido entre modos *user* e *kernel*. Isto é feito observando-se as seguintes convenções [BAR 2000]:

- Cada argumento que é passado do modo *user* para o modo *kernel* deve ser de mesmo tamanho, o qual quase sempre corresponde ao tamanho máximo de palavra permitido pelo microprocessador da máquina. Este tamanho é grande o bastante para passar argumentos do tipo inteiro longo, bem como ponteiros. Os tipos de variáveis *char* e *short* são estendidos para tipos de dados maiores antes de serem repassados ao *kernel*; e
- tipo de retorno desta função está limitado a ser do tipo *signed word*. As primeiras centenas de números negativos são reservadas para códigos de erro, e tem um significado comum entre todas as chamadas de sistema.

Diferentemente de programas em C, em que estruturas podem ser passadas como parâmetro por valor (através da pilha), usando-se chamadas de sistema não é permitida a passagem de parâmetros por valor. Além disso, não é permitido que o *kernel* retorne estruturas como resultados para o modo *user*. Quando houver uma quantidade grande de parâmetros a serem passados, isto deve ser feito por referência – usando-se ponteiros.

5.2.4 Código de retorno de chamadas de sistema

Os códigos que são retornados após a execução de uma chamada de sistema são reservados, sendo todos números negativos pequenos. A biblioteca C verifica a existência de erros no momento em que encerra a execução de uma chamada de sistema. Se ocorreu algum erro, o valor do erro é atribuído a uma variável chamada *errno*. Na

maior parte das vezes, para testar se houve erro basta verificar se o resultado da execução desta função foi um número negativo.

5.3 Conceito de programa

Conforme foi dito na Seção 5.2, as chamadas de sistema são a *interface* entre os programas do usuário e os recursos oferecidos pelo *kernel*. Todas as requisições de acesso a recursos do sistema operacional e acesso ao *hardware* são executados via chamadas de sistema.

Para o FIJI, entende-se "programa" como sendo um conjunto de instruções. Este conjunto de instruções, por sua vez, corresponde a um código que é executável. A geração de um programa possui três possibilidades [GON 2001]:

- Uso do próprio código-fonte com uma linguagem interpretada (tal como Perl) ou linguagem de *script* (tal como *shell* ou awk);
- Geração de um metacódigo a ser executado por um interpretador ou máquina virtual (tal como a *jvm* do Java, por exemplo);
- Compilação e ligação de um código-fonte, gerando um código executável nativo.

A ferramenta de injeção de falhas FIJI foi projetada para agir diretamente sobre um código executável. Entretanto, o FIJI agiria de modo diferente para cada um dos tipos de códigos executáveis apresentados.

No caso dos programas que interpretam o próprio código-fonte, o FIJI agiria sobre o interpretador ou *shell* onde os programas em código-fonte são executados. No caso de metacódigo, o FIJI agiria sobre o interpretador ou máquina virtual. Com isso, em ambos os casos todos os programas executados neste ambiente poderiam ser afetados pela injeção de falhas – mesmo que isso não fosse desejado.

Em um caso limite, pode-se imaginar que se fossem interceptadas todas as chamadas de sistema *write* realizadas por um interpretador nós poderíamos até mesmo impedir um usuário trabalhando em um ambiente IDE (de *Integrated Development Environment* - tal como um Turbo-Pascal, por exemplo) de salvar o seu código-fonte. Este processamento seria incorreto, pois o efeito desejado seria impedir que o código-fonte, ao ser executado dentro deste IDE, fizesse operações de escrita.

O FIJI foi projetado para trabalhar apenas sobre processos correspondentes a programas do terceiro tipo: que correspondam à código executável gerado pela compilação e ligação de um código-fonte. Para este caso, o efeito do injetor está restrito à aplicação em si e, de forma moderada, com as aplicações com as quais esta comunica-se. Em suma, o FIJI age sobre código executável.

No caso de compilação e ligação, o código executável é modelado segundo um dos formatos binários reconhecidos pelo Linux. – tal como ELF, COFF ou a .out. Cada formato possui uma série de peculiaridades, sendo ELF o formato padrão para código executável em Linux [BAR 2000].

Um programa compilado pode ser ligado de duas formas: estaticamente com as bibliotecas, ou ligado a bibliotecas dinâmicas (ou compartilhadas). No primeiro caso, funções de uma dada biblioteca utilizada pelo programa são adicionadas ao mesmo durante o processo de ligação. Já no segundo caso, o programa possui apenas referências às funções de uma dada biblioteca, que são adicionadas ao processo no momento da execução [LAD 98].

Programas compilados e ligados estaticamente são maiores, porém possuem uma maior independência do ambiente no tocante à execução. A ausência de uma ou outra biblioteca no sistema não impedem a execução do programa.

A idéia do uso de bibliotecas dinâmicas já vem desde o Multics (um antepassado do Unix). A utilização de bibliotecas dinâmicas possibilita a reutilização de código e facilita a manutenção e correção de erros de código – pois uma correção em uma dada biblioteca seria aproveitada por todos os programas que utilizem-se dela [GON 2001].

5.4 Glibc

A biblioteca Glibc (GNU Libc) é a biblioteca padrão do sistema operacional Linux, responsável pela *interface* entre o programa do usuário e as chamadas de sistema. Ela realiza a conversão dos dados fornecidos a funções de alto nível como *write* ou *printf* para o formato reconhecido pelas chamadas de sistema que atendem à essas requisições. Para este caso, ambas as funções (*write* e *printf*) são atendidas por uma mesma chamada de sistema, *write.*

Muitas, mas não todas as chamadas de sistema descritas na Seção 5.2 são declaradas no arquivo <unistd.h>. Este arquivo tem a função de armazenar todas as chamadas de sistema que não teriam sentido em outras partes do *kernel*. De uma maneira geral, durante a programação em linguagem C não basta apenas fazer o *include* deste arquivo <unistd.h> para usar as funções - o ideal é usar as *manpages* para determinar qual arquivo de *header* incluir para utilizar esta função em um programa.

5.4.1 A função *write*

Tomando-se a função *write,* ela é declarada no arquivo <unistd.h> da forma descrita na Figura 5.2:

```
Ssize_t write (int fd, const void *buf, size_t count);
```

Figura 5.2: Declaração da função *write*

O parâmetro fd é um valor inteiro (32 bits) representando o descritor de arquivo em uso (vide Seção 3.4.6). O segundo parâmetro, buf, é um ponteiro para o *buffer* de memória a ser gravado no arquivo apontado por fd. O último parâmetro, *count*, é a quantidade de *bytes* a serem copiados do *buffer* para o arquivo. Este também é um valor de 32 bits, assim como o ponteiro. Cabe salientar que os valores aqui citados possuem o tamanho de 32 bits na arquitetura utilizada neste experimento, x86 – Intel 32 bits.

Os valores dos parâmetros são copiados na pilha e, então, é chamada a função __libc_write(). Esta função lê os valores da pilha e os armazena em registradores. As chamadas de sistema possuem uma *interface* definida, com um registrador certo para o descritor de arquivo, outro registrador para o ponteiro do parâmetro buf e um terceiro registrador para a quantidade de *bytes* a ser armazenada.

A função __lib_write() coloca o valor 4 no registrador %eax, indicando que a chamada de sistema é a *write,* e executa a interrupção de *software* 0x80. Como foi dito na seção 7.2.2, isto executa a chamada de sistema requisitada passando o controle ao *kernel* e retornando após atender a requisição.

De volta à função __libc_write() é verificado o código de retorno da chamada de sistema, em busca de uma sinalização de erro. Caso tenha sido sinalizado um erro, é chamada a função __syscall_error(). Caso contrário, o resultado da operação é retornado ao processo do usuário, que continua sua execução normal.

Todas as chamadas de sistema possuem uma função correspondente na Glibc. E é esta função respectiva que prepara os dados antes de executar a chamada de sistema.

5.4.2 A função *read*

Tomando-se como exemplo a função *read*, ela é declarada no arquivo <unistd.h> da forma descrita na Figura 5.3:

```
ssize_t read (int fd, const void *buf, size_t count);
```

Figura 5.3: Declaração da função *read*

O parâmetro fd é um valor inteiro (32 bits) representando o descritor de arquivo em uso (vide Seção 3.4.6). O segundo parâmetro, buf, é um ponteiro para o *buffer* de memória para onde deve ser lido o arquivo apontado por fd. O último parâmetro, *count*, é a quantidade de *bytes* a serem copiados do arquivo para o *buffer*. Este também é um valor de 32 bits, assim como o ponteiro.

A relação entre processos, biblioteca Glibc e chamadas de sistema para a chamada de sistema *read* é análoga à da Seção 5.4.1 para a chamada de sistema *write.*

5.5 Recursos de depuração do Linux (ptrace)

O principal recurso de depuração oferecido pelo Linux é a função *ptrace* [BAR 2000]. A imensa maioria das ferramentas de depuração no Linux (incluindo o utilitário *strace* e a ferramenta FIDe, no qual o FIJI foi baseado) são construídas com base nesta função, que possui uma chamada de sistema homônima.

Esta função permite que um processo seja executado de três diferentes maneiras:

- Até que seja encontrado um ponto de parada (ou *breakpoint*). O ponto de parada é a instrução "int 3h" em *assembly*, e pode ser colocada em pontos considerados "estratégicos" no programa sob depuração. Entretanto, a colocação desta instrução em um ponto incorreto pode acarretar alterações no código do programa e, portanto, em seu comportamento – levando a resultados imprevisíveis;
- De modo passo-a-passo (ou *step-by-step*). O processo é executado, uma instrução em *assembly* de cada vez. Para executar o equivalente a uma função ou comando da linguagem C seriam necessárias várias chamadas à função *ptrace*, utilizando-se este método. Geralmente, um comando em linguagem de alto nível traduz-se em várias instruções em linguagem de máquina; e
- Executar até que seja encontrada uma chamada de sistema. O processo executaria até encontrar a função, e pararia ao encontrar a instrução "int 80h" no ponto de entrada. Uma segunda chamada a este método realizaria a

requisição feita pelo processo de usuário e retornaria, parando no ponto de saída. Desta forma, duas chamadas de sistema a este método executam toda uma chamada de sistema.

Para que seja possível controlar a execução de um processo com *ptrace*, é preciso que o depurador esteja conectado ao processo. A funcão *ptrace* permite que faça-se isso de duas formas [LAD 98]:

- Ligando-se a um processo já em execução, por meio da requisição PTRACE_ATTACH; ou
- Ativando o *flag* de depuração do processo antes de executá-lo. Isto geralmente é feito da seguinte forma: quando duplica-se um processo via *fork*, ativa-se o *flag* de depuração via uma requisição PTRACE_TRACEME

Com o processo sob controle, pode-se inspecionar o conteúdo de seus registradores, memória e de uma de suas estruturas de controle – a estrutura *user*. Entretanto, este recurso deve ser usado com moderação pois pedir que um processo execute até um ponto de parada exige um custo computacional de seis chaveamentos de contexto.

Quando o sistema operacional deixa de executar um processo e passa a executar outro, existe um conjunto considerável de procedimentos de administração a serem realizados antes que o novo processo possa ser realmente executado. Uma série de informações do processo atual devem ser salvas e informações do novo processo devem ser atualizadas em memória – tais como valor de registradores e mapas de memória virtual. Listas e tabelas do próprio sistema operacional devem ser atualizadas. A este processo, denomina-se chaveamento de contexto [TAN 97].

Os seis chaveamentos de contexto citados anteriormente dizem respeito à comunicação do processo depurador com o *kernel*, e do *kernel* com o processo sob depuração. Esta comunicação encontra-se descrita mais detalhadamente nas seções seguintes.

5.6 Resumo

Este capítulo apresentou como funcionam os sistemas de sinais (ou *signals*) e chamadas de sistema (ou *syscalls*) para o sistema operacional Linux. Isso é necessário para o entendimento sobre como funciona a ferramenta FIJI. Foram relacionados os sinais e chamadas de sistema usados pelo FIJI, bem como os parãmetros que são afetados durante a sua manipulação.

6 O INJETOR DE FALHAS FIJI

Este capítulo apresenta o modelo de falhas e uma descrição da ferramenta FIJI. Também são apresentados os resultados após a execução de programas de teste que foram projetados para avaliar a intrusividade da ferramenta em diferentes cenários.

6.1 Introdução

Nos capítulos anteriores apenas foram descritas técnicas para injeção de falhas, o sistema de arquivos Linux, os sistemas de arquivos baseados em *journaling* e a execução de processos com os seus mecanismos de depuração. A partir deste ponto, a correlação entre esses elementos começa a tomar forma.

O FIJI baseia-se na função *ptrace* para realizar a depuração. Conforme consta na seção 6.4, esta função *ptrace* oferece três alternativas para depuração: execução passo-a-passo, execução até um ponto de parada ou execução até encontrar uma chamada de sistema. Foram analisadas essas três possibilidades para a escolha do método a ser utilizado com o FIJI.

A execução passo-a-passo refere-se às instruções em código de máquina, onde um comando em linguagem de alto nível pode corresponder em várias instruções em código de máquina. As principais desvantagens deste método residem na dificuldade de definição dos pontos de injeção (pois a escolha de um ponto incorreto pode causar resultados imprevisíveis) e no alto custo computacional envolvido (cada execução e retorno ao injetor custaria seis trocas de contexto).

Esta dificuldade na definição dos pontos de injeção exige uma prévia análise do código objeto da aplicação a ser executada, e entender um código em linguagem de máquina para definir os pontos em que seria interessante injetar um erro é uma tarefa árdua. Ainda que este esforço fosse realizado, ele estaria intimamente ligado com a aplicação que foi estudada – limitando a utilização de outras aplicações.

A segunda alternativa, de execução até um ponto de parada, além das dificuldades existentes na execução passo-a-passo ainda exige que façam-se alterações ou no arquivo objeto, ou no código do processo já em memória para a inserção de *breakpoints*. A inserção de um ponto de parada no local incorreto pode acarretar problemas ao funcionamento da aplicação, ou mesmo impedir o seu funcionamento.

Torna-se necessário um estudo ainda mais minucioso do código da aplicação, aumentando o tempo gasto em etapas anteriores à validação em si. Um outro agravante para a segunda alternativa é que nem sempre o código-fonte da aplicação está disponível. Mas apesar disto, esta alternativa tem um custo computacional menor do que a execução passo-a-passo.

Enfim, executar uma aplicação até que fosse encontrada uma chamada de sistema revelou-se como sendo a alternativa mais interessante – devido, em grande parte, à

arquitetura do Linux. Todo o acesso a recursos do sistema e *hardware* é executado via chamadas de sistema. Assim, pode-se alterar ou adulterar o comportamento e resposta das funções para gravação de arquivos.

6.2 Modelo de falhas

Para a realização de um experimento de injeção de falhas, faz-se necessária a definição de um modelo de falhas. Um estudo do modelo de falhas pode ajudar a aprimorar o processo de desenvolvimento de *software* e prevenir a ocorrência destas.

Um modelo de falhas permite a repetição de um experimento quantas vezes forem necessárias para atingir o objetivo. Para este caso, o objetivo é o de detectar erros ocorridos durante a utilização de um sistema de arquivos baseado em *journaling.*

Existe uma relação estrita entre o modelo de falhas e a implementação das técnicas de tolerância a falhas. Considerando o vínculo de conseqüência entre falhas e erros e a relação entre erros e forma de detecção [LAP 92] (visto que a detecção percebe erros, e não falhas), pode-se perceber a importância da escolha adequada do modelo de falhas para a implementação dos procedimentos de detecção de erros ou testes.

Para a definição do modelo de falhas a ser adotado nesta dissertação foi realizado um levantamento de resultados obtidos por outros pesquisadores avaliando os tipos de falhas que podem ocorrer. A maioria dos mecanismos considera as falhas de *hardware* transientes (mais especificamente de falta de energia), onde assume-se um modelo de colapso para o sistema. Isto representa uma situação em que o sistema de arquivos estaria sendo utilizado e ocorreria uma falha, fazendo com que a máquina seja reiniciada sem que ocorresse um *flush* das informações em memória (tais como *buffers* e *cache*) ou das transações pendentes armazenadas em *log*.

Para os sistemas de arquivos Linux, as falhas de colapso normalmente são resolvidas pelo utilitário *fsck* [BAR 2000]. Conforme é descrito no capítulo 4, o *fsck* realiza uma série de testes para validar o conteúdo de uma unidade de disco e assegurar-se de que ela não contém inconsistências. Entretanto, os problemas com o utilitário *fsck* são de que ele pode consumir um tempo extremamente elevado para unidades de disco com grande capacidade, e também que de acordo com a severidade dos problemas encontrados em disco pode ser necessário a presença do administrador do sistema para informar manualmente a senha de usuário *root* [TAN 97].

Este modelo de falhas define um conjunto de falhas que serão geradas por meio da ferramenta FIJI: ignorar o *log* de transações e a interceptar a manipulação das informações em disco.

6.3 Metodologia de injeção

Buscou-se uma metodologia de injeção de falhas que fosse simples e prática. Simples para que fosse possível reproduzir experimentos sem a necessidade de novos estudos. E prática para que fosse possível iniciar um experimento sem que fosse necessário mais tempo para idealizá-lo do que para realizá-lo.

O FIJI possui um princípio de funcionamento bastante simples: executar um processo durante um tempo e, a partir de um determinado instante, primeiramente descartar todo o conteúdo do *log* do *journaling*. A partir deste momento, quando for encontrada uma chamada de sistema *write* ou *read* deve-se fazer com que o efeito da mesma seja descartado zerando a quantidade de *bytes* a serem manipulados.

A metodologia empregada pelo FIJI não é intrusiva no código da aplicação alvo nem no código do sistema operacional. A intrusão desta ferramenta reside apenas no tempo de execução da aplicação alvo.

6.4 Plataforma de *hardware*

O ambiente de *hardware* constitui-se de uma máquina do tipo PC compatível com microprocessador K6-2 de 450 MHz, e 320 Mb de memória RAM. Ela possui unidades de disco com capacidades de 40 e 6 Gb, identificadas como hda e hdc, respectivamente.

A placa-mãe desta máquina é um modelo SIS598 da fabricante *PC-Chips*, e possui 2 (duas) controladoras de disco IDE que são *on-board* e permitem transferência de dados do tipo PIO nos modos 0 a 4. O modo de transferência que foi detectado pela BIOS na placa-mãe da máquina, para as unidades de disco hda e hdc, foi 4. Além disso, as freqüências de operação podem ser do tipos *bus mastering* e *UltraDMA 33/66*.

Esta máquina encontra-se conectada permanentemente à Internet por meio de uma placa de rede Ethernet, de 10/100 Mbps.

6.5 Plataforma de *software*

Nesta seção, encontram-se descritos os elementos de *software* utilizados para o experimento de injeção de falhas.

6.5.1 Sistema alvo

O sistema alvo é uma máquina baseada na distribuição Red Hat Linux em sua versão 9.0. Ela foi configurada para utilizar a versão 1.3.1 do sistema de arquivos XFS para Linux [XFS 2003]. A escolha da distribuição Red Hat Linux 9.0 deve-se ao fato de que a própria SGI (autora do XFS) apenas disponibiliza os discos de instalação da versão 1.3.1 do XFS para uso com esta distribuição. Inicialmente, este trabalho utilizou uma versão 1.1 do XFS com a distribuição Red Hat Linux 8.0.

6.5.2 Monitoração e coleta de dados

Para a realização das tarefas de monitoração e coleta de dados durante a injeção de falhas, será empregado o utilitário *time*. Por meio do *time*, é possível determinar os tempos necessários para a execução de um comando.

6.5.3 Gerador de carga de trabalho

A carga de trabalho (ou *workload*) à qual um sistema de arquivos estará submetido é algo que depende do tipo de aplicação em que ele será utilizado. De uma maneira geral, observa-se um desempenho muito ruim para sistemas de arquivos que precisem lidar com transferências de dados muito pequenas junto às unidades de disco - ainda mais se para isso ainda estiverem envolvidas várias operações de busca.

Isso acontece porque as operações de busca representam a maior parte no tempo total de acesso à disco - nas operações de escrita a eficiência é algo em torno de 10% [SEL 92]. Para que o tempo de acesso a disco seja ótimo, é preciso que as operações sejam feitas de maneira seqüencial.

Infelizmente, na grande maioria dos casos os acessos feitos à disco são para realizar transferências de apenas alguns poucos *kilobytes* [SAT 81]. Essas operações resultam em uma série de acessos aleatórios, cada uma com seu tempo de acesso. Com as operações sendo feitas em vários arquivos pequenos, a maior parte dos tempos necessários para criação e remoção acaba sendo destinado às atualizações nos metadados dos sistemas de arquivos.

Sendo assim, procurando gerar cargas de trabalho que correspondessem àquelas existentes em ambientes é utilizada a ferramenta *bonnie++* [BON 2003]. O *bonnie++* pode ser utilizado tanto para a geração de um único arquivo de tamanho grande (mais de 700 Mb, no mínimo) como também para a geração de vários arquivos de tamanho aleatório distribuídos ao longo de uma árvore de diretórios.

A ferramenta *bonnie* faz uma medição das taxas com que são feitas operações de leitura e escrita de informações de um arquivo em disco utilizando:

- As funções putc e getc;
- Operações supostamente "eficientes" para manipulação de blocos de dados; e
- Reescrita de um arquivo existente.

O *bonnie* faz a criação de 4 processos-filho, os quais fazem uma busca aleatória dentro do arquivo recém-criado, lêem o conteúdo dos blocos encontrados, e reescreve 10% das informações obtidas nas operações de leitura. A taxa em que estes passos são realizados é medida em operações por segundo. Em todos os casos, o *bonnie* também informa o uso de CPU em cada um dos testes.

Entretanto, a ferramenta selecionada foi o *bonnie++* - a qual é o mesmo *bonnie* descrito anteriormente, mas reescrito em C++ para incorporar as seguintes melhorias:

- Suporte a unidades maiores do que 2Gb usando múltiplos arquivos
- Inclusão de testes para indicar a taxa na qual o sistema de arquivos pode criar um grande número de arquivos

O *workload* gerado não corresponde a nenhum perfil específico – tal como acontece em outras ferramentas para geração de carga. As operações geradas não mantém qualquer relação entre si.

6.6 Arquitetura da ferramenta FIJI

Em função da arquitetura genérica formulada por Hsueh [HSU 97], elaborou-se a arquitetura do FIJI representada na Figura 6.1:

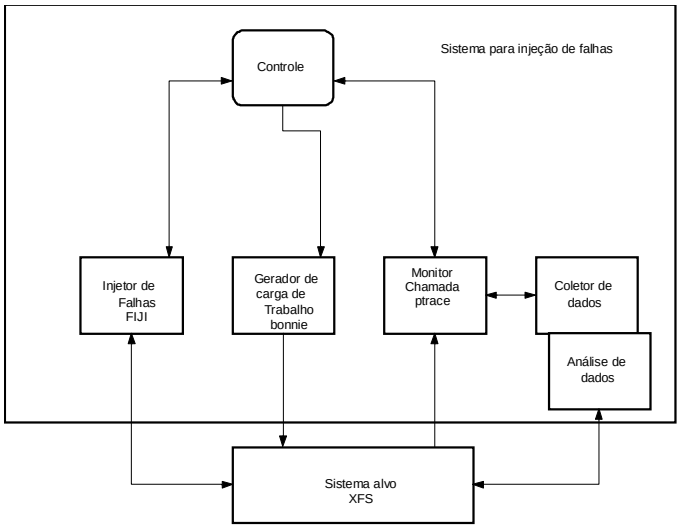

Figura 6.1: Arquitetura do FIJI

Por haver sido criado para trabalhar com *workloads* gerados por ferramentas diversas, o FIJI não possui um gerador de carga de trabalho. O *kernel* monitora a execução do gerador de carga de trabalho, em função de requisições do FIJI. O controle pede que o gerador de carga de trabalho seja executado até que surja a entrada ou saída de uma chamada de sistema.

O monitor do *kernel* nada mais é do que a chamada de sistema *ptrace*. Quando é encontrada uma chamada de sistema, o monitor do *kernel* pára o gerador de carga de trabalho e avisa o FIJI. O monitor recebe um sinal indicando que o gerador de carga de trabalho encontrou uma chamada de sistema, e requisita informações sobre a chamada de sistema em questão. Recebidas estas informações, o monitor verifica se a chamada de sistema é a *write* e, caso seja, o monitor encerra a sua execução e o fluxo desvia-se para o controle.

A biblioteca de falhas está representada pelas ações de omitir os efeitos da chamadas de sistema *read* e *write*, por meio da mudança do parâmetro que informa a quantidade de *bytes* a serem gravados (registrador EDX). São feitos dois testes: as chamadas de sistema read e write serão omitidas se já tiver transcorrido o tempo de espera especificado pelo parâmetro –C e se o tipo de *syscall* (contido no registrador EAX) for igual à 04 (correspondente ao *write*) ou for igual à 03 (*write*).

Durante a execução, o FIJI gera um relato das injeções realizadas - com estatísticas a respeito das injeções de falhas ocorridas. Estas informações caracterizam o coletor de dados do FIJI.

6.7 Funcionamento do FIJI

A Figura 6.2 contém as opções disponíveis para execução do FIJI. Um parâmetro deve ser fornecido ao FIJI pela linha de comando, informando qual o gerador de carga de trabalho a ser utilizado.

```
[root@localhost root]# ./fiji
usage: fiji [-dffhiqrtttTvVxx] [-C val] [-a column] [-e expr] [-o
file] [-p pid] ... [command [arg ...]]
   or: fiji -c [-e expr] ... [-O overhead] [-C val] [-S sortby] [-E
var=val] ... [command [arg ...]]
-C -- time to wait (in seconds) before start the dropping of
read/write syscalls
-c -- count time, calls, and errors for each syscall and report
summary
-f -- follow forks, -ff -- with output into separate files
-F -- attempt to follow vforks,
-h -- print help message
-i -- print instruction pointer at time of syscall
-q -- suppress messages about attaching, detaching, etc.
-r -- print relative timestamp,
-t -- absolute timestamp,
-tt -- with usecs
-T -- print time spent in each syscall,
-V -- print version
-v -- verbose mode: print unabbreviated argv, stat, termio[s], etc.
args
-x -- print non-ascii strings in hex,
-xx -- print all strings in hex
-a column -- alignment COLUMN for printing syscall results (default
40)
-e expr -- a qualifying expression: option=[!]all or options:
trace, abbrev, verbose, raw, signal, read, or write
-o file -- send trace output to FILE instead of stderr
-O overhead -- set overhead for tracing syscalls to OVERHEAD usecs
-p pid -- trace process with process id PID, may be repeated
-s strsize -- limit length of print strings to STRSIZE chars
(default 32)
-S sortby -- sort syscall counts by: time, calls, name, nothing
(default time)
```

Figura 6.2: Opções disponíveis para uso com o FIJI

Este gerador de carga de trabalho é executado pelo próprio FIJI, que na inicialização envia um sinal (por meio da função *ptrace*) indicando que ele deverá estar sob depuração. Baseando-se no modo de execução até uma chamada de sistema, o FIJI executa a aplicação durante uma certa quantidade de tempo sem fazer nada.

Entretanto, no momento em que transcorrer a quantidade de segundos informada como parâmetro pela cláusula -C, todo o conteúdo do *log* utilizado pelo *journaling* será descartado. Também a partir deste instante, deverá ser feita a interceptação de todas as chamadas de sistema *read* e *write*. Espera-se, com isto, comprometer a integridade do sistema de arquivos.

Como padrão, o FIJI imprime a data e hora atuais com precisão de milisegundos. A cada chamada de sistema que for executada, é gerada uma nova linha contendo:

- *timestamp* desta requisição;
- nome da chamada de sistema;
- os parâmetros com os quais ela foi executada; e
- resultado retornado pela mesma.

O FIJI foi construído com base no FIDe e no utilitário *strace*. Ele utilizou a mesma arquitetura proposta para o FIDe, mas com as limitações de conter o cenário de falhas embutido dentro da própria ferramenta.

6.7.1 Nível de abstração

Por manipular as requisições do gerador de carga de trabalho executadas por chamadas de sistema, o FIJI localiza-se entre a aplicação e o sistema operacional. A injeção de falhas começa a ocorrer depois de um certo tempo, apenas na entrada das chamadas de sistema *read* e *write* – zerando a quantidade de *bytes* lidos ou gravados.

A injeção dos erros é bastante pontual. Em outras palavras, pode-se ter certeza da propagação de uma falha injetada e de sua forma de manifestação. Na verdade, tem-se o controle sobre a forma de propagação de uma determinada falha, sobre o erro injetado, a latência dos erros e mesmo sobre o defeito percebido no caso de não-tratamento do erro.

Entende-se melhor os diferentes níveis de abstração entre processos e chamadas de sistema por meio da Figura 6.3 - adaptada com base nas iterações do FIDe [GON 2001]. O processo de usuário chama uma função *write*, preenchendo corretamente seus parâmetros. Esta função é parte da Glibc, e pode estar ligada ao processo de forma estática ou dinâmica.

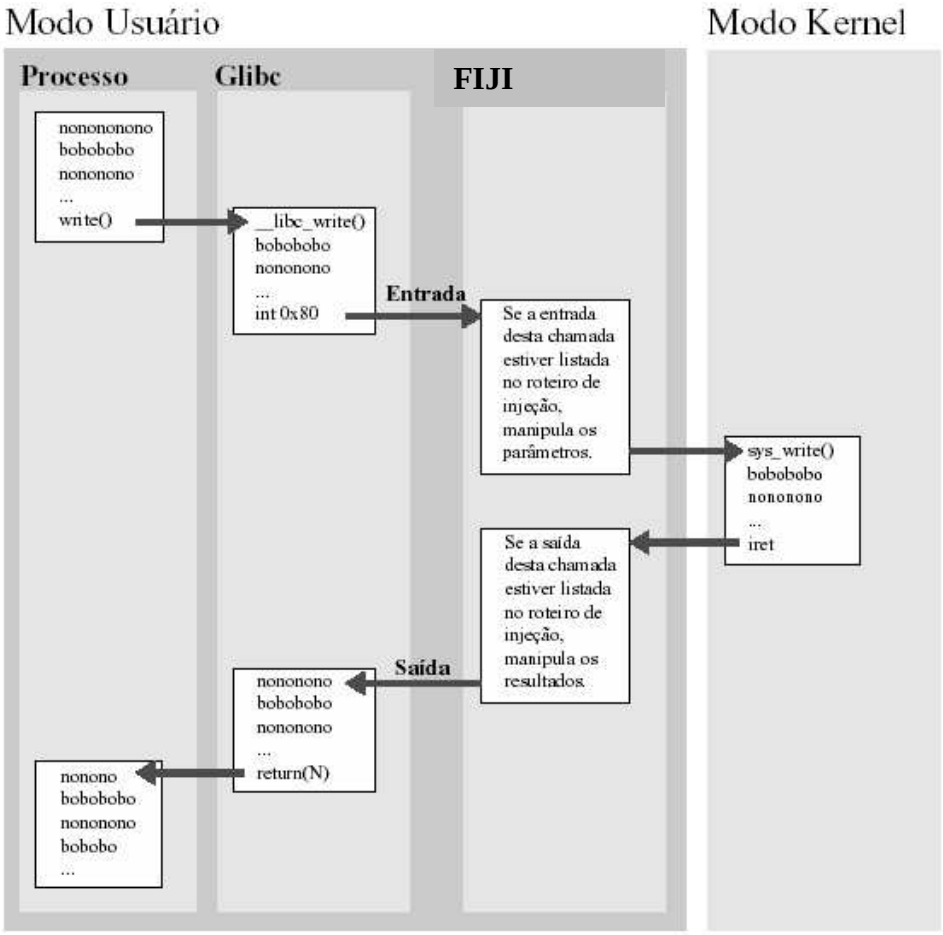

Figura 6.3: Iteração entre processo, Glibc e chamadas de sistema

O FIJI foi idealizado para validação das técnicas de tolerância a falhas utilizadas em *software*. Não pode, dado o nível de abstração em que se encontra, injetar falhas no *hardware* ou mesmo no sistema operacional.

6.7.2 Intrusividade

A intrusividade é uma característica indesejada em experimentos para injeção de falhas. Assim como um amperímetro ao ser colocado em série no circuito introduz uma pequena resistência, um injetor de falhas com uma intrusão muito grande no sistema pode mascarar erros e defeitos. Quanto menor a intrusão gerada pelo injetor, tanto melhores e mais corretos serão os resultados obtidos [LEI 2000][GON 2001].

Como já foi mencionado anteriormente, a intrusão do FIJI reside no tempo de execução. Esta intrusão acontece em dois níveis: na carga do sistema, como um todo, e no tempo de execução da aplicação alvo.

O FIJI é implementado como um processo concorrente à aplicação alvo e, portanto, a carga do sistema é alterada. Apesar de pequeno em tamanho e em consumo de recursos, a simples presença do FIJI no sistema pode crescer em função da política de escalonamento de processos em uso.

Dependendo do poder computacional do computador no qual é realizado o teste, esta presença pode ser considerada desprezível. Nos testes realizados, com os resultados descritos logo adiante, a presença do FIJI na carga do sistema pôde ser considerada pequena o suficiente para ser desprezada.

Além do tempo de execução do FIJI – no qual o gerador de carga de trabalho está parado – acontecem ainda seis trocas de contexto, na operação descrita a seguir. A passagem de um item para outro, nesta lista, significa uma mudança de contexto [GON 2001].

1. FIJI pede ao sistema operacional que execute a aplicação até encontrar uma chamada de sistema;
2. sistema operacional prepara o ambiente (realiza o chaveamento de contexto e inicializa o *flag* de depuração) e começa a executar a aplicação alvo;
3. A aplicação alvo executa até encontrar uma chamada de sistema, momento em que pára e devolve o controle ao sistema operacional;
4. sistema operacional prepara o ambiente e passa a executar o FIJI, avisando que a aplicação alvo requisitou uma chamada de sistema;
5. FIJI pede para o sistema operacional que leia o valor dos registradores da aplicação alvo;
6. sistema operacional executa a leitura e devolve o controle ao FIJI; e
7. FIJI recebe os dados e verifica se esta chamada de sistema deve ser observada ou não.

Este processo todo acontece sempre que encontra-se uma chamada de sistema – seja ela qual for. E quando ocorre uma chamada de sistema *write*, tem-se mais duas trocas de contexto.

Ainda que de forma simplista, mas com propósitos didáticos, poderia-se mensurar esta intrusão em três casos extremos [GON 2001]:

- Um programa de computação intensiva: neste caso, poderia-se utilizar um programa que fizesse inúmeros cálculos e manipulações de dados em memória. Com isso, haveria muito pouco acesso a recursos do sistema;
- Um programa que acesse intensivamente recursos do sistema: um programa que crie outros processos, que leia a data e hora do sistema, aloque e libere

porções de memória e realize outras operações que não sejam de entrada e saída; e

- Um programa que realize acesso intenso a disco: um programa que realize várias operações de entrada e saída, via sistema operacional.

No primeiro caso existiriam poucas chamadas de sistema, e a presença do FIJI seria mascarada pelo tempo gasto na computação de cálculos e manipulação de informações na memória. A Tabela 6.1 contém os resultados da execução deste programa, considerando a execução com e sem o FIJI.

Tabela 6.1: Intrusão no caso 1

Caso	Tempo
Caso 1	1,730s
Caso 1 + FIJI	1,884s

Este programa foi executado 20 vezes, e foram coletados os tempos de execução por meio do utilitário *time*. Destas medições, foi calculado o tempo médio para este caso. A listagem do programa utilizado para testar este primeiro caso está na Figura 6.4 [GON 2001].

```
main()
{
      int i,j;
      long r = 0, s;

      for (j=0; j < 10000; j++) {
            for (i=0; i < 1000; i++) {
                  r += i*31;
                  s = r ^ j;
                  r = s & (i + j) % 97;
                  s = r + 271;
                  r ^= s + j % 13;
            }
      }
}
```

Figura 6.4: Listagem do programa para testar intrusão – caso 1

O programa utilizado para o segundo caso está na Figura 6.5 [GON 2001]:

```
#include <sys/types.h>
#include <unistd.h>
#include <time.h>
#include <stdlib.h>
#include <sched.h>
#include <sys/resource.h>

main()
{
      int i;
      char * ptr;

      for (i=0; i < 10000; i++) {
            getpid();
            getppid();
            time(NULL);
            ptr = (char *) malloc(4000);
            getuid();
            geteuid();
            getgid;
            getegid();
            free(ptr);
            getpriority(PRIO_PROCESS,0);
            sched_getscheduler(0);
      }
      return(0);
}
```

Figura 6.5: Listagem do programa para gerar intrusão – caso 2

Neste segundo, e pior caso, existem muitas chamadas de sistema e a intrusão do FIJI não tem como ser mascarada. O segundo programa realiza vários acessos a recursos e informações do sistema operacional, porém, não realiza operações de entrada e saída. Este é o pior caso no tocante à intrusão, pois aqui ocorrem muitas chamadas de sistema e todas são rápidas o suficiente a ponto de que perceba-se uma intrusão gerada pelos múltiplos chaveamentos de contexto.

A Tabela 6.2 contém os resultados da execução deste programa, considerando a execução com e sem o FIJI. Este programa foi executado 20 vezes, e foram coletados os tempos de execução por meio do utilitário *time*. Destas medições, foi calculado o tempo médio para este caso.

Tabela 6.2: Intrusão no caso 2

Caso	Tempo
Caso 2	2,222s
Caso 2 + FIJI	6m13,638s

O terceiro caso é mais ameno em relação ao segundo, pois os tempos de chaveamento são diluídos pelo tempo das operações de entrada e saída. O terceiro programa realiza múltiplas chamadas de sistema, aqui caracterizadas por operações de entrada e saída.

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main()
{
      int fd;
      int fd2;
      int i,c,j;
      char msg1[20], msg2[20];

      for (j=0; j < 20; j++) {
            if ((fd=open("arq_test.txt", O_SYNC | O_RDWR | O_CREAT
| O_TRUNC, S_IRWXU | S_IRWXG | S_IROTH)) < 0) {
                  printf("Erro ao abrir o arquivo\n");
                  exit(1);
            }

            if ((fd2=open("/dev/random", O_RDONLY)) < 0) {
                  printf("Erro ao abrir o arquivo\n");
                  exit(1);
            }

            for (i=0; i < 5; i++) {
                  read(fd2, msg1, 4); write(fd, msg1, 4);
                  read(fd2, msg2, 4); write(fd, msg2, 4);
            }

            close(fd);
            close(fd2);
      }
      return(0);
}
```

Figura 6.6: Listagem do programa para gerar intrusão – caso 3

A Tabela 6.3 contém os resultados da execução deste programa, considerando a execução com e sem o FIJI. Este programa foi executado 20 vezes, e foram coletados os tempos de execução por meio do utilitário *time*. Destas medições, foi calculado o tempo médio para este caso

Tabela 6.3: Intrusão no caso 3

Caso	Tempo
Caso 3	1,877s
Caso 3 + FIJI	2,499s

Apesar de existir uma grande intrusão gerada, as esperas forçadas por operações de entrada e saída são grandes o suficiente para mascarar quase que na totalidade a intrusão gerada. A listagem do programa utilizado para testar o terceiro caso está na Figura 6.6 [GON 2001].

A intrusão causada pelo FIJI, entre todos os casos considerados, pode ser analisada por meio da Tabela 6.4. Por meio dela, conclui-se que no primeiro caso existem poucas chamadas de sistema e a presença do injetor pode ser mascarada pelo

tempo gasto na computação de cálculos e manipulação de informações na memória. No segundo caso, o pior caso, existem muitas chamadas de sistema e a intrusão do injetor não tem como ser mascarada. O terceiro caso é mais ameno do que o segundo, pois os tempos de chaveamento de contexto são diluídos pelo tempo das operações de I/O.

Tabela 6.4: Intrusão do FIJI

Caso	Tempo
Caso 1	1,730s
Caso 1 + FIJI	1,884s
Caso 2	2,222s
Caso 2 + FIJI	6m13,638s
Caso 3	1,877s
Caso 3 + FIJI	2,499s

6.8 Detalhes de implementação

A partir do momento em que o FIJI é executado, ele cria uma cópia de si mesmo via função *fork*. Neste novo processo gerado (processo filho) [TAN 97] ativa-se o *flag* de depuração chamando-se a função *ptrace* com o parâmetro PTRACE_TRACEME. A seguir, o gerador de carga de trabalho é executado por meio da função *execve*.

Em função de ter o *flag* de depuração ativo, na primeira chamada de sistema encontrada o processo filho pára e sinaliza o FIJI. Enquanto isso, o FIJI espera que o processo filho (já executando comandos) indique estar pronto para ser depurado. A espera é realizada com o auxílio da função *wait4*.

O FIJI executa uma requisição ao sistema operacional via função *ptrace* e espera, via função *wait4*, que o gerador de carga de trabalho sinalize o fim da execução. O sinal enviado, neste caso, é o SIGTRAP.

Após receber o sinal de que o processo do gerador de carga de trabalho encontrou uma chamada de sistema e parou, o FIJI pode requisitar ao sistema operacional que leia ou escreva na memória e nos registradores. O FIJI recebe diretamente do processo filho a indicação de que este está parado.

Agora, o FIJI pode inspecionar e manipular a memória do processo, o valor de seus registradores e a sua estrutura *user*. Isto é feito executando-se a chamada da função *ptrace* com o parâmetro PTRACE_GETREGS.

Por meio da chamada da função *ptrace* com o parâmetro PTRACE_GETREGS, é possível obter o valor de todos os elementos descritos na Figura 6.7. Estes elementos correspondem aos registradores de dados e programa da arquitetura i386 [SIE 93]

```
struct user_regs_struct {
      long ebx, ecx, edx, esi, edi, ebp, eax;
      unsigned short ds, __ds, es, __es;
      unsigned short fs, __fs, gs, __gs;
      long orig_eax, eip;
      unsigned short cs, __cs;
      long eflags, esp;
      unsigned short ss, __ss;
};
```

Figura 6.7: Elementos obtidos por PTRACE_GETREGS

Assim que o FIJI lê o valor dos registradores, ele faz um teste para verificar se o conteúdo do registrador EAX é 03 (o que corresponde a uma chamada de sistema *read*) ou 04 (o que corresponde a uma chamada de sistema *write*) [BAR 2000]. Caso seja uma delas, ele modifica o conteúdo do registrador EDX. Depois disso, executa novamente a função *ptrace* com o parâmetro PTRACE_SETREGS para zerar o registrador EDX e continuar executando a *syscall* com este parâmetro modificado.

6.8.1 Iterações com o *kernel*

A Figura 6.8 mostra este processo de comunicação entre o FIJI, o kernel e o gerador de carga de trabalho. Cada seta indica uma mudança de contexto. Cada mudança de contexto significa uma parcela de atraso na execução da aplicação alvo.

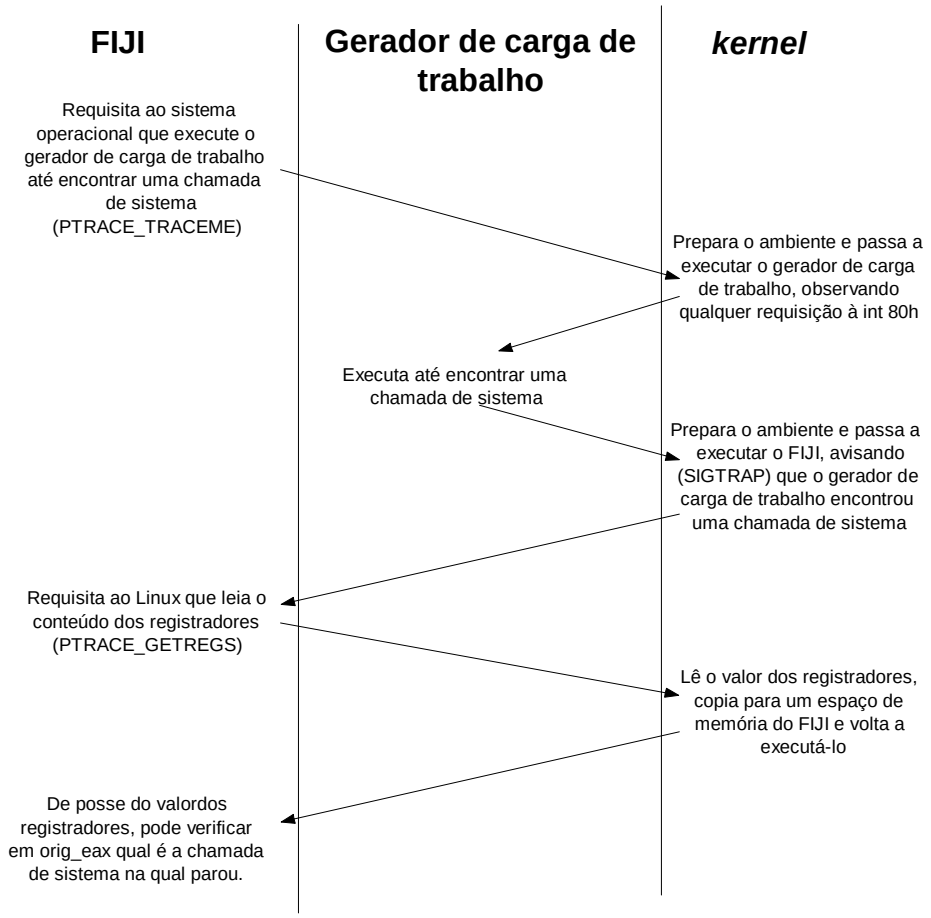

Figura 6.8: Busca da próxima chamada de sistema

Na Figura 6.8, o registrador referenciado é orig_eax. O identificador da chamada de sistema é o registrador EAX, porém, a estrutura de registradores fornecida pela função *ptrace* contém dois valores de EAX. O último valor assumido pelo registrador antes da parada (EAX) e o valor deste no momento da chamada de sistema (orig_eax). Para o FIJI, o valor que realmente importa é o de orig_eax [GON 2001].

A injeção de falhas será feita pelo FIJI considerando se:

1. Já se passou a quantidade de segundos que foi especificada pelo parâmetro –C em relação ao momento em que o experimento iniciou-se; e
2. Se o conteúdo de orig_eax é 3 ou 4.

Caso estas condições confirmem-se, o valor do registrador EDX é zerado no *buffer* para onde foi direcionada a saída da função ptrace com o parâmetro PTRACE_GETREGS. Depois disso, será feita uma chamada da função ptrace com o parâmetro PTRACE_SETREGS. Esta transição está na Figura 6.9

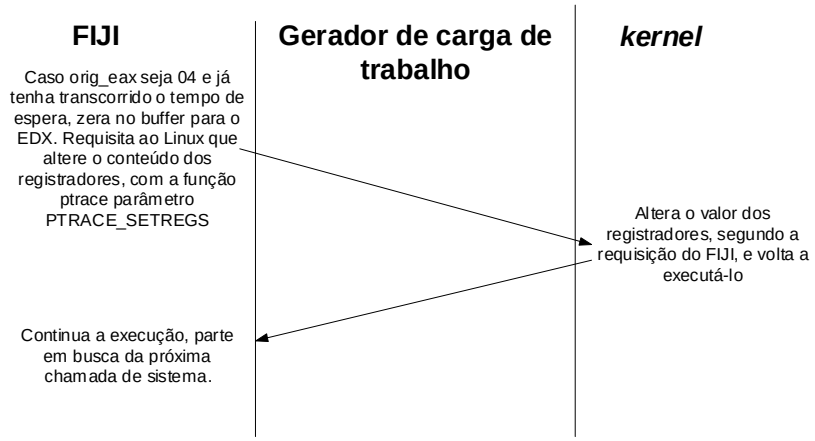

Figura 6.9: Injeção de falha pelo FIJI

6.9 Resumo

Este capítulo apresentou uma descrição sobre o modelo de falhas considerado para a criação do FIJI, bem como o modo como isto é implementado na ferramenta. O ambiente para injeção de falhas é descrito em detalhes, e apresentam-se resultados da execução do FIJI em diferentes cenários visando avaliar a sua intrusividade.

7 TESTES E EXPERIMENTOS PARA INJEÇÃO DE FALHAS

Este capítulo descreve os resultados obtidos durante a realização de testes e experimentos para injeção de falhas no sistema de arquivos XFS.

7.1 Descrição do experimento manual

Será feita uma injeção de falhas manual no sistema de arquivos, visando determinar a ordem de grandeza na diferença entre tempo de recuperação para o sistema de arquivos XFS (baseado em *journaling*) e o tempo de recuperação para um sistema de arquivos tradicional (o ext2). Para isso, utiliza-se a ferramenta para geração de carga de trabalho *bonnie++*.

Espera-se, por meio da ferramenta de monitoração e coleta de dados *time*, determinar o tempo de recuperação para dois sistemas de arquivos de mesmo tamanho - sob o qual é aplicada uma mesma carga de trabalho..

O experimente é realizado no equipamento descrito na seção 6.4, mas com apenas 64 Mb de memória RAM. Isto foi feito para que os efeitos dos *caches/buffers* do Linux não afetem o experimento. A arquitetura a ser empregada é uma variante daquela que encontra-se descrita na Figura 6.1. Ela segue na Figura 7.1, e foi organizada considerando que:

- O controle é representado por um operador humano;
- as funções de biblioteca para o injetor de falhas e injetor de falhas são a tecla de *Reset* da máquina alvo*;*
- as funções de biblioteca para geração de carga de trabalho e gerador de carga de trabalho são representadas pelo *bonnie++;*
- as funções de coleta e monitoração são representadas pelo utilitário *time; e*

- o alvo da injeção de falhas é uma máquina que conterá uma partição Linux (ou XFS, ou ext2).

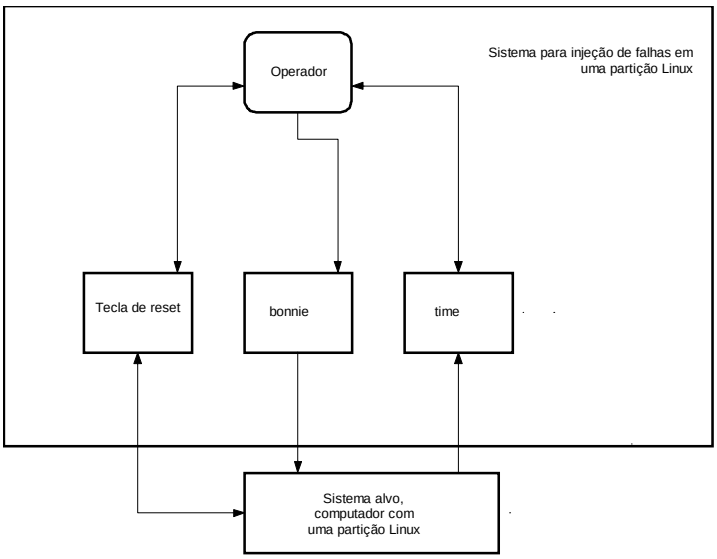

Figura 7.1: Ambiente para injeção manual de falhas

7.2 Sistema de arquivos baseado em blocos – ext2

Segundo a arquitetura para injeção de falhas proposta por Hsueh [HSU 97], temos a função de controle representada pelo operador humano. A função de injetor de carga de trabalho e biblioteca para geração de carga de trabalho é representada pelo utilitário *bonnie++*. O utilitário *time* é o monitor do sistema alvo, e o sistema alvo é uma partição ext2. O injetor de falhas é a tecla de *Reset* desta máquina.

7.2.1 Um único arquivo grande – ext2

Primeiramente foi criada uma partição de 1.5 Gb do tipo 83 (*Linux native*) no disco de 40 Gb. Logo após, ela foi formatada por meio do utilitário mkfs.ext2 e montada no diretório /mnt/teste por meio do comando **mount –t ext2 /dev/hda3 /mnt/teste**.

O experimento consistiu em realizar vários testes em que é gerada uma carga de trabalho através do utilitário *bonnie++*, por meio do comando **bonnie++ –d /mnt/teste –s 130**. Isso é algo que faz com que o *bonnie++* utilize a partição /mnt/teste para a criação do seu arquivo de *scratch*. O parâmetro –s indica que o tamanho para este arquivo de *scratch* é de 130 Mb.

Assim que a execução do comando é concluída e o *prompt* de comando é exibido, pressiona-se a tecla de *Reset* no computador alvo. Este experimento foi coordenado através do operador humano, na função de controle. Com isso, não houve um desligamento normal do sistema e uma série de metadados não foi atualizada para a partição /mnt/teste. Assim, logo que a máquina fosse reiniciada seria necessário tornar o sistema de arquivos consistente.

Teremos que o tempo necessário para tornar o sistema de arquivos consistente (ou T_0) será o tempo necessário para a execução do utilitário *fsck* nesta partição. Logo, executando-se o comando **time *fsck* –y /dev/hda3** obtém-se um valor que será o T_0 desta partição. Assume-se que o comando *fsck* deve ser executado com o parâmetro *–y*

para que não seja feito nenhum pedido de confirmação ao operador responsável. Em um experimento como este, o tempo que seria necessário para exibir uma tela de confirmação ao usuário *root* e ele interpretá-la para responder *yes* ou *no* afetaria em muito as medições. Por isso, presume-se que apenas a execução do utilitário *fsck* com o parâmetro *–y* consegue tornar a partição consistente.

Após uma série de 30 testes, foram obtidas as informações constantes na Tabela 7.1. Por meio dela, determinou-se o tempo médio que é necessário para tornar este sistema de arquivos consistentes através do *fsck*.

Tabela 7.1: Tempo necessário para tornar uma partição ext2 consistente

Iteração	Tempo necessário para tornar o sistema de arquivos consistente
1	2,585s
2	2,532s
3	2,550s
4	2,561s
5	2,578s
6	2,524s
7	2,555s
8	2,541s
9	2,567s
10	2,540s
11	2,572s
12	2,549s
13	2,531s
14	2,547s
15	2,544s
16	2,601s
17	2,513s
18	2,545s
19	2,591s
20	2,516s
21	2,604s
22	2,527s
23	2,501s
24	2,536s
25	2,549s
26	2,552s
27	2,594s
28	2,586s
29	2,564s
30	2,543s

Por meio da segunda coluna da Tabela 7.1, é possível determinar o tempo médio de reparo. Com base nestas 30 medições, este tempo é de 2,553267 segundos com um desvio-padrão de 0,026621 segundos. Com este cenário, o intervalo de confiança será de 2,509478806 e 2,597054528 para uma probabilidade de 0,95.

Quanto à cobertura de falhas, esta será de 100% - haja visto que a falha injetada pressionando-se a tecla *Reset* foi detectada durante a execução do *fsck*. No momento em que a máquina foi reiniciada, o *fsck* indicou que haviam erros a serem corrigidos na unidade de disco.

7.2.2 Milhares de arquivos com tamanho aleatório – ext2

Primeiramente foi criada uma partição de 24 Gb do tipo 83 (*Linux native*) no disco de 40 Gb. Logo após, ela foi formatada por meio do utilitário mkfs.ext2 e montada no diretório /mnt/teste por meio do comando **mount –t ext2 /dev/hda3 /mnt/teste**.

O experimento consistiu em realizar vários testes em que é gerada uma carga de trabalho através do utilitário *bonnie++*, por meio do comando **bonnie –d /mnt/teste –s 700 –u root –f –n 2:100000:1:5** Isso é algo que faz com que o *bonnie++* utilize a partição /mnt/teste para a criação do seu arquivo de *scratch.* O parâmetro –s indica que o tamanho para este arquivo de *scratch* é de 700 Mb. Depois disso, ele irá criar 5 subdiretórios e, dentro deles, criar 2.048 arquivos de tamanho aleatório entre 0 a 100.000 *bytes*.

Assim que a execução do comando é concluída e o *prompt* de comando é exibido, pressiona-se a tecla de *Reset* no computador alvo. Este experimento foi coordenado através do operador humano, na função de controle. Com isso, não houve um desligamento normal do sistema e uma série de metadados não foi atualizada para a partição /mnt/teste. Assim, logo que a máquina fosse reiniciada seria necessário tornar o sistema de arquivos consistente.

Teremos que o tempo necessário para tornar o sistema de arquivos consistente (ou T_0) será o tempo necessário para a execução do utilitário *fsck* nesta partição. Logo, executando-se o comando **time *fsck* –y /dev/hda3** obtém-se um valor que será o T_0 desta partição. Assume-se que o comando *fsck* deve ser executado com o parâmetro *–y* para que não seja feito nenhum pedido de confirmação ao operador responsável. Em um experimento como este, o tempo que seria necessário para exibir uma tela de confirmação ao usuário *root* e ele interpretá-la para responder *yes* ou *no* afetaria em muito as medições. Por isso, presume-se que apenas a execução do utilitário *fsck* com o parâmetro *–y* consegue tornar a partição consistente.

Após uma série de 30 testes, determinou-se o tempo médio que é necessário para tornar este sistema de arquivos consistentes através do *fsck*. É possível determinar o tempo médio de reparo, que é de 1min e 30,247s segundos com um desvio-padrão de 0,01821 segundos.

Quanto à cobertura de falhas, esta será de 100% - haja visto que a falha injetada pressionando-se a tecla *Reset* foi detectada durante a execução do *fsck.* No momento em que a máquina foi reiniciada, o *fsck* indicou que haviam erros a serem corrigidos na unidade de disco.

7.3 Sistema de arquivos baseado em *journaling* - XFS

Neste cenário, o sistema alvo é uma partição XFS.

7.3.1 Um único arquivo grande - XFS

Primeiramente foi criada uma partição de 1.5 Gb do tipo 83 (*Linux native*) no disco de 40 Gb. Logo após, ela foi formatada por meio do utilitário mkfs.xfs e montada no diretório /mnt/teste por meio do comando **mount –t xfs /dev/hda3 /mnt/teste**.

O experimento consistiu em realizar vários testes em que é gerada uma carga de trabalho através do utilitário *bonnie++*, por meio do comando **bonnie++ –d /mnt/teste –s 130**. Isso é algo que faz com que o *bonnie++* utilize a partição /mnt/teste para a

criação do seu arquivo de *scratch*. O parâmetro –s indica que o tamanho para este arquivo de *scratch* é de 130 Mb.

Assim que a execução do comando é concluída e o *prompt* de comando é exibido, pressiona-se a tecla de *Reset* no computador alvo. Este experimento foi coordenado pelo operador humano, na função de controle. Com isso, não houve um desligamento normal do sistema e ficaram metadados a serem atualizados para a partição /mnt/teste. Assim, logo que a máquina fosse reiniciada seria necessário tornar o sistema de arquivos consistente.

Teremos que o tempo necessário para tornar o sistema de arquivos consistente (ou T_1) será o tempo necessário para a montagem, incorporando as transações que encontram-se descritas no *log* da partição. Executando-se o comando **time mount /dev/hda3 /mnt/teste**, obteve-se um valor que será o T_1 desta partição.

Entretanto, deste tempo que foi medido é necessário descontar aquilo que seria necessário para montar esta partição caso não houvesse a necessidade de incorporar mudança alguma do *log* do *journaling*. Logo, teremos na Figura 7.2 que:

T_j = Tempo para montagem, incorporando as mudanças no *log* do *journaling*
T_m= Tempo para montagem, quando não há mudanças no *log* do *journaling*
$T_1 = T_j - T_m$

Figura 7.2: Definição de T_1 como sendo a diferença entre dois tempos

Após uma série de 30 testes, foram obtidas as informações constantes na Tabela 7.2. Por meio dela, determinou-se que o tempo médio para montagem de uma partição para quando não há mudanças nos metadados é de 0,147 segundos, com um desvio padrão de 0,004979 segundos.

Tabela 7.2: Tempo necessário para montar uma partição xfs

Iteração	Tempo para montagem, quando não entradas no log do *journaling*	Tempo para montagem, quando há entradas no log do *journaling*
1	0,149s	0,356s
2	0,151s	0,285s
3	0,140s	0,271s
4	0,150s	0,279s
5	0,141s	0,269s
6	0,153s	0,378s
7	0,152s	0,327s
8	0,144s	0,288s
9	0,145s	0,276s
10	0,151s	0,283s
11	0,147s	0,291s
12	0,147s	0,310s
13	0,149s	0,349s

14	0,141s	0,297s
15	0,132s	0,284s
16	0,142s	0,322s
17	0,143s	0,306s
18	0,150s	0,287s
19	0,151s	0,291s
20	0,149s	0,318s
21	0,143s	0,277s
22	0,152s	0,286s
23	0,148s	0,295s
24	0,142s	0,309s
25	0,153s	0,317s
26	0,141s	0,326s
27	0,152s	0,298s
28	0,148s	0,357s
29	0,151s	0,342s
30	0,147s	0,361s

Por meio da terceira coluna da Tabela 7.2, é possível determinar o tempo médio de recuperação para quando o desligamento é feito através do botão de *Reset* (e, portanto, existem várias transações registradas em *log*). Quando isso acontece, o tempo necessário para montar o sistema de arquivos é de 0,308 segundos, com um desvio-padrão de 0,029977 segundos. Com isto temos que o tempo médio apenas para a tarefa de recuperação é a diferença entre essas medidas do tempo para a montagem (para quando há e para quando não há transações registradas no *log)*, com o valor de 0,161033 segundos. Para estes valores, o intervalo de confiança será de 0,35730776 e 0,258692242 para uma probabilidade de 0,95.

A cobertura de falhas será de 100%, haja visto que todas as falhas injetadas foram detectadas. Utilizou-se o comando **grep –i ´start recovery´ /var/log/dmesg** para determinar se houve recuperação durante a montagem do sistema de arquivos.

7.3.2 Milhares de arquivos com tamanho aleatório - XFS

Primeiramente foi criada uma partição de 24 Gb do tipo 83 (*Linux native*) no disco de 40 Gb. Logo após, ela foi formatada por meio do utilitário mkfs.xfs e montada no diretório /mnt/teste por meio do comando **mount –t xfs /dev/hda3 /mnt/teste**.

O experimento consistiu em realizar vários testes em que é gerada uma carga de trabalho através do utilitário *bonnie++*, por meio do comando **bonnie –d /mnt/teste –s 700 –u root –f –n 2:100000:1:5** Isso é algo que faz com que o *bonnie++* utilize a partição /mnt/teste para a criação do seu arquivo de *scratch.* O parâmetro –s indica que o tamanho para este arquivo de *scratch* é de 700 Mb. Depois disso, ele irá criar 5 subdiretórios e, dentro deles, criar 2.048 arquivos de tamanho aleatório entre 0 a 100.000 *bytes*.

Assim que a execução do comando é concluída e o *prompt* de comando é exibido, pressiona-se a tecla de *Reset* no computador alvo. Este experimento foi coordenado pelo operador humano, na função de controle. Com isso, não houve um desligamento normal do sistema e ficaram metadados a serem atualizados para a partição /mnt/teste. Assim, logo que a máquina fosse reiniciada seria necessário tornar o sistema de arquivos consistente.

Teremos que o tempo necessário para tornar o sistema de arquivos consistente (ou T_2) será o tempo necessário para a montagem, incorporando as transações que

encontram-se descritas no *log* da partição. Executando-se o comando **time mount /dev/hda3 /mnt/teste**, obteve-se um valor que será o T_2 desta partição.

Entretanto, deste tempo que foi medido é necessário descontar aquilo que seria necessário para montar esta partição caso não houvesse a necessidade de incorporar mudança alguma do *log* do *journaling*. Logo, teremos na Figura 7.3 que:

T_j = Tempo para montagem, incorporando as mudanças no *log* do *journaling*
T_m= Tempo para montagem, quando não há mudanças no *log* do *journaling*
$T_2 = T_j - T_m$

Figura 7.3: Definição de T_2 como sendo a diferença entre dois tempos

Após uma série de 30 testes, determinou-se que o tempo médio para montagem de uma partição para quando não há mudanças nos metadados é de 0,246 segundos, com um desvio padrão de 0,003282 segundos. É possível determinar o tempo médio de recuperação para quando o desligamento é feito através do botão de *Reset* (e, portanto, existem várias transações registradas em *log*). Quando isso acontece, o tempo necessário para montar o sistema de arquivos é de 1,692 segundos, com um desvio-padrão de 0,031728 segundos. Com isto temos que o tempo médio apenas para a tarefa de recuperação é a diferença entre essas medidas do tempo para a montagem (para quando há e para quando não há transações registradas no *log)*, com o valor de 1,446 segundos.

A cobertura de falhas será de 100%, haja visto que todas as falhas injetadas foram detectadas. Utilizou-se o comando **grep –i ´start recovery´ /var/log/dmesg** para determinar se houve recuperação durante a montagem do sistema de arquivos.

7.4 Análise dos resultados obtidos

Por meio da realização de uma injeção manual de falhas, pôde-se determinar que o tempo de recuperação para um sistema de arquivos XFS é realmente bastante menor do que o tempo de recuperação para um sistema de arquivos ext2.

Para isso, em um mesmo computador foi gerada uma mesma carga de trabalho de um único arquivo grande por meio do utilitário *bonnie++*. Observou-se que o tempo de recuperação para o XFS foi de 0,161033 segundos, enquanto que para o ext2 este tempo é de 2,553267 – cerca de 16 vezes menor.

Alterando-se a carga de trabalho para que fosse de 2048 arquivos de tamanho aleatório distribuídos ao longo de 5 diretórios observou-se que o tempo de recuperação para o XFS foi de 1,446 segundos enquanto que para o ext2 este tempo foi de 1min e 30.247 segundos: cerca de 62 vezes menor.

Com isto, fica comprovada a hipótese de que o XFS apresenta um tempo de recuperação bem menor do que o ext2. Experimentos manuais como este servem para obter medidas rápidas e não refinadas de disponibilidade, o que permite descartar sistemas alvos de baixa qualidade com pouco esforço.

7.5 Teste da ferramenta

Nesta seção são apresentados os testes inicias da ferramenta, a comprovação da injeção de falhas realizada. Com o uso de um programa exemplo, é feita a injeção de falhas e os resultados são comprovados junto a um arquivo de saída.

Para a realização dos experimentos será utilizado o fonte abaixo, na Figura 7.4, chamada prog_test.c [GON 2001]:

```
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main()
{
     int fd;
     int i,c;
     char * msg1="AAAAAAA\n";
     char * msg2="BBBBBBB\n";

     if ((fd=open("arq_test.txt", O_SYNC | O_RDWR |
O_CREAT, S_IRWXU | S_IRWXG | S_IROTH)) < 0) {
          printf("Erro ao abrir o arquivo");
          exit(1);
     }

     for (i=0; i<5000; i++) {
          c = write(fd, msg1, 8);
          c = write(fd, msg2, 8);
     }

     close(fd);
     return(0);
}
```

Figura 7.4: Listagem do programa para teste do FIJI

A metodologia de testes é bastante simples: tomando-se um programa simples, com o resultado de execução já conhecido, pode-se injetar falhas e observar o resultado desta execução. Como o roteiro de falhas é previamente definido, o resultado da execução – desde que o injetor funcione corretamente – também é conhecido.

Os defeitos provenientes dos erros injetados no experimento, observados no resultado da execução do programa exemplo, são condizentes com as regras de injeção

de falhas consideradas. Desta forma, o funcionamento da ferramenta FIJI foi comprovado.

Visando o teste da ferramenta FIJI, foi criada uma aplicação simples que armazena seqüências de caracteres em um arquivo. A seqüência e o resultado final de uma execução desta aplicação já são conhecidos. Alterações no tamanho do arquivo gerado ou em seu conteúdo significam que o injetor está agindo sobre a aplicação.

Utilizou-se gravação síncrona (parâmetro O_SYNC na função *open*) para garantir que os dados sejam gravados no disco assim que a função for executada. Em suma, assegura-se que haverão 10.000 operações de escrita.

Ao fim de uma execução correta do programa exemplo, existe um arquivo chamado arq_test.txt com 80.000 *bytes*. Ele contém seqüências alternadas de frases "AAAAAAA" e "BBBBBBB" – sempre seguidas por um avanço de linha (caractere "\n").

No teste para injeção de falhas, utilizou-se um roteiro bastante simples. Especificou-se que 3 segundos após o início do processo, todas as chamadas de sistema *write* deveriam ser descartadas.

7.6 Resumo

Este capítulo apresentou os resultados obtidos por meio de experimentos manuais para validação do XFS. Os mesmos foram realizados para diferentes cargas de trabalho: um único arquivo de tamanho grande, e vários arquivos de tamanho pequeno distribuídos ao longo de diversos diretórios. Concluiu-se que o XFS apresenta um tempo de reparo bastante menor do que o ext2.

Além disso, foi apresentado um teste de correção para a ferramenta FIJI. Para um programa do qual o resultado era conhecido, observou-se o modo como a ferramenta afetou os dados na saída durante a execução. E o mais importante: comprovou-se que ela implementa as falhas de acordo com o modelo definido.

8 CONCLUSÕES

Este capítulo apresenta as conclusões obtidas após a implementação do FIJI, faz algumas considerações sobre mudanças que poderiam ser feitas no modelo de falhas e aponta perspectivas para a realização de pesquisas relacionadas com a ferramenta desenvolvida.

8.1 Introdução

Os resultados obtidos foram considerados extremamente positivos. Pôde-se comprovar a validade da injeção de falhas baseada nos mecanismos de depuração oferecidos pelo Linux e suas possibilidades. Foi possível o desenvolvimento de uma ferramenta para injeção de falhas que torna possível inspecionar e alterar o conteúdo dos registradores, bem como das posições de memória por meio da chamada de sistema *ptrace*.

A abordagem apresentada caracterizou-se por trabalhar em um nível de abstração alto, mas nem por isso menos poderoso. A injeção de falhas com um controle maior da forma de manifestação do erro apresentou-se como uma alternativa importante na validação das técnicas de tolerância a falhas usadas nas aplicações. Desta forma, pode-se validar estes mecanismos em função dos erros injetados – tendo-se certeza de que os erros acontecerão, e do momento em que isto ocorrerá. Observar o comportamento de uma aplicação em um cenário conhecido pode fornecer informações precisas a respeito da cobertura e eficiência dos mecanismos de detecção e tratamento de falhas.

8.2 Dificuldades encontradas

A maior parte das dificuldades deve-se ao fato de que dispendeu-se um tempo demasiado procurando adequar o FIDe às necessidades deste experimento, e de que não encontra-se facilmente publicações sobre a abordagem baseada em *journaling*. Entretanto, uma vez que decidiu-se pelo desenvolvimento de uma nova ferramenta os problemas relacionados com adaptações no FIDe foram contornados.

Os problemas encontrados devem-se ao fato de que o FIDe foi desenvolvido com base nos recursos do *kernel* do Linux na série 2.2, enquanto o XFS exige um *kernel* da serie 2.4. Por este motivo, em um primeiro momento procurou-se fazer adequações nesta ferramenta mas após inúmeras tentativas ela não apresentava os resultados esperados no seu teste de avaliação [GON 2001]. Por este motivo, optou-se por utilizar

a mesma abordagem definida para esta ferramenta para desenvolver uma outra completamente nova – a qual foi designada FIJI.

Conforme é descrito na Seção 2, o FIDe é uma ferramenta que permite a criação de cenários de falhas genéricos para a injeção de falhas onde pode ser especificada não só alterações no conteúdo de registradores bem como nas posições de memórias indicadas por endereços. Em suma, ele seria uma ferramenta ideal para os testes devido à facilidade para representar qualquer modelo de falhas. Essa possibilidade encontra-se apontada como uma perspectiva de trabalho futuro.

Quanto ao acesso às publicações, a maior parte do material sobre sistemas de arquivos Unix trata sobre a abordagem *Soft Updates*. Ainda assim, alguns poucos artigos disponíveis no próprio site do sistema de arquivos XFS [XFS 2004] foram úteis para elucidar detalhes sobre este tipo de organização.

8.3 Melhorias na implementação atual

Existem ainda muitos detalhes a serem melhorados na implementação do FIJI:

- Mudar a resolução de tempo de segundos para milissegundos;
- Permitir a descrição, via *script*, dos cenários de falhas a serem considerados ao invés de mantê-los incorporados ao código-fonte da ferramenta. Isso deve poder ser realizado de um modo mais genérico, tal como acontece no FIDe; e
- Estender o modelo de falhas atual, para abranger outros tipos de falhas. Como é dito na seção 5.1.1, o estado de um sistema de arquivos baseado em *journaling* é resultado de uma associação entre as informações localizadas em três lugares diferentes: disco, memória e *log*. O modelo de falhas poderia ser estendido para abranger também falhas de corrupção de memórias, entre vários outros tipos de erros que são possíveis considerando-se estes três diferentes lugares.

8.4 Resultados alcançados e trabalhos futuros

Entende-se que o principal resultado alcançado com este trabalho tenha sido o desenvolvimento do injetor de falhas FIJI. Esta ferramenta comprova o sucesso da abordagem para injeção de falhas baseando-se nos recursos para depuração no Linux disponibilizados por meio da chamada de sistema *ptrace*.

Embora a realização de testes com o FIJI tenha sido limitada apenas aos aspectos de avaliação da ferramenta, é possível assegurar que podem ser realizados novos trabalhos. A ferramenta realmente simulou os efeitos da falta de energia elétrica fazendo com que os logs das operações realizadas fossem ignorados, e todos os acessos a disco subseqüentes fossem impedidos.

E quanto às perspectivas para realização de trabalhos futuros, uma das principais é o estudo da dependabilidade oferecida por outros sistemas de arquivos baseados em *journaling*, além do XFS. Por fim, o ideal seria fazer um trabalho comparativo relacionando os resultados obtidos com os diferentes sistemas de arquivos – ext3, JFS, ReiserFS e XFS.

Praticamente ao término deste trabalho, houve o lançamento da versão 9.0 da distribuição Suse Linux - a qual também está permitindo o uso do sistema de arquivos XFS. Sendo assim, também seria interessante utilizar como sistema alvo uma máquina

baseada na distribuição Suse Linux 9.0. Isso é algo que traria a vantagem de possibilitar a utilização da ferramenta para injeção de falhas *Faumachine* (vide seção 2.6.3). Por meio desta ferramenta, podem ser definidos cenários de falhas mais complexos – envolvendo corrupção de memória ou disco. Isto seria bastante útil para estender o modelo de falhas atual.

REFERÊNCIAS

[BAR 2000] BAR, M. **Linux Kernel Internals**. New York: McGraw-Hill, 2000. 351 p.

[BON 2003] BONNIE Benchmark. Disponível em: <http://www.textuality.com/bonnie>. Acesso em: abr. 2003.

[BRY 2002] BRYANT, R.; FORESTER, R.; HAWKES, J. Filesystem Performance and Scalability in Linux 2.4.17. In: USENIX ANNUAL TECHNICAL CONFERENCE, 2002. **Proceedings...** Disponível em: <http://oss.sgi.com/projects/xfs/publications.html>. Acesso em: out. 2003 .

[CAR 95] CARD, R.; TS'O, T.; TWEEDIE, S. Design and Implementation of the Second Extended Filesystem. In: DUTCH INTERNATIONAL SYMPOSIUM ON LINUX, 1995. **Proceedings...** Disponível em: <http://web.mit.edu/tytso/www/linux/ext2intro.html>. Acesso em out. 2003.

[CAR 95b] CARREIRA, J.; MADEIRA, H.; SILVA, J. G. Xception: Software-Fault Infection and Monitoring in Processor Functional Units. In: WORKING CONFERENCE ON DEPENDABLE COMPUTING FOR CRITICAL APPLICATIONS, 1995. **Proceedings…** Urbana-Champaign, Estados Unidos: [s.n.], 1995.

[CHR 91] CHRISTIAN, F. Understanding Fault-Tolerant Distributed Systems. **Communications of the ACM**, New York, v. 34, n. 2, p. 56-78, 1991.

[DAW96a] DAWSON, S.; JAHANIMAN, F.; MITTON, T. A Probing and Fault Injection Environment for Testing Protocol Implementations. In: IPDS, 1996. **Proceedings...** Urbana-Champaign, Estados Unidos: [s.n.], 1996.

[DAW96b] DAWSON, S.; JAHANIMAN, F.; MITTON, T.; TUNG, T. Testing of Fault-Tolerant and Real-Time Distributed Systems via Protocol Fault Injection. In: FAULT-TOLERANT COMPUTING SYMPOSIUM, 26., 1996. **Proceedings…** New York: IEEE Computer Society Press, 1996.

[DEP 2003] DEPENDABILITY Benchmarking Project. Disponível em: <http://www.laas.fr/Dbench>. Acesso em: out. 2003.

[GON 2001] GONÇALVES, L. C. R. **Injeção de Falhas por Depuração**. 2001. 131p. Dissertação (Mestrado em Ciência da Computação) – Instituto de

Informática, UFRGS, Porto Alegre.

[GRA 93] GRAY, J.; REUTER, A. **Transaction Processing**: Concepts and Techniques. San Mateo, CA: Morgan Kaufmann, 1993.

[HAG 2004] HAGEN, B. V. **Overview of Linux *Journaling* Filesystems**. Disponível em: <http://www.informit.com/articles/article.asp?p=26153&seqNum=1>. Acesso em: mar. 2004.

[HOX 2002] HÖXER, H. J.; BUCHACKER, K.; SIEH, V. UMLinux - A Tool for Testing a Linux System's Fault Tolerance. In: LINUXTAG, 2002, Karlsruhe, Germany. **Proceedings...** Disponível em: <http://www3.informatik.uni-erlangen.de/Publications/Articles>. Acesso em: out. 2003.

[HSU 97] HSUEH, M. C.; TSAI, T. K.; IYER, R. K. Fault-Injection Techniques and Tools. **Computer**, New York, v. 30, n. 4, p.75-82, Apr. 1997.

[JAC 2004] JACQUES-SILVA, G.; MORAES, R.L.; WEBER, T.S.; MARTINS, E. Validando sistemas distribuídos desenvolvidos em Java utilizando injeção de falhas de comunicação por *Software*. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, WTF, 2004. **Anais...** Gramado, RS: Sociedade Brasileira de Computação, 2004. v. 1, p. 53-64.

[JAL 94] JALOTE, P. **Fault-Tolerant in distributed systems**. New Jersey: Prentice Hall, 1994. 432 p.

[JFS 2003] JFS Sistema de arquivos para Linux. Disponível em: <http://oss.*software*.ibm.com/jfs>. Acesso em: out. 2003.

[KOO 99] KOOPMAN Jr, P. J.; DEVALE, J. Comparing the Robustness of POSIX Operating Systems. In: INTERNATIONAL SYMPOSIUM ON FAULT TOLERANT COMPUTING SYSTEM, 1999. **Proceedings...** [S.l.:s.n.], 1999. p. 30-37.

[LAD 98] JOHNSON, M. K. **Linux application development**. [S.l.]:Addison-Wesley, 1998. 538 p.

[LAP 92] LAPRIE, J. C. **Dependability**: Basic Concepts and Terminology**.** Vienna: Springer-Verlag, 1992.

[LEI 2000] LEITE, F. O. **ComFIRM – Injeção de Falhas de Comunicação Através da Alteração de Recursos do Sistema Operacional.** 2000. 114 p. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[LYU 96] LYU, M. R. **Software Reliability Engineering**. New York: McGraw-Hill, 1996.

[MAE 87] MAES, P. Concepts and Experiences in Computational Reflection. In:

OBJECT-ORIENTED PROGRAMMING, SYSTEMS, LANGUAGES & APPLICATIONS, 1987. **Proceedings...** Orlando, Estados Unidos: [s.n.], 1987.

[MOS 2000] MOSTEL, J. et al. Porting the SGI XFS File System to Linux. In: USENIX ANNUAL TECHNICAL CONFERENCE, 2000. **Proceedings...** Disponível em: <http://www.usenix.org/publications/library/proceedings/usenix2000/freenix/mostek.html>. Acesso em: out. 2003.

[PIE 2002] PIERNAS, J.; CORTES, T.; GARCIA, J. M. DualFS: a New *Journaling* File System without Meta-Data Duplication. In: ANNUAL ACM INTERNATIONAL CONFERENCE ON SUPERCOMPUTING 16., 2002. **Proceedings...** New York City: [s.n.], 2002.

[PRA 96] PRADHAN, D. **Fault-tolerant computer systems design**. [S.l.]: Prentice-Hall, 1996.

[REI 2003] REISER, H. **Sistema de arquivos ReiserFS para Linux**. Disponível em: <http://www.namesys.com>. Acesso em: out. 2003.

[ROS 98] ROSA, A. A.; MARTINS, E. Using reflexive programming to inject faults in object-oriented systems. In: IFIP INTERNATIONAL WORKSHOP ON DEPENDABLE COMPUTING AND ITS APPLICATIONS, 1998. **Proceedings...** Johannesburg, South Africa: [s.n.], 1998. p. 227-236.

[SAT 81] SATYANARAYANAN, M. A study of file sizes and functional lifetimes. In: ACM SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 8., 1981. **Proceedings...** [S.l.:s.n.], 1981. p. 96-108.

[SEL 92] SELTZER, M. **File System Performance and Transaction Support**.1992. 131p. Tese de doutorado. University of California, College of Engineering, Computer Science Division.

[SEL 2000] SELTZER, M. I. et al. *Journaling* versus soft-updates: Asynchronous meta-data protection in file systems. In: USENIX ANNUAL TECHNICAL CONFERENCE, 2000. **Proceedings...** San Diego, California, USA, 2000. Disponível em: < http:// www.lcs.ece.cmu.edu/~soule/papers/seltzer.pdf>. Acesso em: ago. 2003.

[SHE 98] SHERMAN, L. **Choosing the right availability solution – High-Availability Clusters & Fault Tolerant Systems**. 1998. 10p. Disponível em: < http://www.highavailabilitycenter.com/ha_reports/stratus2.pdf>. Acesso em: dez. 2003.

[SIE 93] SIEH, V. **Fault-Injector Using UNIX ptrace Interface.** [S.l.]: IMMD3, Universität Erlangen-Nürnberg, 1993. (Internal Report 11/93).

[SIE 2002] SIEH, V.; BUCHACKER, K. UMLinux - A Versatile SWIFI Tool. In: EUROPEAN DEPENDABLE COMPUTING CONFERENCE, 4.,

2002, Toulouse, France. **Proceedings...** Berlin: Springer-Verlag, 2002. p. 159-171. (Lecture Notes in Computer Science, v. 2485). Disponível em: <http://www3.informatik.uni-erlangen.de/Publications/Articles/sieh_edcc2002.pdf>. Acesso em: ago. 2003.

[SWE 96] SWEENY, A.; DUCETTE, D. et al. Scalability in the XFS File System. In: USENIX TECHNICAL ANNUAL CONFERENCE, 1996. San Diego, California, USA. Disponível em: <http://oss.sgi.com/projects/xfs/papers/xfs_usenix>. Acesso em: out. 2003.

[TAN 97] TANENBAUM, A. S.; WOODHULL, A. S. **Operating systems**: design and implementation. New Jersey: Prentice Hall, 1997. 759 p.

[TRA 2003] TRAUTMAN, P.; MOSTEK, J. **Scalability and Performance in Modern File Systems**. Disponível em: <http://oss.sgi.com/projects/xfs/papers/xfs_white/xfs_white_paper.html>. Acesso em: ago. 2003.

[TRI 82] TRIVEDI, K. S. **Probability and Statistics with Reliability, Queuing, and Computer Science Applications**. Englewood Cliffs, NJ: Prentice Hall, 1982.

[TWE 98] TWEEDIE, S. C. *Journaling* the Linux ext2fs Filesystem. In: LINUXEXPO, 1998. **Proceedings...** Disponível em: <http://www.skylab.org/~sabre/os/S3FileSystems/journal-design.pdf>. Acesso em: jul. 2003.

[TWE 2003] TWEEDIE, S. C. Ext3, *Journaling* Filesystem for Linux. In: OTTAWA LINUX SYMPOSIUM, 2000. Disponível em: <http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html>. Acesso em: out. 2003

[WEY 2001] WEYGANT, P.; HEWLETT-PACKARD COMPANY. **Clusters for High Availability**. New Jersey: Prentice Hall, 2001.

[XCE 2003] XCEPTION - injetor de falhas. Disponível em: <http://www.xception.org/home/index.php>. Acesso em: dez. 2003.

[XFS 2003] XFS Sistema de arquivos para Linux. Disponível em: <http://xfs.sourceforge.net>. Acesso em jun. 2003.

[ZEM 2003] TEST of six filesystems for linux on kernel 2.4.5 – ext2, ext3, jfs, ReiserFS, vfat and xfs. Disponível em: <http://aurora.zemris.fer.hr/filesystems>. Acesso em: jan. 2003.