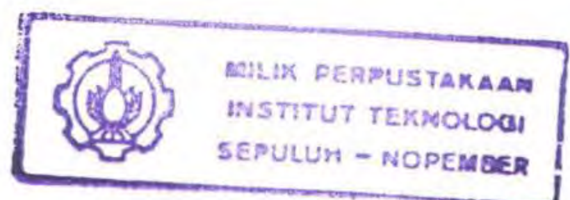
22595/H/05



PERANCANGAN DAN PEMBUATAN PERANGKAT LUNAK HISTORICAL FILE SYSTEM DI LINUX

TUGAS AKHIR

RSIF
005.1
Wal
p-1
2005

PERPUSTAKAAN
ITS

Tgl. Terima	16-2-2005
Terima Dari	H
No. Agenda Prp.	221447

Disusun Oleh :

EDDY YUNIAR WALUYO
5197 100 052

JURUSAN TEKNIK INFORMATIKA
FAKULTAS TEKNOLOGI INFORMASI
INSTITUT TEKNOLOGI SEPULUH NOPEMBER
SURABAYA
2005

PERANCANGAN DAN PEMBUATAN PERANGKAT LUNAK HISTORICAL FILE SYSTEM DI LINUX

TUGAS AKHIR

Diajukan Untuk Memenuhi Sebagian Persyaratan

Memperoleh Gelar Sarjana Komputer Pada

Jurusan Teknik Informatika

Fakultas Teknologi Informasi

Institut Teknologi Sepuluh Nopember

Surabaya

Mengetahui / Menyetujui

Dosen Pembimbing I	Dosen Pembimbing II
Yudhi Purwananto, S.Kom, M.Kom	Wahyu Suadi, S.Kom, M.Kom
NIP. 132 172 210	NIP. 132 303 065

Surabaya
Januari, 2005

ABSTRAK

Historical file merupakan teknik untuk mencatat sejarah perubahan yang pernah terjadi khususnya pada regular file. Aplikasi dari versioning mencakup backup dan disaster recovery, juga digunakan untuk memonitor aktifitas penyusup (intruder). File system yang didukung dan yang berjalan di Linux tidak menyediakan model file versioning yang berjalan secara transparan dan mudah untuk digunakan.

Dalam Tugas Akhir ini dibuat historical file system yang mendukung file versioning. Historical file system bekerja di atas file system yang sudah didukung oleh Linux. Salah satunya adalah Ext2 dan Reiserfs. File system ini menyediakan konfigurasi historical file bagi user berupa policy penyimpanan. Beberapa kumpulan utility pada level user disediakan untuk men-setting atribut historical ini dan juga memanipulasi file-file versi yang terbentuk. Aplikasi yang dibuat di Tugas Akhir ini diimplementasikan dengan menggunakan pendekatan yang hampir sama dengan teknik stackable file system, yaitu teknik injeksi fungsi VFS milik native file system. Desain ini menyebabkan file system baru dapat bekerja diatas file system yang sudah ada.

Dari hasil uji coba yang telah dilakukan, menunjukkan historical file system yang didesain menggunakan teknik injeksi VFS, sebagai alternatif teknik stackable file system, dapat berjalan di atas native file system yang didukung Linux. Begitu juga dengan utility pada level user untuk memanipulasi file versi, dapat berjalan dengan baik.

KATA PENGANTAR

Puji syukur dipanjatkan kepada Tuhan Yang Maha Esa yang telah memberi restu dan anugrah kepada penulis untuk dapat menyelesaikan salah satu kewajiban dalam pendidikan perguruan tinggi yaitu Tugas Akhir. Adapun Tugas Akhir yang penulis beri judul:

" PERANCANGAN DAN PEMBUATAN PERANGKAT LUNAK HISTORICAL FILE SYSTEM DI LINUX "

merupakan salah satu syarat untuk mendapatkan gelar sarjana strata satu (S1) Teknik Informatika Fakultas Teknologi Informasi, Institut Teknologi Sepuluh Nopember (ITS) Surabaya. Tugas Akhir yang berbobot 4 sks ini bertujuan mengasah kemampuan mahasiswa untuk memberikan suatu solusi terhadap permasalahan sesuai ilmu yang diperoleh dalam perkuliahan sekaligus mengimplementasikan solusi tersebut. Penulis berharap Tugas Akhir ini dapat memberikan manfaat bagi pembaca, terutama yang tertarik dengan topik *system* operasi khususnya *file system*. Dan semoga dapat memberikan tambahan ilmu bagi pembaca.

Ada banyak rintangan dan kesulitan dalam menyelesaikan karya ini sehingga penulis menyadari bahwa Tugas Akhir ini tidak akan selesai tanpa bantuan orang lain. Oleh karena itu penulis ingin menyampaikan terima kasih dan penghargaan yang sebesar-besarnya kepada pihak-pihak yang membantu secara langsung maupun tidak langsung baik pada saat penulis kuliah maupun pada saat mengerjakan Tugas Akhir ini, antara lain:

1. Bapak Yudhi Purwananto, S.Kom M.Kom, selaku dosen pembimbing I, yang telah memberikan bimbingan, motivasi dan arahan kepada penulis pada pengerjaan Tugas Akhir.
2. Bapak Wahyu Suadi, S.Kom M.Kom, selaku dosen pembimbing II, yang telah memberikan bimbingan, motivasi dan solusi pada saat pengerjaan Tugas Akhir.
3. Bapak Ir. Aris Tjahyanto, M.Kom, selaku dosen wali, yang telah memberikan bimbingan, motivasi dan solusi selama masa perkuliahan.
4. Seluruh dosen Teknik Informatika yang telah menyampaikan ilmu dan pengalamannya pada saat proses perkuliahan.
5. Seluruh karyawan Teknik Informatika, terima kasih atas kerjasama yang baik semasa penulis mengikuti perkuliahan.
6. Bapak dan ibu yang ada di malang, terima kasih atas kesabarannya menunggu anaknya lulus dari TC ITS; maaf jika tidak bisa melakukan sesuatu yang lebih baik.
7. Andre, Benny, Christin, dan Doni yang dapat menjadi teladan bagi adiknya untuk menentukan jalan hidup serta karir.
8. Teman-teman C0D semua, serta yang pernah membantu penulis untuk keluar dari masalah drop-out, terima kasih sudah bersusah payah.
9. Teman-teman AJK, angkatan 1995 sampai dengan angkatan 2000 yang terlalu banyak untuk disebutkan; terima kasih atas kebersamaannya selama ini (makan, tidur, cangkruk bareng) serta kontribusi yang sangat besar baik jasmani dan rohani bagi penulis saat hidup di TC.

10. David dan Maya, yang telah membantu penulis untuk merasakan sejenak tinggal di luar negeri; maaf jika pernah menyinggung kalian secara tidak sengaja.
11. Teman-teman diskusi mengenai hidup dan menjadi manusia, terima kasih telah meluangkan waktunya dengan penulis.
12. Para ahli dibidangnya, terima kasih karena penulis dapat belajar tentang kehidupan dari kacamata master-nya.
13. Para komunitas open source, yang banyak menimbulkan inspirasi ke penulis tentang pentingnya keterbukaan, kerjasama, altruistik, dan seni dalam memprogram.

Akhirnya kepada pihak-pihak yang telah membantu, penulis menyampaikan terima kasih dan semoga Tuhan memberikan anugerah dan kedamaian.

Surabaya, Januari 2005

Eddy Yuniar Waluyo

DAFTAR ISI

ABSTRAK i
KATA PENGANTAR ii
DAFTAR ISI v
DAFTAR TABEL DAN GAMBAR vii
BAB I 1
1.1 LATAR BELAKANG 1
1.2 PERMASALAHAN 2
1.3 TUJUAN 2
1.4 PEMBATASAN MASALAH 3
1.5 METODOLOGI 3
1.6 SISTEMATIKA PEMBAHASAN 4
BAB II 6
2.1 ARSITEKTUR KERNEL LINUX 6
2.1.1 Process Scheduler 7
2.1.2 Memory Manager 9
2.1.3 Virtual File system 11
2.1.4 Interprocess Communication 13
2.1.5 Network interface 15
2.2 PENGENALAN FILE SYSTEM UNIX 17
2.2.1 File 17
2.2.2 Hard link dan Soft link 18
2.2.3 Tipe-tipe File 19
2.2.4 File Descriptor dan Inode 19
2.2.5 Ijin akses dan mode file 20
2.2.6 System call untuk penanganan file 21
2.3 USER MODE LINUX 21
2.4 PEMROGRAMAN C 23
2.4.1 Struktur Data 23
2.4.2 Proses pembuatan 24
BAB III 26
3.1 DESAIN HISTORICAL FILE 26
3.1.1 Teknik penyimpanan historical file 27
3.1.2 Mekanisme historical file 28
3.1.3 Policy di historical file 30
3.2 ARSITEKTUR FILE SYSTEM 33
3.2.1 Pendekatan desain arsitektur 33
3.2.2 Stackable file system 34
3.2.3 Teknik injeksi/hooking fungsi VFS (XPackfs) 36
3.2.4 Pseudo file system di XPackfs 43
3.2.5 Teknik penyimpanan data atribut historical di XPackfs 45
3.3 USERSPACE UTILITY 47
BAB IV 50

4.1 LINGKUNGAN IMPLEMENTASI 50
4.2 IMPLEMENTASI DATA 51
4.2.1 Struktur data kernel 51
4.2.1.1 Struktur data XPackfs 51
4.2.1.2 Struktur data Pseudofs 53
4.2.1.3 Struktur data database inode 54
4.2.1.4 Struktur data database atribut historical 55
4.2.2 Struktur data userspace utility 57
4.3 IMPLEMENTASI PROSES 58
4.3.1 Inisialisasi XPack di Hostfs 58
4.3.2 Fungsi historical file 64
4.3.2.1 Akses database atribut 64
4.3.2.2 Fungsi pembentukan historical file 66
4.3.2.3 Fungsi pembuatan versi 67
4.3.3 Fungsi manipulasi file versi 69
BAB V 70
5.1 LINGKUNGAN UJI COBA 70
5.2 PELAKSANAAN UJI COBA 70
5.2.1 Uji coba I 71
5.2.2 Uji coba II 74
5.2.3 Uji coba III 75
5.2.4 Uji coba IV 77
5.3 EVALUASI UJI COBA 78
BAB VI 80
6.1 KESIMPULAN 80
6.2 SARAN 80
DAFTAR PUSTAKA 82

DAFTAR TABEL DAN GAMBAR

Gambar 2.1 Desain konseptual kernel Linux ... 6
Gambar 2.2 Arsitektur konkrit kernel Linux ... 7
Gambar 2.3 Struktur process scheduler ... 8
Gambar 2.4 Struktur memory manager ... 11
Gambar 2.5 Struktur Virtual File System ... 13
Gambar 2.6 Struktur IPC ... 15
Gambar 2.7 Struktur Network Interface ... 17
Gambar 2.8 Contoh directory tree ... 18
Gambar 2.9 Struktur layout Linux ... 21
Gambar 2.10 Struktur layout UML di Linux ... 22
Gambar 2.11 Struktur program C ... 24
Gambar 2.12 Alur proses pembuatan program ... 25
Gambar 3.1 Letak file dan versinya pada hirarki ... 27
Gambar 3.2 Alur proses perubahan file ... 29
Gambar 3.3 Alur proses perubahan nama file ... 30
Gambar 3.4 Interval waktu untuk memicu terjadinya versi ... 32
Gambar 3.5 Informasi dan aliran proses pada stackable file system ... 35
Tabel 3.1 System call yang ditangani VFS ... 37
Gambar 3.6 Struktur data utama objek-objek VFS di kernel Linux 2.6.x ... 38
Gambar 3.7 Alur proses (normal) dari operasi di file system ... 39
Gambar 3.8 Alur proses (setelah di hook) dari operasi di file system ... 40
Gambar 3.9 Keberadaan virtual file di hirarki XPackfs ... 44
Gambar 3.10 Alur proses pembacaan virtual file yang mengacu ke file Hostfs... 45
Gambar 3.11 Hubungan antara database inode dan database atribut ... 47
Gambar 3.12 Alur proses system call ioctl ... 48
Gambar 4.1 Struktur xpack_info ... 51
Gambar 4.2 Struktur xinode_info ... 52
Gambar 4.3 Struktur pseudo_info ... 53
Gambar 4.4 Struktur pinode_info ... 53
Gambar 4.5 Struktur xp_database_d ... 54
Gambar 4.5 Struktur xp_db_index ... 54
Gambar 4.6 Struktur xp_db_info_d ... 55
Gambar 4.7 Struktur xp_historical_d ... 55
Gambar 4.8 Struktur xp_historical ... 56
Gambar 4.9 Struktur xp_attribute_u ... 57
Gambar 4.10 Struktur xp_historical_u ... 58
Gambar 4.11 Potongan kode inisialisasi awal XPackfs ... 59
Gambar 4.12 Struktur operasi superblock ... 60
Gambar 4.13 Struktur operasi dentry ... 60
Gambar 4.14 Struktur operasi inode ... 61
Gambar 4.15 Potongan kode inode lookup ... 62
Gambar 4.16 Potongan kode directory readdir ... 62

Gambar 4.17 Struktur operasi file 63
Gambar 4.17 Penambahan satu record pertama 65
Gambar 4.18 Penambahan record kedua 65
Gambar 4.19 Penghapusan record kedua 65
Gambar 4.20 Penghapusan record pertama 66
Gambar 4.21 Potongan kode ioctl 67
Gambar 4.22 Potongan kode write trigger 68
Gambar 4.23 Potongan kode time trigger 68
Gambar 4.24 Potongan kode ioctl untuk manipulasi file versi 69
Gambar 5.1 XPackfs berjalan diatas Reiserfs 71
Gambar 5.2 File modules di set atribut historical 72
Gambar 5.3 File modules setelah 3 kali mengalami perubahan 73
Gambar 5.4 Memanggil file-file versi milik modules 73
Gambar 5.5 Mengembalikan file-file versi milik modules 73
Gambar 5.6 Setting atribut historical di file smb.conf 74
Gambar 5.7 Setting script di bootmisc.sh 75
Gambar 5.8 Isi file modules pada awalnya 75
Gambar 5.9 Isi file modules setelah mengalami perubahan 76
Gambar 5.10 Perbedaan current file dengan file versi 1 77
Gambar 5.11 Perbedaan current file dengan file versi 2 77
Gambar 5.12 Perbedaan current file dengan file versi 3 77
Tabel 5.1 Uji coba aplikasi historical file 78

BAB I

PENDAHULUAN

1.1 LATAR BELAKANG

Salah satu kesuksesan dari sistem operasi Linux adalah kemampuan untuk menjalankan beberapa *file system* yang sering digunakan oleh *system* operasi lainnya. Linux dapat melakukan *mount* pada *disk* atau partisi pada format *file system* yang digunakan oleh Windows, Unix, ataupun *operating system* yang kurang popular seperti Amiga. Linux mendukung beberapa *file system* sama halnya dengan varian Unix lainnya, melalui konsep *Virtual File System*.

Ide dibalik *Virtual File System* adalah meletakkan berbagai informasi di dalam *kernel* yang merepresentasikan bermacam-macam tipe *file system*; terdapat field atau fungsi untuk mendukung setiap operasi yang diberikan oleh *file system* yang di-*support* oleh Linux. Untuk setiap pembacaan, penulisan, atau pemanggilan fungsi lainnya, *kernel* melakukan substitusi fungsi untuk mendukung *file system* Linux *native*, *file system* NT, atau *file system* lain.

Sudah banyak *file system* yang dikembangkan di Linux memiliki fasilitas-fasilitas baik untuk *performance*, *network file system*, *embedded file system*, dll. Untuk kebutuhan *security* masing - masing *file system* menyediakan atribut *read*, *write*, *access* (atribut standar). Pada versi Linux yang baru sudah menyediakan fasilitas *Access Control Lists* (ACLs) ataupun *Extended Attributes*.

Dari *file system* yang didukung oleh Linux, belum ada *file system* yang menyediakan fungsi untuk mencatat aktivitas perubahan terhadap *file*, sehingga

dalam kasus keamanan, apabila *file-file* yang penting untuk keamanan baik itu konfigurasi *file* (yang terdapat di /etc) ataupun *log file* (/var/log) mengalami perubahan oleh serangan *hacker* yang menyusup sebagai *user* yang tertinggi (*root*), dapat merubah konfigurasi ataupun menghapus *log* aktivitas selama terjadinya penyusupan guna menghilangkan jejak atau melemahkan *system security*.

1.2 PERMASALAHAN

Permasalahan yang dihadapi pada pengerjaan Tugas Akhir ini adalah :

- Bagaimana melakukan pencatatan perubahan *file* baik itu penambahan atau pengurangan data untuk keperluan *security audit* ataupun *versioning*.
- Bagaimana mengimplementasikan *historical file system* di atas *native file system* yang telah ada (seperti Ext2, Reiserfs).

1.3 TUJUAN

Tujuan dari Tugas Akhir ini adalah membuat suatu aplikasi *historical file system* pada *kernel* Linux untuk keperluan *security audit* ataupun *versioning* pada *file-file* teks (konfigurasi ataupun *log file*). Segala aktivitas pada *file* dapat tercatat baik itu perubahan, penghapusan; sehingga isi data *file* awal dan perubahan yang terjadi pada periode tertentu dapat terpantau.

1.4 PEMBATASAN MASALAH

Dalam penyusunan Tugas Akhir ini, dititikberatkan pada permasalahan dengan batasan sebagai berikut :

- *Historical file system* hanya berjalan diatas *native file system* Ext2, Reiserfs. Untuk jenis *file system* lain, diperlukan adanya penelitian lebih lanjut.
- Target *file* untuk pencatatan *historical* dititikberatkan untuk tipe data teks, dikarenakan umumnya ukurannya kecil, sehingga tidak terlalu memakan kapasitas disk untuk menampung *file-file* versinya.

1.5 METODOLOGI

Metode penelitian yang dilakukan dengan menggunakan langkah-langkah berikut :

1. Studi Literatur.

   Mencari, mempelajari, dan merangkum berbagai macam literatur yang berkaitan dengan proses pengembangan perangkat lunak. Diantaranya tentang Teori dan Arsitektur *Virtual File System* (VFS) pada Linux *Kernel*.

2. Perancangan Perangkat Lunak.

   Pada tahap ini dilakukan perancangan struktur data dari *historical file system* dan mengacu pada arsitektur dari *Virtual File System* (VFS).

3. Pembuatan Perangkat Lunak.

Pada tahap ini dilakukan pembuatan perangkat lunak dengan perencanaan pada tahap sebelumnya.

4. Implementasi dan Uji Coba Perangkat Lunak.

   Pada tahap ini akan dilakukan implementasi perangkat lunak yang sudah dibuat, disertai dengan melakukan *testing* pada *historical file system* yang berjalan di Linux.

5. Penulisan Tugas Akhir.

   Pada tahap ini akan dilakukan penyusunan laporan yang menjelaskan dasar teori dan metode yang digunakan dalam tugas akhir ini serta hasil dari implementasi perangkat lunak yang telah dibuat.

1.6 SISTEMATIKA PEMBAHASAN

Buku tugas akhir ini terdiri dari beberapa bab yang tersusun secara sistematis sebagai berikut :

BAB I Pendahuluan

Bab ini menjelaskan beberapa hal pokok dari tugas akhir ini, antara lain : latar belakang yang mendasari pembuatan tugas akhir, perumusan masalah, tujuan pembuatan tugas akhir, batasan masalah, metodologi yang berkaitan dengan pengerjaan tugas akhir ini dan sistematika penulisan tugas akhir ini.

BAB II Dasar Teori

Pada bab ini dibahas secara singkat teori-teori yang digunakan sebagai referensi dalam pengerjaan tugas akhir ini, meliputi: Arsitektur *kernel* Linux, pengenalan *file system* pada Unix, *User Mode* Linux, dan C *programming*.

BAB III Perancangan Perangkat Lunak

Perangkat lunak *Historical File system* pada Linux dirancang dan dibangun pada bab ini. Dibahas mulai dari desain *historical file*, arsitektur *file system*, dan perancangan *userspace utility* bagi *user*.

BAB IV Implementasi Perangkat Lunak

Perangkat lunak diimplementasikan dengan menggunakan bahasa umum yang terdapat pada *kernel* Linux yaitu C. Dijelaskan mengenai struktur data dan proses yang berjalan pada *historical file system*.

BAB V Uji coba dan Evaluasi

Bab ini digunakan untuk membahas pengujian dan evaluasi. Hasil dari pengembangan aplikasi akan diuji dan dievaluasi dengan berbagai macam kondisi sehingga tercapai hasil yang sesuai dengan yang diinginkan.

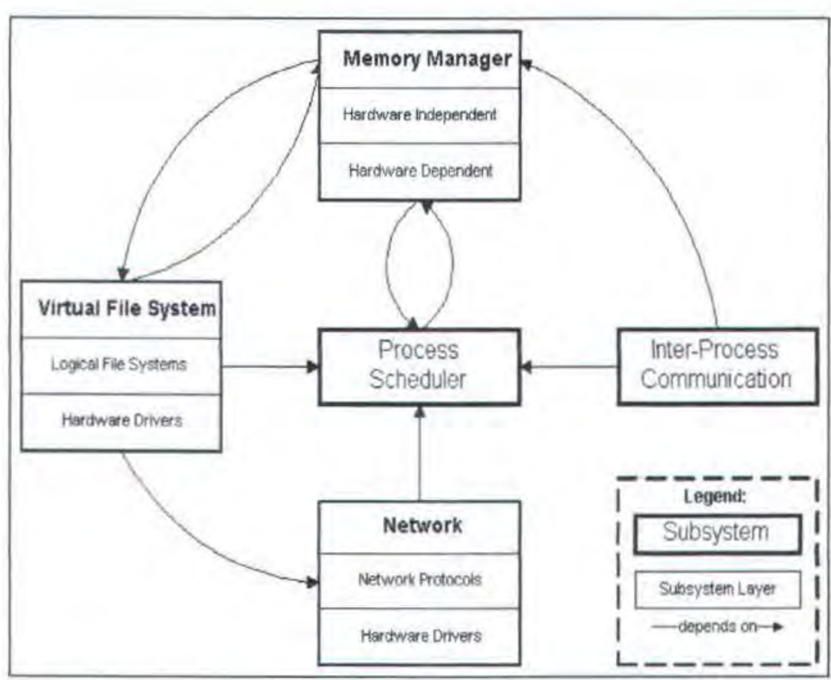
BAB VI Kesimpulan dan Saran

Bab ini berisi kesimpulan–kesimpulan yang dapat diambil dari proses pengembangan aplikasi tugas akhir ini. Juga berisi saran-saran untuk kepentingan pengembangan selanjutnya.

BAB II

DASAR TEORI

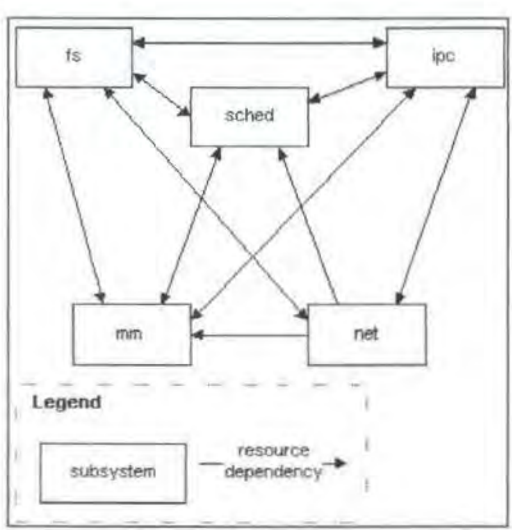
2.1 ARSITEKTUR KERNEL LINUX



Gambar 2.1 Desain konseptual kernel Linux

Linux terdiri dari 5 *subsystem* penting yakni *process scheduler* (sched), *memory manager* (mm), *virtual file system* (vfs), *network interface* (net), dan *interprocess communication* (ipc). Gambar 2.1 menunjukkan desain secara konseptual dari masing-masing *subsystem* dan keterkaitannya [7]. Secara konseptual dalam arti bahwa hanya berkonsentrasi pada desain *high level*, tidak secara sempurna merefleksikan implementasinya, tetapi dapat menyediakan model yang berguna bagi *developer* untuk memahami *kernel*.

Meskipun secara konseptual mempunyai sedikit ketergantungan, pada arsitektur konkritnya setiap *subsystem* sangatlah saling berkaitan. Gambar 2.2 menunjukkan arsitektur konkrit dari *kernel* Linux [8].



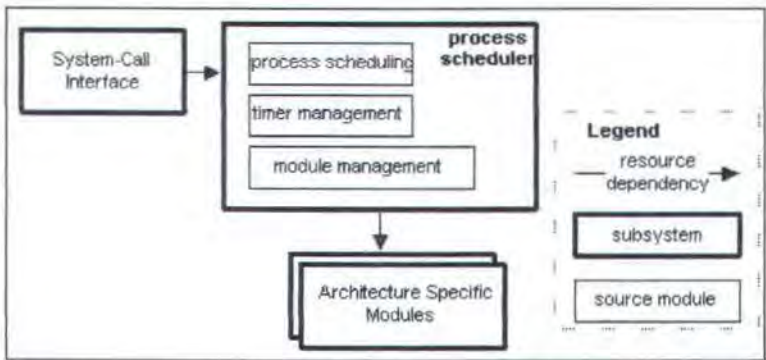
Gambar 2.2 Arsitektur konkrit kernel Linux

2.1.1 Process Scheduler

Penjadwalan proses adalah inti/pusat dari *system* operasi Linux. *Process scheduler* mempunyai tanggung jawab untuk:

- Mengijinkan proses untuk membuat duplikat baru dari dirinya
- Menentukan proses mana yang dapat mengakses CPU dan efek dari perpindahan antar proses
- Menerima *interrupt* dan meneruskannya ke *subsystem* lain dari *kernel*
- Mengirim *signal* ke *user* proses
- Mengatur waktu pada *hardware*
- Membersihkan *resource* pada proses ketika proses sudah berakhir

Struktur dari *process scheduler* dapat dilihat di gambar 2.3 :



Gambar 2.3 Struktur process scheduler

Manajemen proses diatur dengan memantau struktur data *task_struct* dari setiap proses. Terdapat *field* yang menunjukkan status dari proses, yang terdiri atas: *running,* kembali dari *system call,* memproses rutin *interrupt,* memproses *system call, ready, waiting.*

Dan juga terdapat *field* yang menunjukkan prioritas dari proses, *field* yang menyimpan jumlah *tick* dari waktu (interval 10 milidetik) dimana proses masih dapat melanjutkan eksekusinya sebelum dilakukan penjadwalan. Selain itu juga terdapat *field* yang menyimpan nomor error dari terakhir kali melakukan *system call.*

Untuk memantau semua proses, maka *task_struct* dari semua proses disimpan di *linked list.* Karena setiap proses mempunyai relasi dengan proses lain maka terdapat *field* yang berisi: *original parent, parent, child* termuda, *child* termuda, saudara lebih muda, dan saudara tertua.

Terdapat juga struktur *nested mm_struct* yang berisi informasi manajemen *memory*, (seperti alamat awal dan akhir dari *segment* kode). Informasi Proses ID juga disimpan dalam *task_struct*. Proses dan grup ID disimpan. *Array* dari grup ID juga disediakan supaya proses dapat mempunyai lebih dari satu grup.

Data *file* proses terletak di substruktur *fs_struct*, yang menyimpan *pointer* ke *inode* yang menunjuk ke *root directory* dan *working directory*. Semua *file* yang dibuka oleh proses akan disimpan di substruktur *files_struct* dalam *task_struct*.

Terdapat juga *field* yang menyimpan informasi waktu; misalnya, jumlah waktu dari proses yang dihabiskan di *User Mode*.

2.1.2 Memory Manager

Memory manager mempunyai kapabiltas berikut terhadap kliennya:

- *Address space* yang besar – program *user* dapat mereferensikan memori lebih besar dari kapasitas *memory* fisikal.
- Proteksi – *memory* untuk setiap proses adalah *private* dan tidak dapat dibaca atau dimodifikasi oleh proses lainnya; juga, *memory manager* mencegah proses untuk menulis/*overwrite* kode dan data yang *read-only*.
- *Memory mapping* – klien dapat *mapping file* ke area dari *memory virtual* dan mengakses *file* sebagai *memory*.
- Penanganan akses ke *memory* fisikal secara adil – *memory manager* memastikan bahwa semua proses mempunyai akses yang sama ke *resource memory* fisikal
- *Shared memory* – *memory manager* mengijinkan proses untuk men-*sharing* sebagian dari *memory*-nya. Misalnya, kode *executable* biasanya di-*share* antar proses

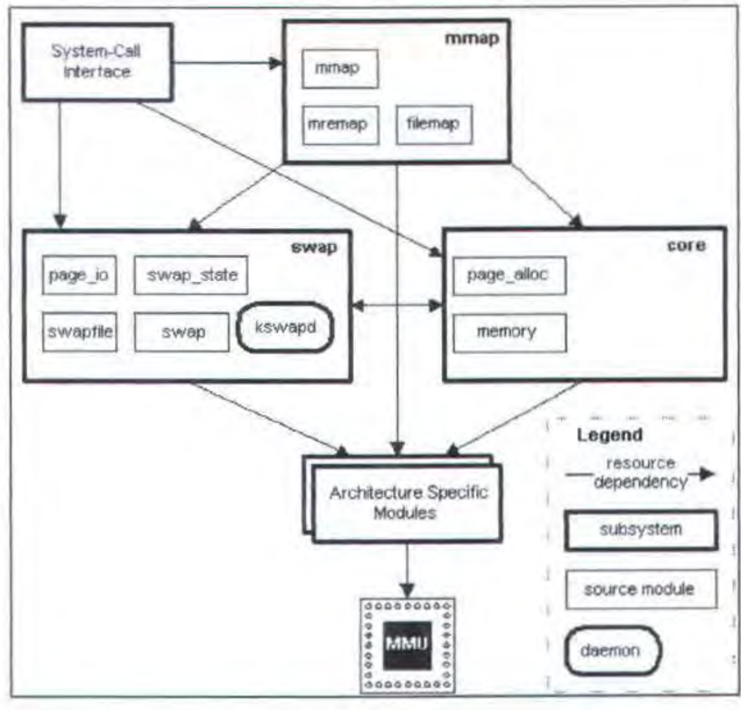
Struktur data yang relevan dengan *memory manager* adalah:

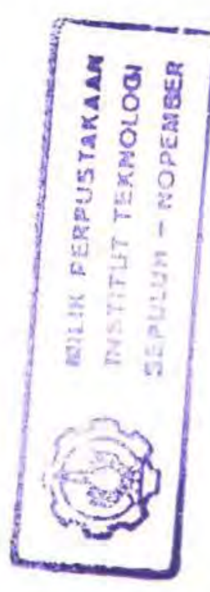
- *vm_area*. *Memory manager* menyimpan struktur data setiap proses tentang area dari *memory virtual* mana yang di-*mapping* ke *page* fisikal.

Struktur data ini juga menyimpan kumpulan *pointer* ke fungsi yang boleh digunakan pada area *memory virtual* milik proses. Misalnya, area kode *executable* dari proses tidak perlu di-*swap* ke *file system paging* karena dapat menggunakan *file executable* yang ada di *disk*. Ketika area dari *memory virtual* proses di *map*, *vm_area_struct* akan diset untuk setiap area yang berdekatan. Karena faktor kecepatan sangat penting pada saat mencari *page fault* pada *vm_area_struct*, maka struktur disimpan dalam bentuk *red-black trees*.

- *mem_map*. *Memory manager* menyimpan struktur data dari setiap *page* dari *memory* fisikal pada *system*. Struktur data ini berisi status/*flag* yang mengindikasikan status dari *page* (misalnya, apakah sedang digunakan). Setiap *page* disimpan dalam bentuk *vector* (*mem_map*), yang diinisialisasi oleh *kernel* pada saat pertama kali *booting*.
- *free_area*. *Vector free_area* digunakan untuk menyimpan informasi dari *page* pada fisikal *memory* yang belum dialokasikan; *page* di hapus dari *free_area* ketika dialokasikan, dan dikembalikan pada saat dibebaskan. *Buddy system* menggunakannya pada saat *page* dialokasikan.

Struktur dari *memory manager* dapat dilihat di gambar 2.4:




MILIK PERPUSTAKAAN INSTITUT TEKNOLOGI SEPULUH - NOPEMBER

Gambar 2.4 Struktur memory manager

2.1.3 Virtual File system

Tujuan dari *virtual fileystem* yang ada di Linux adalah untuk mendukung:

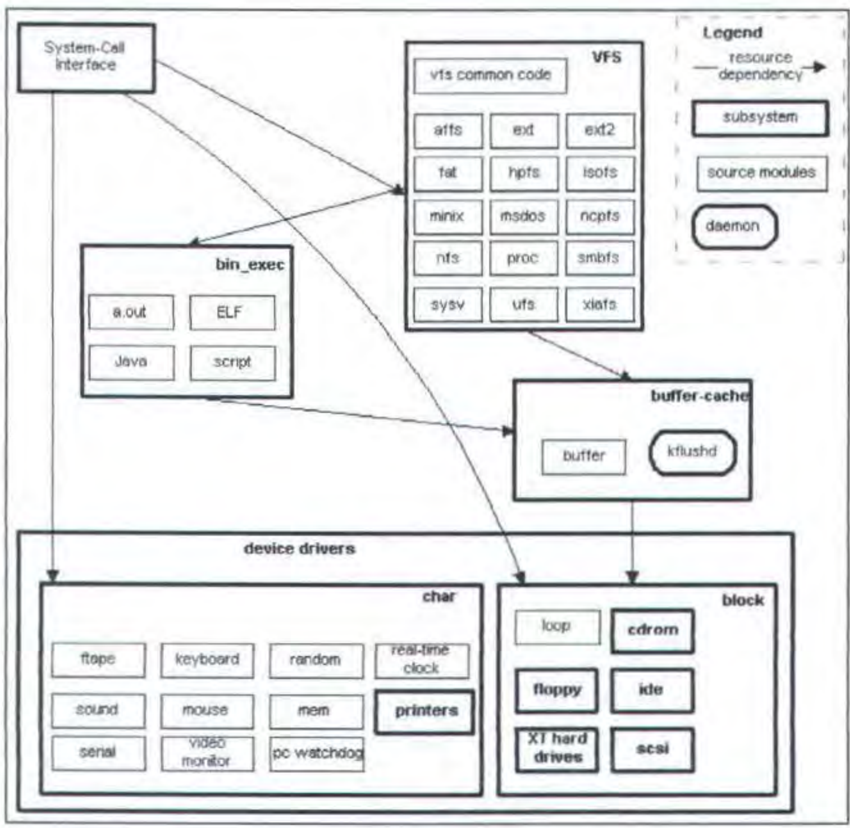
- Banyak *hardware device*; menyediakan akses ke *hardware device* yang berbeda.
- Banyak *logical file system*; mendukung banyak *logical file system* yang berbeda.
- Banyak format *executable*; mendukung beberapa *file* format (seperti a.out, ELF, java).
- *Interface* general ke semua bentuk *logical file system* dan *hardware device*.
- Performansi dalam mengakses *file*.
- Keamanan dari data sehingga tidak hilang ataupun *corrupt*.

- *Security*; membatasi akses *file* pada *user*; membatasi ukuran *file* dengan menggunakan kuota.

Berikut ini merupakan struktur data yang relevan dengan *file*:

- *super_block*; setiap *logical file system* mempunyai *superblock* yang berisi informasi tentang semua *file system* yang di-*mount*, baik itu tentang blok mana yang digunakan, berapa ukuran dari setiap blok, dan sebagainya. *Superblock* hampir sama dengan *inode* dalam hal menyediakan *virtual interface* untuk *logical file system*.
- *inode*; menyediakan semua informasi tentang *file* yang ada di *disk*. Satu *inode* mungkin digunakan untuk beberapa proses yang membukanya. *Accounting*, *buffering*, dan *memory mapping*, semua informasi disimpan dalam *inode*.
- *file*; struktur *file* mewakili sebuah *file* yang dibuka oleh proses. Semua *file* yang dibuka disimpan dalam bentuk *double linked list*. *file descriptor* yang digunakan dalam model POSIX merupakan indeks dari *file* yang dibuka dalam *linked list*.

Gambaran dari struktur *virtual file system* ada di gambar 2.5 :



Gambar 2.5 Struktur Virtual File System

2.1.4 Interprocess Communication

Linux menyediakan bentuk-bentuk dalam IPC sebagai berikut:

- *Signal*; merupakan pesan yang dikirim secara asinkronus pada proses
- *Wait queue;* mekanisme untuk membuat status proses menjadi *sleep* ketika menunggu operasi selesai.
- *File lock*; mekanisme yang digunakan proses untuk menentukan area dari *file*, atau seluruhnya menjadi *read-only* pada semua proses kecuali yang memegang *file lock*.

- *Pipe* dan *Named pipe*; mekanisme koneksi transfer data secara bidireksional antara dua proses baik secara eksplisit membuat koneksi *pipe* atau dengan berkomunikasi melalui *named pipe* yang berada dalam *file system*.
- *System V IPC* (*semaphore*, *message queue*, dan *shared memory*)
- *Unix domain socket*; mekanisme komunikasi data yang sama dengan model pada INET *socket*.

Signal diimplementasikan dalam struktur *task_struct* field *signal*. Setiap *signal* direpresentasikan sebagai bit, sehingga dalam Linux hanya di-*support* terbatas. Field *blocked* digunakan untuk memblokir *signal* pada proses. Struktur *wait_queue* berisi *pointer* ke *task_struct* yang bersangkutan.

File lock disimpan di struktur *file_lock*. Struktur ini berisi *pointer* ke *task_struct* proses yang bersangkutan, *file descriptor* dari *file* yang di-*lock*, proses dalam status *wait* yang menunggu *lock* untuk dilepaskan, dan area dari *file* mana yang di-*lock*.

Pipe, baik itu yang punya nama dan tidak, direpresentasikan sebagai *inode*. *Inode* ini menyimpan informasi spesifik dalam struktur *pipe_inode_info*. Struktur ini berisi *wait queue* dari proses yang diblokir, *pointer* ke *memory page* yang digunakan untuk *buffer*, ukuran dari data dalam *pipe*, dan jumlah proses yang pada saat itu sedang membaca dan menulis dari/ke *pipe*.

Semua objek dari *system* V IPC dibuat dalam *kernel*, dan masing-masing mempunyai ijin akses. Ijin akses disimpan dalam struktur *ipc_perm*. *Semaphore* direpresentasikan dalam struktur *sem*, yang menyimpan nilai dari *semaphore* dan

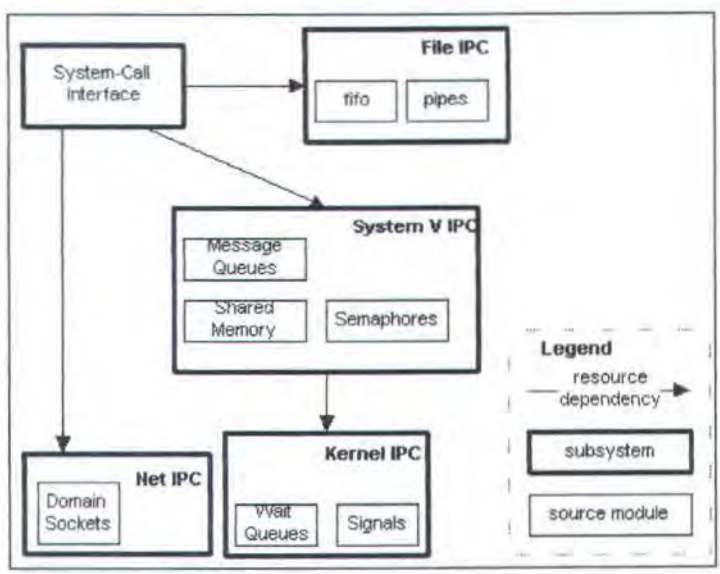
PID dari proses yang melakukan operasi. *Array semaphore* ada di struktur *semid_ds*, yang menyimpan ijin akses, waktu terakhir *semaphore* digunakan, *pointer* ke *semaphore* pertama di *array*, dan antrian proses yang diblokir.

Message queue dibuat di struktur *msquid_ds*, menyimpan informasi tentang manajemen dan kontrol.

Shared memory diimplementasikan berdasarkan struktur *shmid_ds*, yang berisi informasi yang sama dengan *msquid_ds*.

Unix domain socket berdasarkan pada struktur data *socket*.

Struktur dari IPC pada Linux digambarkan di gambar 2.6:



Gambar 2.6 Struktur IPC

2.1.5 Network interface

Network interface pada Linux menyediakan konektivitas antar mesin, dalam model komunikasi *socket*. Dua tipe *socket* disediakan: BSD *socket* dan INET *socket*. BSD *socket* diimplementasikan menggunakan INET *socket*.

Untuk protokol *transport* menggunakan dua macam model. Ada yang *unreliable, message based* seperti protokol UDP dan *reliable, streamed* seperti protokol TCP. Semua diimplementasikan diatas protokol IP. INET *socket* diimplementasikan di atas protokol transport dan protokol IP.

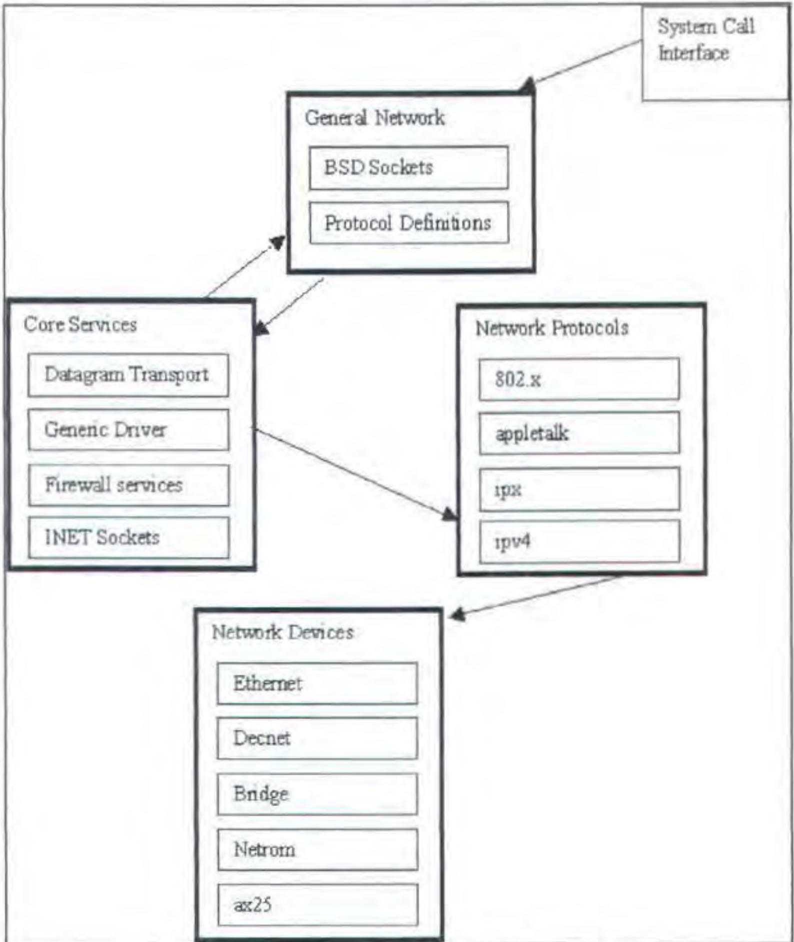
Protokol IP berdiri diatas *device driver*. *Device driver* menyediakan tiga tipe koneksi: *serial line* (SLIP), *parallel line* (PLIP), dan koneksi ethernet. Protokol ARP menjembatani antara IP dan *driver* ethernet. *Address resolver* digunakan untuk translasi antara alamat *logical* IP dan alamat fisikal ethernet.

Implementasi BSD *socket* direpresentasikan dengan struktur *socket*. Berisi *field* tentang tipe dari *socket* (*streamed* atau *datagram*), dan *state* dari *socket* (terkoneksi atau tidak). *Field* yang menyimpan status/*flag* digunakan untuk memodifikasi operasi pada *socket*. Setiap BSD *socket* berelasi dengan sebuah *inode*.

Struktur *sk_buff* digunakan untuk menangani komunikasi paket secara individu. *Buffer* ini merujuk pada *socket* yang menggunakannya, berisi waktu terakhir data ditransfer, dan *link* ke semua paket yang berhubungan dengan *socket* yang digunakan dalam bentuk *linked list*. Alamat sumber dan tujuan, informasi *header*, dan data paket disimpan dalam *buffer* ini.

Struktur *sock* merujuk pada informasi spesifik dari INET *socket*. Isi dari struktur adalah perhitungan pembacaan dan penulisan pada *memory* yang di-*request* oleh *socket*, *sequence number* yang dibutuhkan oleh protokol TCP, status/*flag* untuk mengubah perilaku dari *socket*, manajemen *buffer*, dan *wait*

queue. *Pointer* ke daftar fungsi yang secara spesifik pada setiap protokol juga disediakan dalam struktur *proto*. Struktur dari *network interface* ada gambar 2.7:

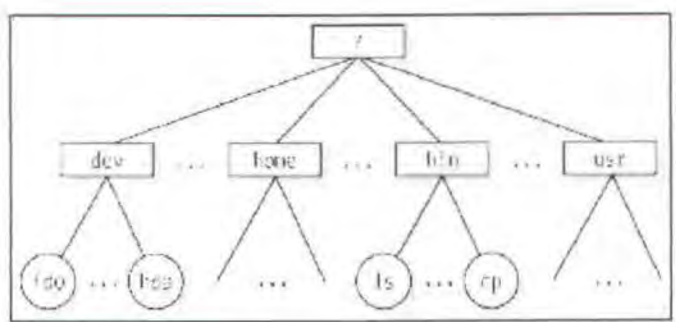


Gambar 2.7 Struktur Network Interface

2.2 PENGENALAN FILE SYSTEM UNIX

2.2.1 File

File merupakan wadah informasi terstruktur sebagai sekumpulan *byte*; *kernel* tidak menginterpretasi isi dari *file* [4]. Dari sudut pandang *user*, *file* diorganisasi dalam bentuk *tree*, seperti terlihat pada gambar 2.8:



Gambar 2.8 Contoh directory tree

Semua *node* dalam *tree*, kecuali *leave*, dinyatakan sebagai nama *directory*. *Directory* yang merujuk pada akar/*root* dari *tree* dinamakan *root directory* dan dinamakan dengan garing (/).

Pada setiap proses terdapat *current working directory*; termasuk dalam konteks eksekusi proses, dan menujukkan *directory* yang saat itu digunakan oleh proses. Untuk menunjukkan secara spesifik ke *file*, proses menggunakan *pathname*, yang berisi urutan nama *directory* yang dipisah oleh garing yang menuju ke lokasi *file*. Jika *item* yang pertama adalah garing, maka *pathname* dinyatakan sebagai *absolute*, karena merupakan lokasi awal dari *root directory*. Selain itu dinamakan sebagai *relative*, karena lokasi awalnya adalah *current directory* milik proses. Selain itu terdapat notasi "." dan ".." yang menunjukkan *current directory* dan *directory parent*.

2.2.2 Hard link dan Soft link

Nama *file* yang terdapat dalam *directory* dinamakan sebagai *hard link file* atau sederhananya *link*. *File* yang sama bisa mempunyai beberapa *link* didalam *directory* yang sama atau berbeda, jadi dapat mempunyai beberapa nama *file*.

Perintah Unix:

$ ln f1 f2

digunakan untuk membuat *hard link* baru yang mempunyai *pathname* f2 untuk *file* yang diidentifikasikan sebagai *pathname* f1.

Hard link mempunyai dua batasan:

- *User* tidak dapat membuat *hard link* untuk *directory*. Ini memungkinkan *tree* menjadi bentuk *cyclic*, sehingga menjadi tidak mungkin untuk mencari *file* menurut namanya.
- *Link* dapat dibuat antar *file* yang masih dalam satu *file system*. Ini menjadi masalah serius, karena *system* Unix saat ini dapat berisi beberapa *file system* yang berada di *disk*/partisi yang berbeda, dan *user* tidak sadar tentang hal tersebut.

Untuk mengatasi masalah ini, maka diperkenalkan istilah *soft link* (juga disebut sebagai *symbolic link*). *Symbolic link* merupakan *file* yang berisi *pathname* ke *file* lain. *Pathname* dapat merujuk pada *file* di *file system* lain; bahkan dapat merujuk pada *file* yang tidak berada/noneksis.

2.2.3 Tipe-tipe File

File pada Unix mempunyai beberapa tipe : *regular file*, *directory*, *symbolic link*, *block-oriented device file*, *character-oriented device file*, *pipe* dan *named pipe* (dikenal sebagai FIFO), *socket*.

2.2.4 File Descriptor dan Inode

Unix membedakan secara jelas antara isi dari *file* dan informasi tentang *file*. Dengan pengecualian *file device* dan spesial, setiap *file* berisi urutan karakter-

karakter. *File* tidak menyertakan informasi kontrol, seperti panjang dan penanda akhir *file*/EOF.

Semua informasi yang dibutuhkan oleh *file system* untuk menangani *file* disertakan dalam struktur data *inode*. Setiap *file* mempunyai *inode* sendiri, yang digunakan *file system* untuk mengidentifikasikan *file*.

2.2.5 Ijin akses dan mode file

User dari *file* dapat dikategorikan menjadi 3 kelas:

- *User* yang menjadi pemilik *file*
- *User* yang berada dalam grup yang sama, tidak termasuk pemilik
- *User* lainnya (*other*)

Terdapat tiga tipe ijin akses; *read, write,* dan *execute*. Sehingga ijin akses ke *file* terdapat 9 kombinasi. Tiga penanda/*flag* tambahan seperti *suid* (*Set User ID), sgid (Set Group ID),* dan *sticky*, mendefinisikan mode *file*. Penanda ini digunakan ketika *file* merupakan *executable*:

- *suid*; Proses menjalankan *file* normalnya sebagai *User* ID dari pemilik proses. Namun jika penanda *suid* digunakan, maka proses akan mempunyai UID dari pemilik *file*
- *sgid*; Jika proses menjalankan *file* dengan penanda *sgid*, maka proses mendapatkan ID dari grup *file*
- *sticky*; *file* diberi penanda *sticky* akan meminta *request* ke *kernel* untuk tetap menyimpan program di *memory* walaupun eksekusi berakhir

2.2.6 System call untuk penanganan file

Ketika *user* mengakses isi baik itu *file* dan *directory*, sebenarnya *user* sedang mengakses data yang disimpan di *block device hardware*. Karena proses di *User Mode* tidak dapat berinteraksi langsung dengan komponen *hardware*, setiap operasi *file* dikerjakan dalam *Kernel Mode*. Sehingga, *system* operasi Unix mendefinisikan *system call* yang berhubungan dengan operasi *file*.

2.3 USER MODE LINUX

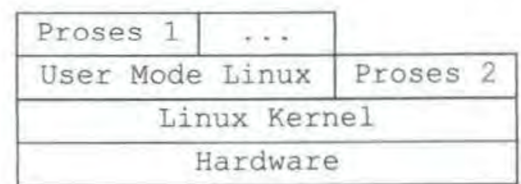
User Mode Linux (UML) merupakan suatu mesin *virtual* yang dapat digunakan untuk menjalankan proses pada Linux secara aman tanpa beresiko merusak lingkungan Linux yang sebenarnya [6]. UML secara umum bisa disebut sebagai *kernel* dalam *kernel*. *Kernel* dalam UML benar-benar mirip dengan *kernel* yang menjalankan sebuah *system* operasi linux. Bisa diubah kodenya, di-*compile*, di-*start*, di-*restart*, dan di-*shutdown*.

Kernel linux yang biasa berjalan dan berkomunikasi di atas hardware PC. Strukturnya bisa diilustrasikan pada gambar 2.9.

Proses 1	Proses 2	...
Linux Kernel		
Hardware		

Gambar 2.9 Struktur layout Linux

Sedangkan UML saat dijalankan akan menciptakan sebuah proses untuk *instance*-nya sendiri-sendiri di atas *kernel* linux yang asli. Strukturnya hampir sama, seperti pada gambar 2.10, yaitu:



Gambar 2.10 Struktur layout UML di Linux

Pada sebuah UML dimungkinkan untuk ditambahkan *resource hardware* ataupun *software* secara *virtual*.

Penyimpanan data yang merupakan *block device* dari *virtual* mesin tersebut hanya berupa satu *file* tunggal untuk satu mesin *virtual*. Satu *file* merupakan *root file system* yang berjalan di UML.

Berikut adalah beberapa kegunaan dari *User Mode* Linux:

- *Virtual hosting*
- *Kernel development* dan *debugging*
- *Process debugging*
- Secara aman mencoba *kernel* linux terbaru
- Mencoba distribusi terbaru linux
- Jaringan *virtual*
- *Disaster recovery practice*, dan sebagainya

Berikut adalah *hardware* yang di-*support* UML:

- *Block device*. Pada UML digunakan satu *file* pada *host* yang berisikan *file system*. *File* ini akan menjadi *block device* di UML dan akan di-*mount* seperti halnya *file system* pada *disk* fisik seperti pada *kernel* umumnya.

- *Console* dan *serial lines*. Merupakan *interface host* termasuk *file descriptor*, *ptys*, *ttys*, *pts* devices, dan xterm.
- *Network device*, SCSI *device*, *sound* device, PCI *device*, dan USB *device*.

Alasan-alasan mengapa perlu memakai UML:

- Dapat menjalankan sebuah UML meski tidak menggunakan *login root*.
- Dapat melakukan *debugging* terhadap UML sama seperti proses-proses lainnya.
- Dapat menjalankan gprof (*profiling*) dan gcov (*coverage testing*).
- Dapat bermain-main *kernel* tanpa merusak apapun.
- Dapat menggunakannya untuk mencoba aplikasi-aplikasi baru.
- Dapat mencoba mengembangkan sebuah *kernel* baru secara aman.
- Dapat menjalankan banyak distro yang berlainan secara bersamaan.

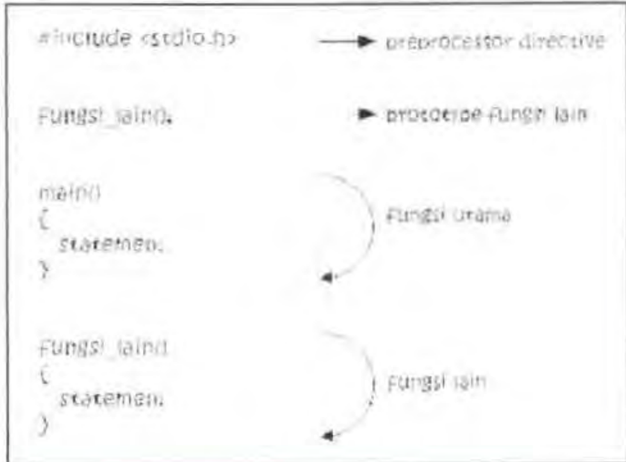
2.4 PEMROGRAMAN C

2.4.1 Struktur Data

Program bahasa C adalah suatu program terdiri dari satu atau lebih fungsi-fungsi. Fungsi utama dan harus ada pada program C yang dibuat adalah fungsi main(). Fungsi main() ini adalah fungsi pertama yang akan diproses pada saat program di-*compile* dan dijalankan, sehingga bisa disebut sebagai fungsi yang mengontrol fungsi-fungsi lain. Karena struktur program C terdiri dari fungsi-fungsi lain sebagai program bagian (*subroutine*), maka bahasa C biasa disebut sebagai bahasa pemrograman terstruktur. Cara penulisan fungsi pada program

bahasa C adalah dengan memberi nama fungsi dan kemudian dibuka dengan kurang kurawal buka ({) dan ditutup dengan kurung kurawal tutup (}).

Fungsi-fungsi lain selain fungsi utama bisa dituliskan setelah atau sebelum fungsi utama dengan deskripsi *prototype* fungsi pada bagian awal program. Bisa juga dituliskan pada *file* lain yang apabila ingin memakai atau memanggil fungsi dalam *file* lain tersebut, harus menuliskan *header file*nya, dengan *preprocessor directive* #include. *File* ini disebut *file* pustaka (*library file*). Struktur program bahasa C terlihat pada gambar 2.11:



Gambar 2.11 Struktur program C

MILIK PERPUSTAKAAN INSTITUT TEKNOLOGI SEPULUH - NOPEMBER

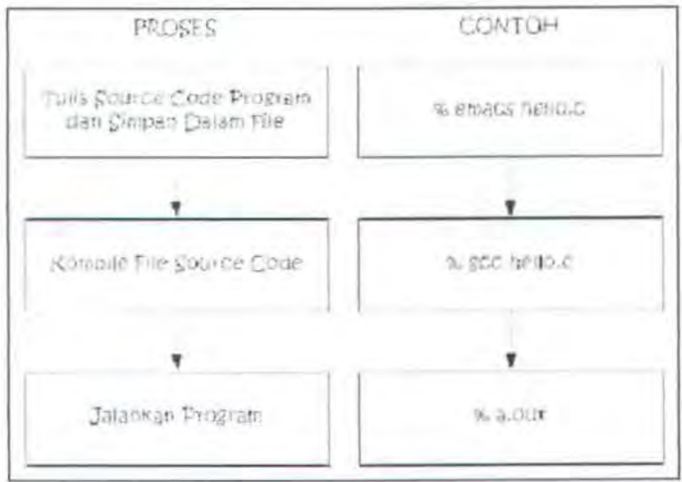
2.4.2 Proses pembuatan

1. Tulis *source code* program bahasa C dengan menggunakan *text editor*, kemudian simpan dalam sebuah *file*. *Text editor* disini bisa merupakan aplikasi *vi*, *emacs* yang cukup terkenal.
2. *Compile file* yang berisi *source code* program bahasa C. Kompilasi (*compile*) adalah suatu proses merubah *source code* ke bahasa mesin sehingga bisa dieksekusi (*executable*) atau dijalankan. Dibutuhkan

compiler (*GCC, GNU C Compiler*) dan *shell* (*Bash Shell*) untuk membuat program C

3. Jalankan program yang telah di-*compile*. Setelah di-*compile file* yang berisi *source code*, maka sebagai hasil kompilasi tersebut akan mendapatkan suatu *file* yang bisa dijalankan (*executable file*).

Gambaran lengkapnya dapat dilihat di gambar 2.12.



Gambar 2.12 Alur proses pembuatan program

BAB III

PERANCANGAN PERANGKAT LUNAK

Pembahasan bab ini terdiri dari 3 bagian, yaitu meliputi desain *historical file*, arsitektur *file system*, dan perancangan *userspace utility* bagi *user*. Perangkat lunak yang dirancang meliputi pembuatan *file system* baru yang bekerja pada *kernel* Linux dan *utility* pada tingkat level *user* yang berfungsi untuk memodifikasi atribut-atribut *historical* di *file*.

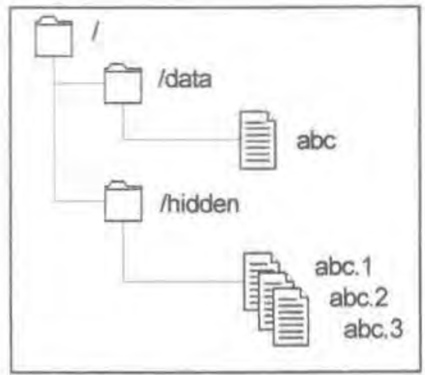
3.1 DESAIN HISTORICAL FILE

Historical file merupakan berkas-berkas catatan sejarah perubahan yang pernah terjadi dalam sebuah *file*. Sejarah perubahan ini berguna untuk mengembalikan versi *file* sebelumnya, mengumpulkan berkas-berkas perubahan yang penting saat-saat tertentu, atau melacak aktifitas penyusup (*intruder*) pada *file system*. Ketika Unix mulai populer, *user* banyak menginginkan bentuk *historical/versioning* pada *file system*.

CVS adalah salah satu *utility* yang popular untuk melakukan *versioning*. CVS mengijinkan grup *user* untuk mencatat perubahan pada *file* didalam sebuah penyimpanan/*repository*, dan memperoleh kembali versi yang sebelumnya tersimpan dalam CVS *repository*. Namun, CVS berbeda dengan *historical file system* yang bekerja secara transparan pada semua aplikasi, sehingga dalam mengakses *historical file*, tidak perlu melakukan modifikasi/perlakuan khusus kepada aplikasinya.

3.1.1 Teknik penyimpanan historical file

Di *historical file system*, *head/current file* (yang menyebabkan terjadinya versi) disimpan sebagai *regular file*, sehingga mempertahankan karakteristik akses yang normal pada *file system*. Desain ini mencegah turunnya *performance* disaat mengakses *current file*. Setiap versi dari *file* disimpan dalam satu *file* yang berbeda, dan ditempatkan di *directory* khusus penyimpanan semua *file-file versi* yang bekerja pada satu *file system*, seperti terlihat pada gambar 3.1. Pada contoh tersebut, *file-file* versi dari abc disimpan di *directory hidden*, dan diberi nama sesuai dengan *current file* disertai penambahan nomor versinya. Kumpulan dari *file-file* versi yang dibuat dinamakan sebagai *version set*.



Gambar 3.1 Letak file dan versinya pada hirarki

Untuk membuat *user* dan *administrator* menjadi fleksibel, *historical file* mendukung beberapa *policy* penyimpanan (*retention*). Aturan-aturan penyimpanan ini menentukan berapa banyak versi *file* yang akan disimpan. Sebuah *utility* pada mode *user* memudahkan *user* dalam memanipulasi aturan-aturan penyimpanan *version set* dan juga memantau masing-masing *version set file* yang lama. Semua fungsionalitas dari *utility* ini menghindari *user* untuk

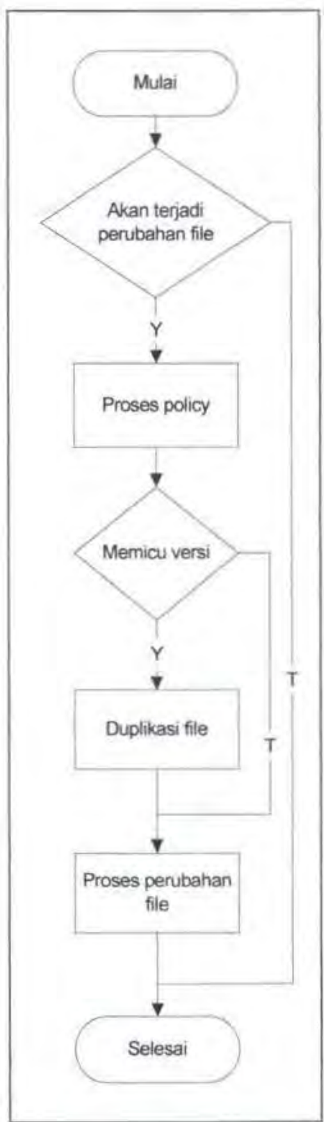
memodifikasi aplikasi (*editor*) dalam mengakses sebuah versi *file* dan juga memberikan fleksibilitas bagi *user* dalam bekerja pada *historical file*.

3.1.2 Mekanisme historical file

Versi *file* yang dihasilkan menggunakan model *copy-on-write*. Setiap perubahan pada *file* akan menghasilkan versi *file* yang baru. Terdapat 3 tipe operasi yang dapat memicu sebuah versi : penulisan (melalui *write* atau *mmap write*), *truncate*, dan perubahan nama file (*rename*). Gambar 3.2 menjelaskan alur proses sebelum terjadi perubahan *file*.

Setiap operasi penulisan bila sesuai dengan proses *policy* akan menghasilkan satu versi *file* yang baru. Sebelum terjadi penulisan pada *current file* (pada contoh ini *file* abc), dilakukan proses penduplikasian *current file*. Hasil duplikasi diletakkan di *directory version set* (jika pada contoh terdapat di *directory hidden*), dan diberi nama sesuai nama *current file* bersama dengan penomoran versi yang selanjutnya.

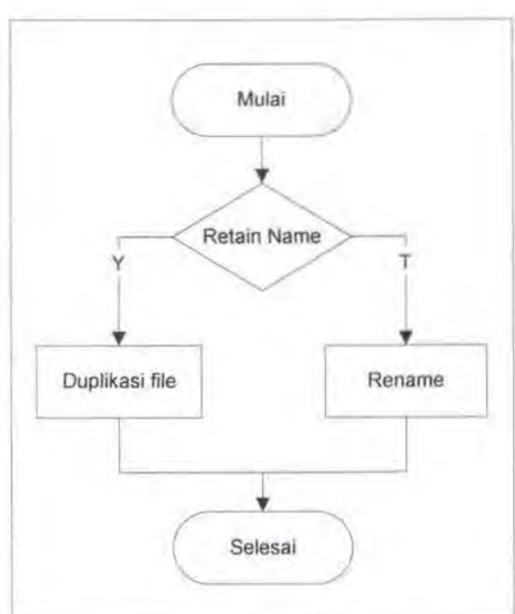
Proses *truncate* penanganannya hampir sama dengan proses penulisan. Jika dalam *policy* memicu untuk terciptanya versi, maka dilakukan proses duplikasi *current file* untuk menghasilkan versi *file* yang baru. Baik operasi *write* ataupun *truncate*, setiap pembukaan dan penutupan sebuah *file*, hanya satu *file* versi yang akan dibuat. Pendekatan ini untuk menyelaraskan perilaku dari aplikasi *editor* dalam melakukan manipulasi *file*. Karena dimungkinkan sebuah aplikasi menuliskan datanya dalam bentuk serangkaian pemanggilan operasi *write*, atau hanya sekali pemanggilan operasi *write*, meskipun perubahan data yang dihasilkan adalah sama.



Gambar 3.2 Alur proses perubahan file

Untuk perubahan nama *file* (*rename*), terdapat dua opsi yang membedakan alur perilaku proses. Bila nama dari *current file* ingin dipertahankan, maka setiap perubahan nama *file* (*rename*), *file* akan terduplikasi ke nama yang baru (lebih mirip sebagai proses penduplikasian *file*). Atribut *historical* masih dimiliki oleh pemilik nama *file* lama. Namun jika tidak ingin dipertahankan, maka tidak ada

proses penduplikasian *file*, hanya proses pergantian nama *file* lama ke nama *file* yang baru (operasi *rename* pada umumnya). Mempertahankan nama *file* berguna untuk aplikasi *editor* yang melakukan perubahan pada *file*, dengan menggunakan *temporary file* untuk menyimpan perubahan *file*. Dalam kasus aplikasi *vi* dalam manipulasi *file*; pada saat penulisan, *vi* merubah nama *file* yang di-*edit* (digunakan untuk *backup file*), membuka *file* baru (dengan opsi *truncate* jika *file* eksis) dengan nama lama, kemudian berlanjut dengan menulis data. Gambar 3.3 memperlihatkan alur proses dari perubahan nama *file*.



Gambar 3.3 Alur proses perubahan nama file

3.1.3 Policy di historical file

Sebelum terjadinya perubahan *file*, terdapat *policy* yang mengatur terjadinya sebuah versi [3]. *Policy* yang dirancang pada *historical file* mencakup 3 hal:

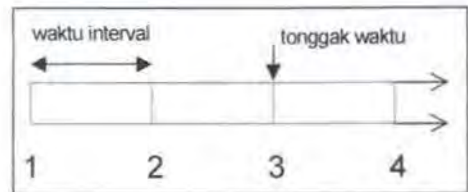
- Jumlah versi: *user* dapat mengatur nilai minimum dan maksimum dari versi yang dibuat. Aturan ini memungkinkan beberapa versi masih tersimpan.
- Waktu: *user* dapat mengatur tanggal minimum dan maksimum dari versi *file* yang masih perlu disimpan. Ini untuk memastikan bahwa versi masih tersimpan untuk beberapa periode.
- Kapasitas: user dapat mengatur jumlah kapasitas makimum dari *version set*. *Policy* ini mengijinkan banyak versi *file* untuk ukuran kecil, tetapi tidak mengijinkan satu *file* ukuran besar menghabiskan kapasitas disk.
- *Trigger*: *user* dapat menentukan pemicu terjadinya versi dari *current file*. Terdapat dua model pemicu; berdasarkan jumlah perubahan *file*, dan interval waktu.

Sebuah *file* versi tidak pernah dibuang/hapus jika dibuang maka akan melanggar *policy* minimum. Nilai minimum memiliki prioritas lebih tinggi daripada nilai maksimum. Jika sebuah versi tidak melanggar nilai minimum dan melebihi nilai maksimumnya, maka versi akan dibuang dari *version set*.

Menyediakan kombinasi minimum dan maksimum dari sebuah versi akan berguna saat digunakan. Misalnya, *user* dapat menentukan bahwa jumlah versi yang harus tersimpan berkisar antara 10 – 100 *file* dan 2 – 5 hari. *Policy* ini memastikan 10 *file* versi tetap disimpan dan setidaknya *file* file yang tersimpan berumur 2 hari. Nilai minimum memastikan *file* versi tidak dihapus secara dini, dan nilai maksimum menentukan kapan *file* versi harus dihapus.

Setiap *user* dan *administrator* dapat mengatur *policy* yang terpisah untuk setiap ukuran *file*. *Policy* ukuran *file* berguna untuk memastikan bahwa ukuran *file-file* yang besar tidak terlalu memakan kapasitas *disk* untuk sebuah *version set*-nya. Dengan adanya total kapasitas maksimum dari *version set* dari satu *file*, *file* dengan ukuran yang besar akan memperoleh jumlah *file* versi lebih sedikit daripada *file* dengan ukuran yang kecil.

Untuk pengaksesan/perubahan data pada satu *file* secara bersamaan dan terus-menerus dapat mengakibatkan munculnya banyak versi *file*. Untuk membatasi frekuensi munculnya banyak versi, perlu adanya aturan dalam men-*trigger* terjadinya versi. Ada 2 macam bentuk *trigger*, yakni berdasarkan interval waktu dan jumlah penulisan data. Dengan *policy* tersebut, frekuensi jumlah munculnya versi *file* yang berlebihan dapat terkontrol. Untuk *policy* interval waktu, jika terjadi perubahan *file* melebihi tonggak waktu yang ditentukan, maka akan memicu versi baru. Tonggak waktu terus ter-*update* sesuai dengan jumlah interval waktu saat menghasilkan sebuah versi (seperti pada gambar 3.4). *Policy* untuk jumlah penulisan data, menentukan ambang batas jumlah penulisan pada *file*. Jika jumlah penulisan pada *file* melebihi ambang batas dari nilainya, maka akan memicu terjadinya sebuah *file* versi.



Gambar 3.4 Interval waktu untuk memicu terjadinya versi

3.2 ARSITEKTUR FILE SYSTEM

3.2.1 Pendekatan desain arsitektur

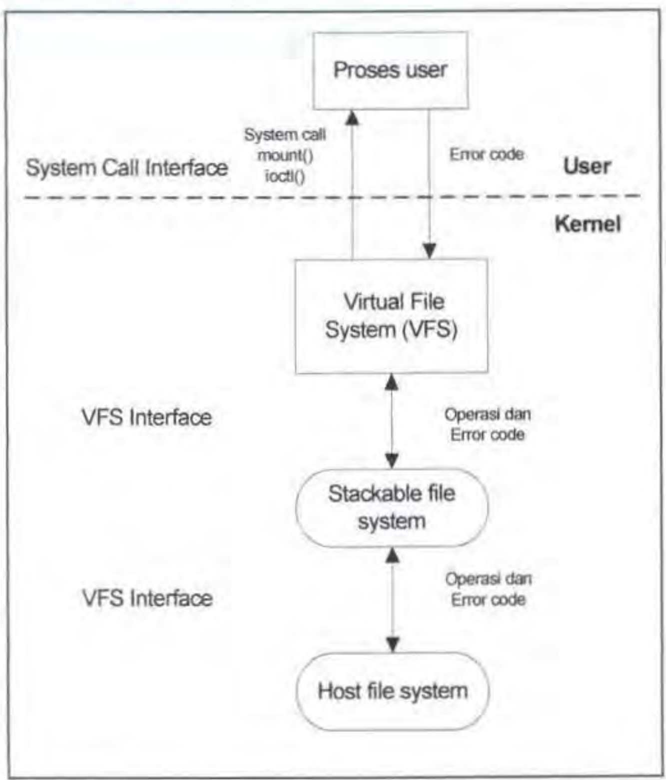
Dalam mendesain *file system* yang baru, terdapat dua pilihan pendekatan untuk arsitektur *file system*. Yang **pertama** adalah memodifikasi *file system* yang telah ada di *source kernel* Linux, seperti Ext2, Reiserfs, Minix dll. Dengan memodifikasi *file system* yang sudah ada, akan terbentuk *file system* baru dengan fungsionalitas yang diinginkan. Yang **kedua**, adalah dengan memanfaatkan teknik *file system stacking*. Teknik ini memungkinkan membuat fungsionalitas yang diinginkan berjalan di atas *file system* yang sudah ada.

Pendekatan yang pertama mempunyai beberapa kendala. Pertama, jika dengan mengambil Ext2 untuk menjadi *file system* baru dengan fungsionalitas *historical file*, maka terjadi kemungkinan struktur data (*metadata*) yang sifatnya persisten (tersimpan dalam *disk*) akan berbeda dengan Ext2. Perbedaan ini dapat menyebabkan masalah yang akan muncul di aplikasi *utility* untuk manajemen *disk*. Aplikasi seperti *fsck*, yang melakukan pengecekan dan perbaikan *file system* dan juga *mkfs*, yang berguna untuk membuat *file system* pada *disk*; perlu di modifikasi untuk kesesuaian struktur data fisik yang telah mengalami perubahan. Kendala kedua adalah bahwa dengan pendekatan yang diambil, nantinya tidak dapat mencoba untuk menjalankannya pada *system* yang sudah berjalan. Karena merupakan *file system* yang berbeda, maka untuk menjalankannya, perlu membuat *file system* baru pada *disk*, dan memulai instalasi awal system, jika *file system* tersebut dijadikan sebagai *root file system*. *Root file system* merupakan *file system* yang pertama kali di-*mount* pada saat *booting kernel*.

3.2.2 Stackable file system

Pada akhirnya diambillah pendekatan kedua, yakni menggunakan model *stackable file system* (*Stackablefs*) [1], tetapi dengan modifikasi ke arah teknik pengijeksian (*hooking*) fungsi-fungsi *historical file* pada *native file system* (untuk selanjutnya juga dinamakan sebagai *Hostfs*). Dalam model *stackable file system*, *file system* bekerja dalam bentuk *layer-layer*. *Hostfs* menempati *layer* di bawah *Stackablefs*. *Stackablefs* bekerja secara transparan dalam mengubah perilaku *file system*, tanpa disadari oleh *Hostfs*. Jika di lihat dari sisi *user*, dapat diketahui bahwa disk ter-*mounting* sebagai *Stackablefs*, bukan *Hostfs*. *Stackable file system* memungkinkan dapat bekerja bertingkat (*layer* diatas *layer*) dengan tipe *stackable file system* yang berbeda.

Seperti terlihat pada gambar 3.5, aliran proses operasi *file* pada *stackable file system* dimulai dari sisi proses *user* menuju ke *layer* paling bawah dari *file system*. Proses di mode *user* umumnya mengakses *file system* dengan menjalankan *system call*, yang mengaktifkan ke mode *kernel*. VFS di *kernel* kemudian menerjemahkan *system call* ke operasi VFS, dan memanggil fungsi *file system*. Jika yang dipanggil adalah *Stackablefs*, selanjutnya olehnya akan memanggil fungsi *file system* di layer bawahnya. Disaat aliran proses mencapai pada tingkat *file system* yang terbawah, *error code* dan nilai kembali (*return value*) mulai bergerak naik ke *layer* atasnya, sampai menuju proses *user*.



Gambar 3.5 Informasi dan aliran proses pada stackable file system

Dikarenakan *stackable file system* menggunakan model *layering file system*, masing-masing *file system* baik itu *Stackablefs* dan juga *Hostfs* mempunyai *page cache* sendiri [2]. *Page cache* disini merupakan *disk cache*, yakni mekanisme dari *system* untuk menyimpan informasi yang seharusnya tersimpan dalam *disk*, sehingga akses selanjutnya ke informasi tersebut, tidak perlu akses ke *disk*. Isi data *cache* di masing-masing *file system* akan berbeda jika *layer* yang berada di atasnya memanipulasi data *file* dari *layer* di bawahnya. Namun jika tidak terdapat manipulasi data, maka tidak ada perbedaan dalam data *cache*; dan ini mengakibatkan *redudancy* data, sehingga untuk pengaksesan data untuk banyak *file* akan cukup menghabiskan *memory* di *system*.

3.2.3 Teknik injeksi/hooking fungsi VFS (XPackfs)

Arsitektur *file system* yang dibuat di Tugas Akhir ini sedikit berbeda secara teknis. Jika *stackable file system* menjadikan *file system* dalam bentuk *layer-layer*, maka yang diimplementasikan ini menggunakan cara injeksi fungsi-fungsi VFS. Meskipun sama-sama berjalan diatas *Hostfs*, *file system* yang dibuat (disebut juga sebagai *XPackfs*) cenderung menempel ke *Hostfs*, meskipun ada salah satu karakteristiknya yang sama seperti *stackable file system*, yakni keberadaannya tidak disadari oleh *Hostfs*.

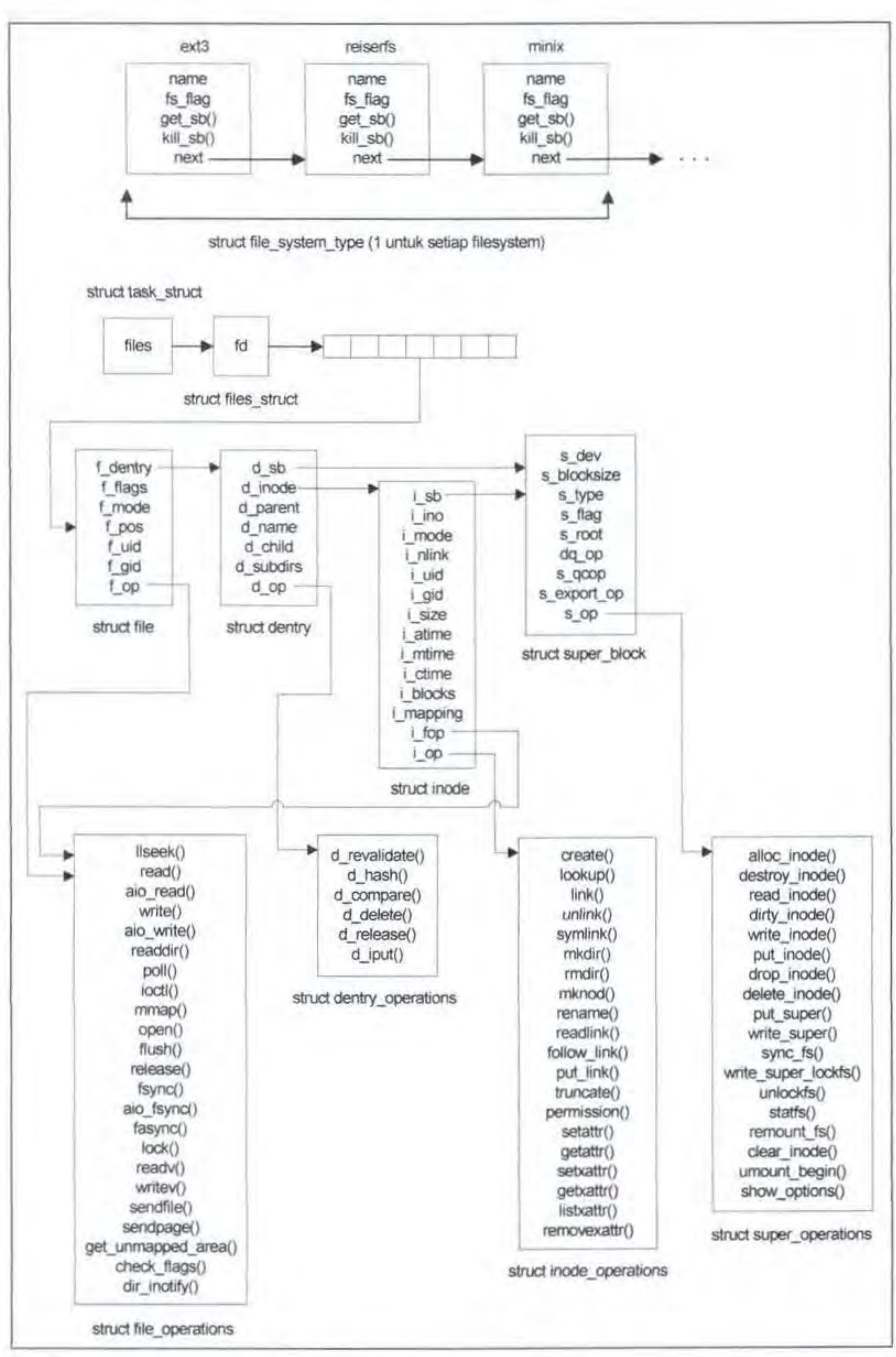
Virtual File System (VFS) merupakan *subsystem* dari *kernel* yang salah satu kegunaannya adalah mentransformasi pemanggilan operasi *file* dari *system call* ke bentuk fungsi standar yang digunakan oleh beberapa *file system*. Jadi, *kernel* Linux tidak mengimplementasikan operasi-operasi *file* seperti *read()* atau *ioctl()* dalam bentuk *hardcoding*. Tetapi, *kernel* menggunakan bentuk *pointer* untuk setiap operasinya; *pointer* dibuat untuk menunjuk ke fungsi spesifik dari *file system* yang diakses (*interface*). Tabel 3.1 menunjukkan *system call* yang ditangani oleh VFS, untuk kemudian dilanjutkan dengan memanggil fungsi-fungsi spesifik *file system* jika diperlukan.

Seperti yang diilustrasikan gambar 3.6, masing-masing struktur data dari objek-objek VFS (*file, dentry, inode, superblock*) mempunyai *field* yang berisi *pointer* ke fungsi-fungsi untuk memanipulasi objek tersebut. Untuk objek *file*, tersedia *pointer* (terdeklarasi sebagai *struct file_operations*) ke operasi yang berhubungan dengan *file*. Fungsi manipulasi objek *dentry* terdeklarasi sebagai *struct dentry_operations*, untuk objek *inode* terdeklarasi sebagai *struct*

inode_operations, dan objek *superblock* dideklarasikan sebagai *struct super_operations*. Setiap objek-objek VFS yang digunakan spesifik *file system* mempunyai implementasi fungsi-fungsi VFS sendiri, yang berbeda dengan *file system* lainnya [6].

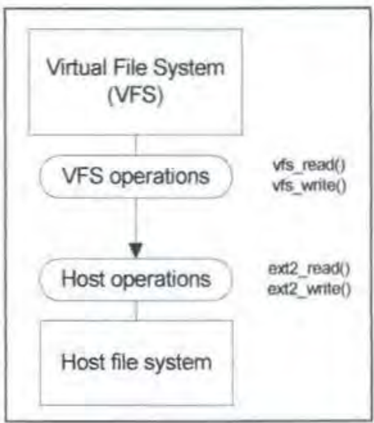
Tabel 3.1 System call yang ditangani VFS

Nama system call	**Deskripsi**
mount, umount	Melakukan mount/unmount file system
sysfs	Memperoleh informasi file system
statfs, fstatfs, ustat	Memperoleh statistik file system
chroot, pivot_root	Mengubah directory root
chdir, fchdir, getcwd	Manipulasi directory current
mkdir, rmdir	Membuat dan menghapus directory
getdents, readdir, link, unlink, rename, ioctl, truncate, ftruncate	Manipulasi directory dan file
readlink, symlink	Manipulasi soft link
chown, fchown, lchown	Modifikasi kepemilikan file
chmod, fchmod, utime	Modifikasi atribute file
stat, fstat, lstat, access	Membaca status file
open, close, creat, umask	Membuka dan menutup file
dup, dup2, fcntl	Manipulasi deskriptor file
select, poll	Notifikasi I/O secara asinkronus
truncate, ftruncate	Mengubah ukuran file
lseek, _llseek	Mengubah pointer file
read, write, readv, writev, sendfile, readahead	Operasi I/O file
pread, pwrite	Mengubah pointer file & mengaksesnya
mmap, munmap, madvise, mincore	Menangani memory mapping file
fdatasync, fsync, sync, msync	Sinkronisasi data file
flock	Manipulasi locking pada file
setxattr, listxattr, getxattr, removexattr	Manipulasi extended attribute



Gambar 3.6 Struktur data utama objek-objek VFS di kernel Linux 2.6.x

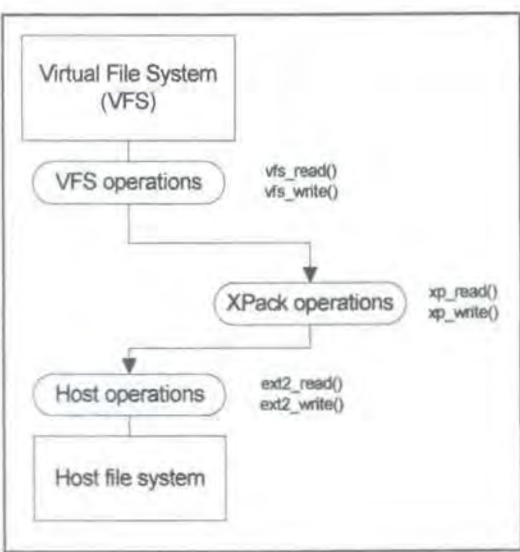
Teknik injeksi ke fungsi-fungsi VFS bersifat ikut turut campur tangan ke jalannya proses di *host*. Saat melakukan proses *mounting* ke *disk*, pertama kalinya *xpack* mencari tipe *native file system* yang didukung oleh *kernel* dan juga yang terdapat di *disk*. Setelah berhasil melakukan proses *mounting native file system* yang disebut sebagai *Hostfs*, *xpack* akan melakukan intervensi ke objek-objek VFS yang dimiliki *host*. Gambar 3.7 memperlihatkan konstruksi normal interaksi fungsi-fungsi VFS dengan *Hostfs*.



Gambar 3.7 Alur proses (normal) dari operasi di file system

Intervensi ke objek-objek VFS dilakukan dengan melakukan substitusi fungsi *host* menjadi fungsi *xpack* (terilustrasikan di gambar 3.8). Informasi fungsi-fungsi yang dimiliki *host* masih tersimpan oleh *xpack*, untuk memudahkan dalam mengembalikan kontrol fungsi berjalan kembali seperti semula. Jika tidak ada modifikasi alur proses yang terjadi di dalam fungsi *xpack*, maka fungsi akan berjalan normal seperti layaknya alur proses yang terjadi di *host* tanpa intervensi dari *xpack*. Sederhananya, hanya meneruskan jalannya proses ke *host*, jika memang tidak terdapat perubahan secara fungsional dari operasi tersebut.

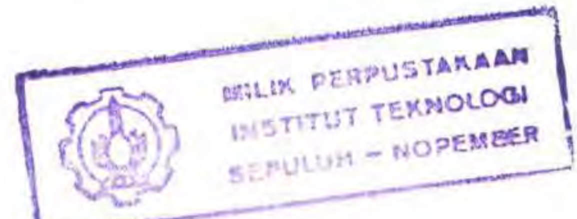
Misalnya dalam contoh; jika dalam implementasi *file system* yang baru tidak diperlukan modifikasi (perubahan fungsionalitas) dalam teknik pembacaan *file*, maka fungsi *vfs_read* akan memanggil *xp_read* milik *xpack* dan melanjutkan saja (*bypass*) seterusnya ke fungsi *ext2_read* milik *host* (dalam hal ini adalah Ext2).



Gambar 3.8 Alur proses (setelah di hook) dari operasi di file system

Dengan adanya intervensi ke *host*, dapat dipastikan akan menemui beberapa masalah. Untuk **pertama** adalah bagaimana manajemen *xpack* untuk menyimpan struktur data miliknya sendiri yang dapat menjadi sebagai titik acuan referensi ke setiap objek VFS. Dan **kedua** adalah bagaimana memonitor informasi fungsi-fungsi yang tersubstitusi oleh *xpack*, tidak diubah/diganti oleh *host* sendiri. Karena *host* juga berhak melakukan perubahan fungsi miliknya sendiri.

Untuk masalah yang pertama, dilakukan analisa ke *source code* dari *subsystem* VFS sendiri di *kernel* Linux. Dari situ, dilakukan usaha untuk mencari solusi bagaimana cara memasukkan struktur data *xpack* ke objek VFS yang akan dimiliki oleh *host*. Pendekatan yang utama adalah bagaimana dalam


MILIK PERPUSTAKAAN
INSTITUT TEKNOLOGI
SEPULUH - NOPEMBER

implementasinya nanti, tidak akan melakukan perubahan yang berarti pada kode *subsystem* VFS. Kemudian ditemukan bahwa pada masing-masing objek VFS terdapat *field* kosong yang khusus digunakan untuk keperluan spesifik *file system*. Yang pada kenyataannya pada tipe *Hostfs* yang didukung (Ext2 dan Reiserfs) tidak ditemukan kode yang mengakses *field* tersebut. Untuk objek *file*, terdapat *field private_data* yang digunakan spesifik *file system* dan tidak digunakan oleh VFS. Untuk objek *dentry* juga dapat ditemukan pada *field d_fsdata*, dan untuk objek *inode* dapat ditemukan di *field u.generic_ip*. Untuk objek *superblock* tidak diperlukan, dikarenakan untuk setiap *file system*, hanya memerlukan satu objek *superblock*. Sehingga mudah bagi *XPackfs* untuk mendapatkan referensi ke objek *superblock* milik *host*.

Masalah yang kedua dari teknik injeksi ini dapat ditangani jika mampu memahami *source code* yang terdapat di Ext2 dan Reiserfs. Dari hasil analisis, tidak ditemukan alur perubahan fungsi tersebut. Tetapi diluar tipe *Hostfs* yang didukung, ada alur perubahan fungsi di *file system* Ext3. Solusi dari masalah tersebut dapat dipecahkan dengan men-*trap* fungsi yang menyebabkan perubahan fungsi. Pada *file system* Ext3, parameter yang dimasukkan ke fungsi *ioctl* dapat mengubah fungsi operasi *address space*. Dengan men-*trap* fungsi *ioctl*, dan memantau apakah terjadi perubahan fungsi, *xpack* dapat mengembalikan lagi fungsi miliknya di objek VFS milik *host* serta melakukan *update* informasi fungsi *host* yang telah mengalami perubahan.

Pendekatan teknik injeksi ini diambil untuk beberapa pertimbangan, jika dibandingkan dengan teknik *stackable file system*. Pertimbangan yang pertama

adalah tidak diinginkannya adanya duplikasi objek (struktur data) seperti yang terjadi pada teknik *stackable file system*. Untuk teknik *stackable file system* ini, setiap *layer* memiliki objek-objek VFS sendiri (*file*, *dentry*, *inode*, *superblock*). Dan juga semua data/*page cache* yang terdapat pada *file* terduplikasi di setiap *layer*nya, jika tidak terdapat modifikasi. Pada teknik injeksi yang tidak di desain untuk model *layer-layer*, *xpack* memanfaatkan objek-objek VFS milik *host*. Dengan adanya duplikasi objek-objek VFS di *stackable file system* menjadikannya sebagai pertimbangan kedua. Masalahnya terjadi pada sinkronisasi antara masing-masing *layer* pada *stackable file system*. Jika tidak terjadi perubahan data pada objek-objek VFS, maka setiap *layer* akan meng-*update* data di objek VFS miliknya sesuai dengan *layer* yang berada dibawahnya. Proses sinkronisasi data ini sebenarnya sedikit menyita waktu di jalannya proses. Dengan pendekatan teknik yang diambil, sinkronisasi data tidak diperlukan.

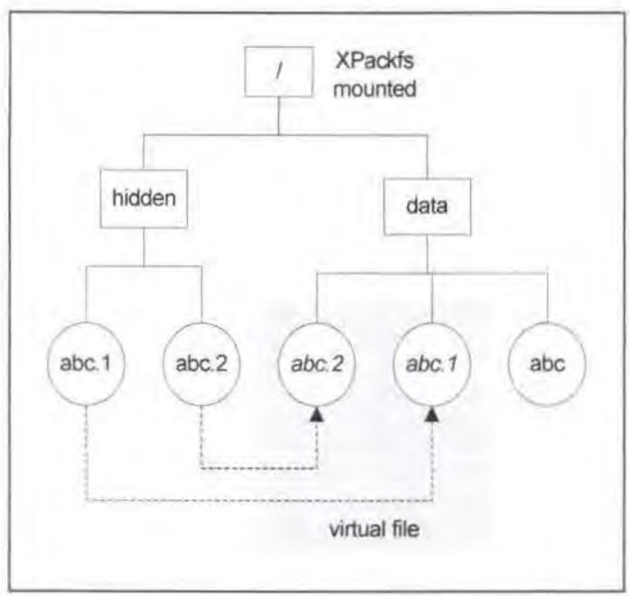
Namun tidak berarti teknik injeksi fungsi-fungsi dari VFS tidak mempunyai beberapa kelemahan dibandingkan dengan teknik *stackable file system*. Oleh karena intervensi *xpack* ke objek struktur data VFS milik *host*, maka harus ada antisipasi apabila *host* melakukan modifikasi datanya. Sebab terdapat kemungkinan akan mengganggu aktifitas *XPackfs*, yang menyebabkan kegagalan pada *system* baik itu nanti secara keseluruhan. Ketergantungannya terhadap *host*, mengakibatkan pembenahan atau perbaikan algoritma dari *xpack* sendiri, jika memang terjadi perubahan alur proses baik dari subsystem VFS dan dari *Hostfs* yang didukung.

3.2.4 Pseudo file system di XPackfs

Desain di *XPackfs* juga mengimplementasikan *pseudo file system* yang berdiri berdampingan dengan *Hostfs*. *Pseudo file system* (*Pseudofs*) digunakan untuk membangun sebuah *virtual/temporary file* agar berada di dalam hirarki *Hostfs*. Untuk kondisi tertentu, perubahan fungsionalitas dari *file system* memerlukan *output* proses berupa *file* yang sifatnya sementara dan dapat mempunyai karakteristik berbeda dari *file* milik *host*; Misalnya saja pada desain proses *historical file*, yaitu untuk menampilkan *file* versi di *directory current file* berada yang sebenarnya letaknya berada di *directory hidden*. Maka perlu membuat sebuah *file* sementara/*temporary* untuk berada di *directory current file*, yang mana nantinya *temporary file* tersebut akan mengacu ke *file* versi yang tersimpan di *directory* tempat semua *version set* disimpan. *File* versi yang ditampilkan di *directory current file* merupakan *virtual file* yang mempunyai karakteristik bahwa *file* tersebut hanya dapat untuk dibaca. Perubahan nama dan juga perubahan data (penulisan) pada *file* versi tidak diperbolehkan. Detail dari manipulasi *file* versi akan dijelaskan di subbab 3.3.

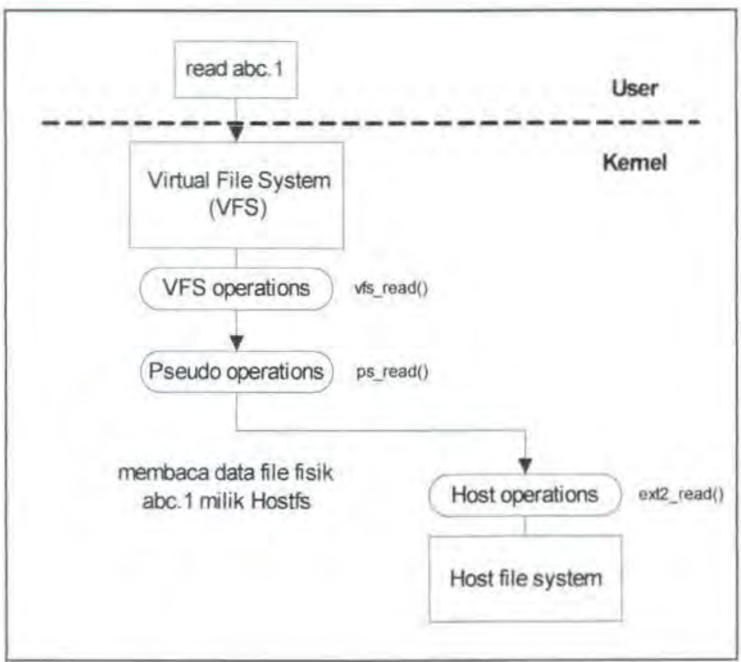
Saat *xpack* menjalankan proses *mounting* ke disk, *xpack* mencari *native file system* (*Hostfs*) yang cocok dengan *file system disk*, serta memuat *Pseudofs* ke dalam *memory*. Jadi sebenarnya *xpack* menjalankan dua *file system* secara bersamaan, yakni *Hostfs* dan *Pseudofs*. *File-file* yang terdapat di *Pseudofs* dapat ditempatkan di hirarki *Hostfs*. Misalnya dalam contoh yang diilustrasikan gambar 3.9, versi *file* abc.1 dan abc.2 yang seharusnya secara fisik berada di *directory*

hidden, di tempatkan di *directory current file* sebagai *virtual file*. Objek *virtual file* nantinya akan merujuk ke *file* fisik dari *host*.



Gambar 3.9 Keberadaan virtual file di hirarki XPackfs

Untuk proses pembacaan data di *virtual file*, *Pseudofs* akan memanggil fungsi *read* milik *host* (terlihat pada gambar 3.10). Proses pembacaan data dari *disk* selalu melibatkan *page cache*. Setiap akses pembacaan dan penulisan data selalu terjadi di *page cache*, yang oleh *subsystem* VFS akan disinkronisasikan ke *disk*. Saat *Pseudofs* membaca *file* milik *host*, yang sebenarnya terjadi adalah duplikasi data pada *page cache*, yaitu *page cache* milik *file host* dan *page cache virtual file* milik *Pseudofs*. Isi *file* yang terdapat di *host page cache*, di transfer ke *Pseudofs page cache*. Namun, karena pembuatan *virtual file* di *XPackfs* tidak terlalu sering terjadi, maka akan sedikit mengkonsumsi *memory*.



Gambar 3.10 Alur proses pembacaan virtual file yang mengacu ke file Hostfs

3.2.5 Teknik penyimpanan data atribut historical di XPackfs

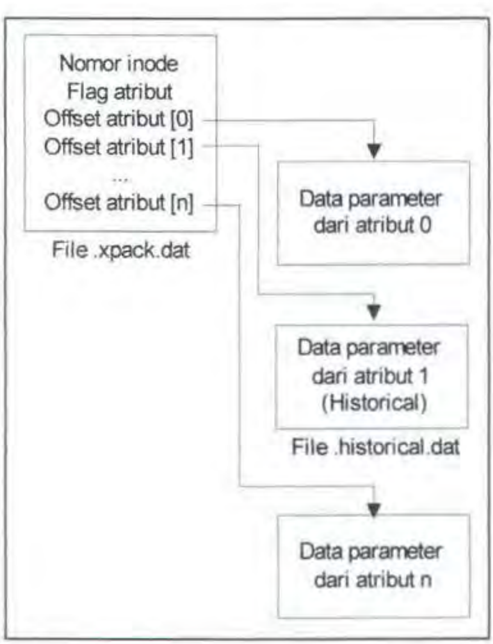
Data atribut yang sifatnya persistent terkait dengan fungsionalitas baru dari *file system* (*XPackfs*), tersimpan dalam satu *file*. Setiap *file* yang ada di *host* dapat memiliki tambahan atribut-atribut khusus dari *xpack*. *XPackfs* dalam hal penyimpanan informasi-informasinya terbagi menjadi 2 macam. Yang **pertama** digunakan untuk menyimpan informasi *file-file* mana yang terdapat di *host* mempunyai atribut khusus *xpack*. Dan yang **kedua** adalah sebagai tempat untuk menyimpan semua parameter-parameter atribut yang dimiliki di setiap *file* beratribut khusus tersebut.

Untuk penyimpanan informasi *file* yang beratribut (tersimpan di *file database* .xpack.dat) mempunyai format sebagai berikut:

- Nomor unik *inode*: karena semua *file* dan *directory* di *file system* direpresentasikan sebagai *inode*, maka setiap *inode* mempunyai nomor unik.
- *Flag* atribut: *XPackfs* mendukung beberapa atribut untuk setiap *file*, jadi tidak hanya atribut *historical*.
- *Array* dari *offset file database* atribut: karena mendukung beberapa atribut, maka data setiap atribut yang tersimpan berada di *file database* atribut yang berbeda. Untuk memudahkan dalam pencarian data, maka perlu mengetahui *offset file* untuk akses datanya.

Namun jika dilihat pada kode *subsystem* VFS, tidak semua *file system* menggunakan mekanisme *inode numbering*. Sebenarnya *inode numbering* hanya terdapat di model *file system* Unix. Windows *file system* (seperti VFAT, NTFS) tidak mengimplementasikan model penomoran *inode* secara unik, karena konsep *file system*-nya sendiri berbeda (pada model *file system* Unix, objek-objek yang bekerja adalah *superblock, inode*, dan *file*). Dan untuk implementasinya di *subsystem* Linux, VFS menggunakan *temporary inode numbering* untuk *file system* yang tidak menggunakan model *file system* umum di Unix. Untuk Windows *file system*, *xpack* tidak dapat memasukkannya dalam kategori *Hostfs*. Karena setiap kali *file system* tersebut di *mount*, nomor *inode* untuk tiap *file/directory* akan berbeda-beda *XPackfs* tidak dapat menyimpan nomor *inode* yang sifatnya *temporary*. Karena jika dipaksakan untuk menjadi *Hostfs*, setiap kali terjadi proses *mounting* akan menyebabkan atribut khusus *xpack* merujuk pada *file/directory* yang berbeda.

Semua parameter-parameter atribut tersimpan di satu *file database*. Untuk atribut *historical* disimpan di *file .historical.dat*. Gambar 3.11 menunjukkan hubungan antara *database inode* dengan *database* atribut. Parameter-parameter atribut di *historical file* akan dijelaskan di subbab 3.3 tentang *userspace utility*.

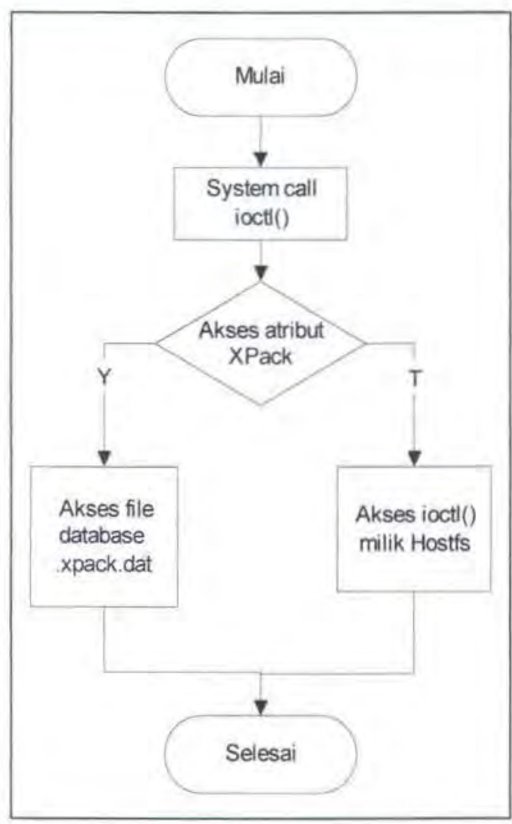


Gambar 3.11 Hubungan antara database inode dan database atribut

3.3 USERSPACE UTILITY

Atribut-atribut khusus milik *xpack* dapat diakses dari *userspace* dengan 2 macam aplikasi *utility*. Aplikasi pertama adalah *utility* untuk men-*set*, memperoleh, dan menghapus atribut *xpack*. Dan yang kedua adalah utility khusus untuk penanganan atribut fungsional tertentu yang diberikan *XPackfs*. Di *historical file*, utility tersebut digunakan untuk memanipulasi *version set* dari *current file*.

Semua mekanisme akses atribut *xpack* menggunakan *system call ioctl* yang telah di-*trap*. Jika parameter yang digunakan dalam pemanggilan *ioctl* merupakan pengaksesan atribut *xpack*, maka alur proses diarahkan ke akses *data base file .xpack.dat* yang menyimpan informasi atribut (lihat gambar 3.12). Jika bukan parameter untuk pengaksesan atribut, maka proses berjalan seperti biasanya, yaitu melanjutkannya untuk memanggil fungsi *ioctl* milik *host*.



Gambar 3.12 Alur proses system call ioctl

Parameter-parameter atribut *historical* merupakan *policy* dalam penyimpanan *file* versi. Adapun perincian dari parameter *historical* yang dapat diakses oleh *user* melalui *system call ioctl* adalah :

- Flag *historical*: digunakan untuk men-*set* status *historical file*, RETAIN_NAME, supaya jika terjadi perubahan nama *file*, nama *historical file* tetap ada. (lihat di gambar 3.3).
- Jumlah minimum dan maksimum *file* versi yang diijinkan.
- Nilai *trigger* jumlah penulisan yang menyebabkan terbentuknya *file* versi. Jika bernilai 0, maka aturan ini di tidak berlaku/*disabled*.
- Nilai *trigger* untuk interval waktu yang ditentukan menghasilkan *file* versi. Jika bernilai 0, maka aturan ini di tidak berlaku/*disabled*.
- Nilai minimum dan maksimum waktu yang diijinkan untuk *file* versi tetap berada di *version set*.
- Maksimum kapasitas *version set* yang diijinkan.
- Informasi status jumlah *file* versi saat ini. Sifatnya *read-only*.
- Informasi status kapasitas *version set* saat ini. Sifatnya *read-only*.
- Informasi tonggak waktu dari *current file* yang bisa menghasilkan *file* versi. Sifatnya *read-only*.

Agar user dapat melihat *file-file* versi dari *current file*, maka diperlukan *utility* untuk memberitahukan ke *XPackfs* bagaimana cara mengaksesnya. *Utility* tersebut berinteraksi dengannya juga melalui *system call ioctl* yang telah di-*trap* oleh *XPackfs*. *User* dapat memanggil *file-file* versi untuk dibaca oleh aplikasi umum *editor* dan juga dapat mengembalikannya lagi. *File-file* versi yang ditampilkan merupakan *virtual file* milik *Pseudofs* yang telah di bahas di subbab sebelumnya.

BAB IV

IMPLEMENTASI PERANGKAT LUNAK

4.1 LINGKUNGAN IMPLEMENTASI

Implementasi *file system* untuk tugas akhir ini menggunakan bahasa C, karena *kernel* Linux sendiri dibuat dengan bahasa C (ANSI) dengan optimasi pada level arsitektur menggunakan assembly. Program untuk kompilasi *kernel* adalah menggunakan GCC. Untuk arsitektur yang digunakan awalnya adalah menggunakan User Mode Linux (UML), dikarenakan kemudahannya untuk proses *debugging kernel*. UML saat ini cukup tepat untuk digunakan sebagai tempat *development kernel*, khususnya untuk implementasi kode-kode inti dari *kernel* (*core*). Kode-kode inti umumnya bersifat independen terhadap arsitektur-arsitektur yang didukung *kernel* Linux. Salah satunya adalah kode pada *subsystem* VFS dan *file system* yang didukungnya. Dan untuk proses *debugging*nya, menggunakan aplikasi GDB. Analisa *source code* dari *kernel*, khususnya yang dipelajari adalah dengan menggunakan *source navigator*. Program ini memudahkan menganalisa kode program yang kompleks, dan juga mampu memperlihatkan interkoneksi antara fungsi-fungsi (dalam bentuk grafik), serta lokasi *file* letak deklarasi dan implementasi variabel dan fungsi. Jadi benar-benar memudahkan dalam menganalisa kode *kernel* yang tercatat berisikan banyak *file*. Dan untuk utility pada tingkat level user juga di implementasikan dalam bahasa C dan berjalan di mode *console*.

4.2 IMPLEMENTASI DATA

Untuk implementasi data, dibagi menjadi dua bagian, yakni struktur data di *kernel* dan struktur data di *userspace utility*.

4.2.1 Struktur data kernel

Struktur data yang diimplementasikan di *kernel* dibagi menjadi 4 kategori, yakni struktur data *XPackfs* yang melekat ke objek-objek VFS milik *Hostfs*, struktur data *Pseudofs*, struktur data untuk akses *file database inode* baik di *disk* dan *memory*, dan yang terakhir adalah struktur data untuk akses *file database* atribut yang berisi parameter-parameter atribut, khususnya *historical file*.

4.2.1.1 Struktur data XPackfs

Struktur data *XPackfs* secara garis besarnya terbagi menjadi 2, struktur *xpack_info* dan struktur *xinode_info*. Struktur *xpack_info* pada dasarnya mirip dengan struktur *superblock*. Tiap satu *file system* mempunyai satu instan. Struktur ini dirujuk oleh struktur *xinode_info*.

```
struct xpack_info {
        struct dentry                        *x_dir;
        struct dentry                        *x_db;
        struct dentry                        *x_attrdb[MAX_ATTRIBUTE];
        xp_db_info                           x_db_info;
        struct semaphore                     x_sem;
        unsigned int                         x_state;
        struct vfsmount                      *x_pseudomnt;
        struct super_operations              x_sop;
        struct address_space_operations      x_aop;
        struct dentry_operations             x_dop;
        struct dentry_operations             *x_hdop;
        struct super_operations              *x_hsop;
        struct super_block                   *x_hsb;
};
```

Gambar 4.1 Struktur xpack_info

Field-field-nya berisi informasi tentang *pointer* ke lokasi *directory database* (*x_dir*), *pointer* ke *file database inode* (*x_db*), *pointer* ke *file-file database* atribut (*x_attrdb*) dengan nilai maksimum atribut yang boleh dimiliki satu *file* adalah 32 (MAX_ATTRIBUTE), *pointer* ke *native file system* yang berjalan (*x_pseudomnt*), kumpulan fungsi-fungsi VFS milik *xpack* (*x_sop, x_aop, x_dop*), *pointer* ke fungsi-fungsi VFS milik *host* yang tersubstitusi dengan milik *xpack* (*x_hsop, x_hdop*), serta *pointer* ke *superblock* milik *host* (*x_hsb*). Selengkapnya terlihat di gambar 4.1.

Struktur *xinode_info* digunakan untuk informasi spesifik *xpack* di tiap-tiap objek *inode host*. Struktur ini nantinya akan diletakkan di *field u.generic_ip* milik *inode host*.

```
struct xinode_info {
        struct xpack_info                  *i_xpi;
        struct inode                       *i_hinode;
        struct semaphore                   i_sem;
        struct rw_semaphore                i_mapsem;
        unsigned int                       i_attr;
        unsigned int                       i_dirty;
        unsigned int                       i_state;
        struct rw_semaphore                i_attrsem;
        xp_db_index                        i_dbidx;
        void                               *i_attrdata[MAX_ALLOWED_ATTR];
        struct address_space               *i_map;
        int                                i_nmap;
        unsigned long                      *i_buffer;
        int                                i_buflenbit;
        pgoff_t                            i_buffidx;
        struct inode_operations            *i_hiop;
        struct file_operations             *i_hfop;
        struct address_space_operations    *i_haop;
};
```

Gambar 4.2 Struktur xinode_info

Field-field-nya berisi informasi tentang *pointer* ke *inode host* (*i_hinode*), atribut-atribut yang dimiliki *file host* (*i_attr*), *pointer* ke parameter-parameter

spesifik dari masing-masing atribut yang dimiliki (*i_attrdata*), serta *pointer* ke fungsi-fungsi VFS milik *host* yang tersubstitusi oleh *xpack* (*i_hiop, i_hfop, i_haop*). Untuk lebih lengkapnya dapat dilihat di gambar 4.2.

4.2.1.2 Struktur data Pseudofs

Kegunaan struktur data *Pseudofs* hampir sama dengan struktur data *XPackfs*, hanya berbeda *field-field*-nya saja dan berdiri sendiri sebagai *file system* yang tidak melekat ke *Hostfs*. Struktur *pseudo_info* berisi *field pointer* ke *superblock xpack*/ *xpack_info* (*p_xpi*) (lihat gambar 4.3). Struktur *pinode_info* berisi informasi spesifik dari objek *inode* di *subsystem* VFS. *Field*-nya terdiri dari: atribut yang berjalan pada *pseudo file* (*i_attr*), *pointer* ke data spesifik milik *xpack* yang menggunakan *pseudo file* (*i_data* dan *i_private*), status dari *inode* (*i_state*), objek *inode* yang dipakai oleh VFS (*vinode*), serta fungsi khusus yang akan dipanggil saat *pseudo file* di hapus (*i_unlink*) (lihat gambar 4.4).

```
struct pseudo_info {
        struct xpack_info           *p_xpi;
};
```

Gambar 4.3 Struktur pseudo_info

```
struct pinode_info {
        unsigned int          i_attr;
        void                  *i_data;
        unsigned int          i_state;
        struct inode          vinode;
        struct list_head      i_list;
        unsigned long         i_private;
        void                  (*i_unlink) (struct pinode_info*);
};
```

Gambar 4.4 Struktur pinode_info

4.2.1.3 Struktur data database inode

Informasi *inode-inode* yang memiliki atribut *xpack* disimpan dengan format sebagai berikut (gambar 4.5) : nomor unik *inode* milik *host* (*inode_num*), *flag* dari atribut; misal *historical file* (*attr_flag*), array indeks untuk id tiap atribut (*attr_idx*), dan *array offset* ke lokasi *file database* atribut (*db_offset*).

```
typedef struct {
        __u32   inode_num;
        __u32   attr_flag;
        __u8    attr_idx[MAX_ALLOWED_ATTR];
        __u32   db_offset[MAX_ALLOWED_ATTR];
}xp_database_d;
```

Gambar 4.5 Struktur xp_database_d

Struktur *xp_database_d* merupakan format untuk penyimpanan fisik di *disk*, sedangkan struktur data untuk menampungnya di *memory* adalah *xp_db_index*. Detil atributnya meliputi status atribut xpack mengalami perubahan (*dirty*), lokasi *file database inode* untuk memudahkan pencarian *record* (*offset*), serta *field-field* yang fungsinya sama dengan *field-field* yang ada di *xp_database_d* (*attr_idx*, dan *db_offset*) (gambar 4.6). Struktur ini direferensi oleh *xinode_info*.

```
typedef struct {
        unsigned int     dirty;
        unsigned int     offset;
        unsigned char    attr_idx[MAX_ALLOWED_ATTR];
        unsigned int     db_offset[MAX_ALLOWED_ATTR];
}xp_db_index;
```

Gambar 4.5 Struktur xp_db_index

Dalam *record database inode*, dialokasikan ruang untuk beberapa byte untuk menampung informasi *file-file database*. Alokasi ruang terletak diawal *file* dan direferensikan sebagai struktur *xp_db_info_d*. *Field*-nya meliputi *counter*

untuk men-*generate* nama *file* yang di*reserve* oleh *xpack* (*reserved_cnt*), ukuran dari *file database inode*, serta ukuran dari masing-masing *file database* atribut. Deklarasi struktur tersebut terlihat pada gambar 4.6.

```
typedef struct {
        __u32 reserved_cnt;
        __u32 xpdb_size;
        __u32 attrdb_size[MAX_ATTRIBUTE];
}xp_db_info_d;
```

Gambar 4.6 Struktur xp_db_info_d

4.2.1.4 Struktur data database atribut historical

Struktur *database* atribut *historical* dibagi menjadi 2 bagian, yang satu untuk mengakses data di *file*, dan yang satunya untuk mengakses data di *memory*.

```
typedef struct {
        __u16   rec_len;
        __u8    hist_flag;
        __u8    max_write_cnt;
        __u8    cur_write_cnt;
        __u8    max_hist_cnt;
        __u8    min_hist_cnt;
        __u8    cur_hist_idx;
        __u32   max_hist_size;
        __u32   cur_hist_size;
        __u32   check_time;
        __u32   interval_time;
        __u32   min_date;
        __u32   max_date;
        __u32   hist_file[0];
}xp_historical_d;
```

Gambar 4.7 Struktur xp_historical_d

Format dari data yang tersimpan di *file* adalah panjang *record* (*rec_len*), *flag* untuk *historical* (*hist_flag*), nilai maksimum *file* versi (*max_hist_cnt*), nilai minimum *file* versi (*min_hist_cnt*), jumlah penulisan untuk men-*trigger* versi

(*max_write_cnt*), nilai *counter* penulisan yang sudah terjadi (*cur_write_cnt*), indeks *file* versi saat ini (*cur_hist_idx*), tonggak waktu untuk pen-*trigger*an versi (*check_time*), waktu interval pen-*trigger*an versi (*interval_time*), tanggal minimum *file* versi yang disimpan (*min_date*), tanggal maksimum *file* versi yang tersimpan (*max_date*), kapasitas maksimum dari *version set* (*max_hist_size*), kapasitas *version set* saat ini (*cur_hist_size*), dan *array* yang berisi informasi nama-nama *file* versi dalam bentuk angka. Deklarasinya terlihat di gambar 4.7.

Struktur *xp_historical* digunakan menampung parameter-parameter *historical* di *memory*. *Field-field*-nya sama dengan struktur *xp_historical_d*, hanya saja *xp_historical* tidak menyimpan semua informasi *file* versi di *memory*, untuk mengaksesnya memang harus membaca lewat *file database*. Pertimbangannya adalah untuk tidak terlalu menkonsumsi *memory*. Struktur ini juga memuat *list* dari *file* versi yang dijadikan sebagai *virtual file* (*Pseudofs*). Deklarasi *xp_historical* dapat dilihat di gambar 4.8.

```
typedef struct {
        unsigned char          hist_flag;
        unsigned char          max_hist_cnt;
        unsigned char          min_hist_cnt;
        unsigned char          max_write_cnt;
        unsigned char          cur_write_cnt;
        unsigned char          cur_hist_idx;
        unsigned long          check_time;
        unsigned long          interval_time;
        unsigned long          min_date;
        unsigned long          max_date;
        unsigned int           max_hist_size;
        unsigned int           cur_hist_size;
        struct list_head       ps_hist;
}xp_historical;
```

Gambar 4.8 Struktur xp_historical

4.2.2 Struktur data userspace utility

Struktur data yang berada di aplikasi *utility historical* mempunyai kemiripan dengan struktur data di level *kernel*. Struktur ini merupakan media komunikasi antara mode *user* dan mode *kernel* melalui *system call ioctl*. Struktur yang di *passing* ke mode *kernel* adalah struktur *xp_attribute_u*, yang *field-field*-nya adalah meliputi tipe dari atribut yang akan di akses (misalnya *historical*) (*type*), serta parameter-parameter atribut dari tipe tersebut (*union attr*). Field *union attr* yang dideklarasikan untuk saat ini, yang baru diimplementasikan di *XPackfs* hanyalah atribut *historical*. Untuk detil deklarasinya ada di gambar 4.9.

```
typedef struct {
        unsigned int type;
        union {
                xp_historical_u hist;
        }attr;
}xp_attribute_u;
```

Gambar 4.9 Struktur xp_attribute_u

Parameter-parameter atribut dari *historical* sendiri yang masuk dalam *field union attr* di *xp_attribute_u*, isi-isinya juga sama dengan struktur *xp_historical* di *kernel*. Hanya saja yang tidak dimasukkan dalam struktur *xp_historical_u* ini adalah *field cur_hist_idx* milik *xp_historical*, yang merupakan *field* internal yang digunakan untuk *kernel* sendiri, tidak perlu di ekspos ke level *user*. Untuk *field* lainnya masih dapat di ekspos ke level *user*, hanya saja untuk beberapa *field* tertentu seperti *cur_hist_cnt*, *cur_hist_size*, *check_time* tidak dapat di ubah/*set* nilainya, karena sifatnya hanya *read-only* (untuk informasi pemberitahuan kepada *user*). Gambar 4.10 menunjukkan deklarasi dari struktur *xp_historical_u*.

```
typedef struct {
        unsigned char    hist_flag;
        unsigned char    max_hist_cnt;
        unsigned char    min_hist_cnt;
        unsigned char    cur_hist_cnt;
        unsigned char    max_write_cnt;
        unsigned int     max_hist_size;
        unsigned int     cur_hist_size;
        time_t           check_time;
        time_t           interval_time;
        time_t           min_date;
        time_t           max_date;
}xp_historical_u;
```

Gambar 4.10 Struktur xp_historical_u

4.3 IMPLEMENTASI PROSES

4.3.1 Inisialisasi XPack di Hostfs

Saat *file system* di *mount* dengan tipe *XPackfs*, *subsystem* VFS akan memanggil fungsi *xp_get_sb* yang terdefinisi di struktur *file_system_type*. Di fungsi ini, *xpack* memanggil *native file system* yang terdapat di *disk* dan juga *Pseudofs*. Proses inisialisasi dapat dilihat di gambar 4.11. Jika tipe *native file system* tidak disebutkan saat proses *mounting* (opsinya tidak disebutkan), maka *xpack* akan memanggil fungsi *get_sb* dari masing-masing *file system* yang terdaftar di *kernel* sampai ditemukan yang sesuai. Sesudah berhasil menjalankan *mounting* ke *native file system*, *xpack* akan mencari *directory* tempat penyimpanan *database* milik *xpack* (*.xpack*). Jika tidak ditemukan, maka *xpack* tidak akan melanjutkan inisialisasinya, tetapi *native file system* tetap akan berjalan tanpa intervensi dari *xpack*. Jika berhasil menemukan *directory .xpack*, maka akan dilanjutkan dengan membaca *file database inode*, dan apabila tidak ada, akan dibuat sendiri oleh *xpack*.

```
printk(KERN_INFO "Filesystem Expansion Pack Initializing...\n");
/* search for compatible host fs */
if (data && strlen(data)) {
        subfs_param = strchr(data,',');
        if (subfs_param)
                *subfs_param++ = '\0';
        while (fstypes_support[i]) {
                if (!strcmp(data, fstypes_support[i]))
                        break;
                else
                        i++;
        }
        if (!fstypes_support[i]) {
                printk(KERN_ERR "XPack: Unsupported filesystem type.\n");
                goto out;
        }
        /* check kernel support */
        fstype = get_fs_type(fstypes_support[i]);
        if (!fstype) {
                printk(KERN_ERR "XPack: Filesystem not supported/loaded by kernel.\n");
                        goto out;
        }
        hsb = fstype->get_sb(fstype, flags, dev_name, subfs_param);
}else {
        /* for unknown type, iterate all fs support (happens on boot) */
        while (fstypes_support[i]) {
                fstype = get_fs_type(fstypes_support[i]);
                if (!fstype) {
                        i++;
                        continue;
                }
                hsb = fstype->get_sb(fstype, flags, dev_name, data);
                if (!IS_ERR(hsb))
                        break;
                i++;
        }
}
/* initiate xpack & pseudo fs */
if (!IS_ERR(hsb))
        err = xp_init(hsb, xfstype, flags);
put_filesystem(fstype);
/* if failed, leave it to host fs, xpack died */
if (err)
out:
        printk(KERN_INFO "Filesystem Expansion Pack not started...\n");
return hsb;
```

Gambar 4.11 Potongan kode inisialisasi awal XPackfs

Setelah semua berjalan lancar, selanjutnya adalah melakukan intervensi ke objek-objek VFS yang dimilik oleh *Hostfs*. Objek awal untuk di intervensi adalah

superblock, *field* untuk operasi *superblock* diganti dengan milik *xpack*. Deklarasi strukturnya terlihat di gambar 4.12.

```
struct super_operations xpack_sops = {
        .alloc_inode            = xp_alloc_inode,
        .destroy_inode          = xp_destroy_inode,
        .read_inode             = xp_read_inode,
        .dirty_inode            = xp_dirty_inode,
        .write_inode            = xp_write_inode,
        .put_inode              = xp_put_inode,
        .drop_inode             = xp_drop_inode,
        .delete_inode           = xp_delete_inode,
        .put_super              = xp_put_super,
        .write_super            = xp_write_super,
        .sync_fs                = xp_sync_fs,
        .write_super_lockfs     = xp_write_super_lockfs,
        .unlockfs               = xp_unlockfs,
        .statfs                 = xp_statfs,
        .remount_fs             = xp_remount_fs,
        .clear_inode            = xp_clear_inode,
        .umount_begin           = xp_umount_begin,
        .show_options           = xp_show_options,
};
```

Gambar 4.12 Struktur operasi superblock

Objek selanjutnya yang mengalami substitusi operasi adalah *dentry* dan *inode* milik *root* dari *mounted tree*. Field operasi *dentry* milik *Hostfs* diganti dengan operasi *xpack* yang deklarasinya terlihat di gambar 4.13.

```
struct dentry_operations xpack_dops = {
        .d_revalidate   = xp_d_revalidate,
        .d_hash         = xp_d_hash,
        .d_compare      = xp_d_compare,
        .d_delete       = xp_d_delete,
        .d_release      = xp_d_release,
        .d_iput         = xp_d_iput,
};
```

Gambar 4.13 Struktur operasi dentry

Root directory ini merupakan lokasi awal untuk menyebarkan fungsi-fungsi *xpack* ke objek lainnya. *Root directory* merupakan tempat awal untuk me-*resolve*

file-file dan *directory-directory* lainnya. Fungsi VFS yang bertanggung jawab untuk melakukan *resolve file* atau *directory* dibawahnya adalah *lookup*. Fungsi ini hanya terdapat di objek *inode* tipe *directory*. Deklarasi lengkapnya dari masing-masing fungsi *inode* dengan tipe *regular file*, *directory*, dan *symbolic link* dapat di lihat di gambar 4.14.

```
struct inode_operations xpack_dir_iops = {
        .create         = xp_create,
        .lookup         = xp_lookup,
        .link           = xp_link,
        .unlink         = xp_unlink,
        .symlink        = xp_symlink,
        .mkdir          = xp_mkdir,
        .rmdir          = xp_rmdir,
        .mknod          = xp_mknod,
        .rename         = xp_rename,
        .permission     = xp_permission,
        .setattr        = xp_setattr,
        .getattr        = xp_getattr,
        .setxattr       = xp_setxattr,
        .getxattr       = xp_getxattr,
        .listxattr      = xp_listxattr,
        .removexattr    = xp_removexattr,
};

struct inode_operations xpack_reg_iops = {
        .truncate       = xp_truncate,
        .permission     = xp_permission,
        .setattr        = xp_setattr,
        .getattr        = xp_getattr,
        .setxattr       = xp_setxattr,
        .getxattr       = xp_getxattr,
        .listxattr      = xp_listxattr,
        .removexattr    = xp_removexattr,
};

struct inode_operations xpack_sym_iops = {
        .put_link       = xp_put_link,
        .readlink       = xp_readlink,
        .follow_link    = xp_follow_link,
        .permission     = xp_permission,
        .setxattr       = xp_setxattr,
        .getxattr       = xp_getxattr,
        .listxattr      = xp_listxattr,
        .removexattr    = xp_removexattr,
};
```

Gambar 4.14 Struktur operasi inode

Di operasi *lookup* milik *inode* tipe *directory* ini, semua substitusi dari fungsi-fungsi objek *dentry, inode*, dan *file* terjadi. Selain melakukan substitusi fungsi, juga menyembunyikan letak *directory .xpack* dari akses *user*, yang gunanya mencegah manipulasi *file-file* penting yang disimpan oleh *xpack*. Jika *inode* mempunyai atribut khusus milik *xpack* (dengan melihat di *directory inode*), maka atribut tersebut di letakkan di objek *inode*. Penjelasan dari proses *inode lookup* ada di gambar 4.15.

```
if (hidir == hidir->i_sb->s_root->d_inode) {
        /* hide xpack entry files in s_root */
        if(!strncmp(hd->d_name.name, XPACK_DIRECTORY, strlen(XPACK_DIRECTORY)))
                return ERR_PTR(-EACCES);
}
hdres = HIOP(xiidir, lookup, hidir, hd, nd);
if (IS_ERR(hdres) || !hd->d_inode)
        return hdres;
if (!xp_hook_inode(hd->d_inode)) {
        xp_hook_dentry(hd);
        xp_lookup_db(hd->d_inode);
}
```

Gambar 4.15 Potongan kode inode lookup

Penyembunyian *directory .xpack* juga dilakukan di fungsi *readdir* milik objek *file* tipe *directory*. Fungsi ini berguna untuk menampilkan *list regular file* dan *directory* yang berada di bawahnya. Proses penyembunyiannya dapat dilihat di gambar 4.16.

```
if (hf->f_dentry == hf->f_dentry->d_sb->s_root)
        if(!strncmp(name, XPACK_DIRECTORY, strlen(XPACK_DIRECTORY)))
                return over;

over = fill_fn(buf, name, namlen, offset, ino, d_type);
```

Gambar 4.16 Potongan kode directory readdir

Fungsi-fungsi yang berhubungan dengan objek *file* baik itu tipe *regular file* maupun *directory* yang di miliki oleh *xpack* disubstitusikan dengan fungsi-fungsi di *Hostfs;* dapat terlihat di gambar 4.17. File tipe *symbolic link* tidak mempunyai operasi *file*, karena tidak mempunyai isi data seperti yang ada di *regular file* ataupun *directory*.

```
struct file_operations xpack_dir_fops = {
        .llseek         = xp_llseek,
        .read           = xp_read,
        .readdir        = xp_readdir,
        .ioctl          = xp_ioctl,
        .open           = xp_open,
        .flush          = xp_flush,
        .release        = xp_release,
        .fsync          = xp_fsync,
        .fasync         = xp_fasync,
};

struct file_operations xpack_reg_fops = {
        .llseek                 = xp_llseek,
        .read                   = xp_read,
        .write                  = xp_write,
        .poll                   = xp_poll,
        .ioctl                  = xp_ioctl,
        .mmap                   = xp_mmap,
        .open                   = xp_open,
        .flush                  = xp_flush,
        .release                = xp_release,
        .fsync                  = xp_fsync,
        .fasync                 = xp_fasync,
        .lock                   = xp_lock,
        .aio_read               = xp_aio_read,
        .aio_write              = xp_aio_write,
        .aio_fsync              = xp_aio_fsync,
        .readv                  = xp_readv,
        .writev                 = xp_writev,
        .sendfile               = xp_sendfile,
        .sendpage               = xp_sendpage,
        .get_unmapped_area      = xp_get_unmapped_area,
        .dir_notify             = xp_dir_notify,
};
```

Gambar 4.17 Struktur operasi file

Teknik injeksi fungsi VFS ini berperan besar dalam mengubah alur proses dari *Hostfs*, sehingga memudahkan untuk mengimplementasikan fungsional baru pada *native file system*.

4.3.2 Fungsi historical file

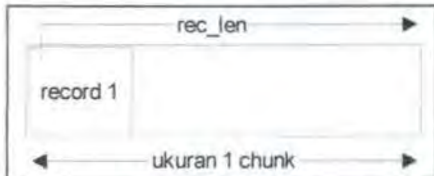
Fungsi yang terdapat di *historical file* secara garis besarnya terbagi 2, yakni proses akses ke *file database* atribut dan proses pembuatan versi dari *current file*.

4.3.2.1 Akses database atribut

Berbeda dengan *database inode* yang *record*-nya berukuran tetap, *database* atribut yang dimiliki *historical* ukurannya bervariasi. Variasi ukuran terjadi dikarenakan jumlah maksimum *file* versi yang ditampung. Maka metode untuk mengaksesnya baik itu menambah atau menghapus *record* di *file database* tidak sesederhana apa yang ada di *database inode*. Semua akses data yang dilakukan oleh *xpack* menggunakan satuan *chunk*. *Chunk* disini berukuran 4096 *byte*, yang mana dari semua arsitektur yang di dukung oleh Linux merupakan ukuran *page* terkecil di *memory*.

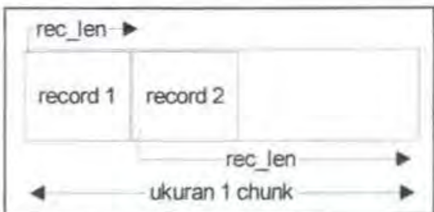
Untuk memudahkan manajemen penyimpanan dan aksesnya, *xpack* selalu menggunakan ukuran 1 *chunk*. Untuk penambahan *record* di *database*, *xpack* akan membaca 1 *chunk* dari *file* dan mencari dalam area tersebut yang masih kosong. Disini *field rec_len* berguna untuk mencari area yang masih kosong. Jika pada awalnya *chunk* masih dalam keadaan kosong sepenuhnya dan terdapat penambahan *record* untuk pertama kalinya, maka isi *rec_len* akan mencakup ukuran satu *chunk* tersebut. Ukuran sebenarnya dari *record* tersebut dapat diketahui hanya dengan melihat panjang *field* dari *xp_historical_d* ditambah

dengan melihat nilai maksimum jumlah versi *file*. Gambar 4.18 menjelaskan awal penambahan *record* di area *chunk* yang masih kosong.



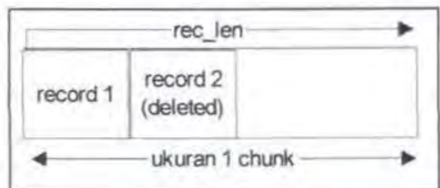
Gambar 4.17 Penambahan satu record pertama

Jika terdapat penambahan *record* lagi, maka *record* yang kedua akan mengambil ruang milik *record* 1, dan meng-*update field rec_len* milik *record* 1. Gambarannya dapat di lihat di gambar 4.18.



Gambar 4.18 Penambahan record kedua

Untuk penghapusan *record*, yang diperlukan hanyalah meng-*update* nilai *rec_len* milik *record* sebelumnya. Nilai *rec_len record* yang dihapus ditambahkan ke *rec_len record* sebelum dirinya. Gambar 4.19 menjelaskan lebih lanjut proses penghapusan *record* (disini *record* 2 yang dihapus).



Gambar 4.19 Penghapusan record kedua

Disini terdapat perkecualian, apabila yang dihapus adalah *record* pertama dari *chunk*, maka *record* akan di tandai sebagai *unused record* di *field hist_flag*. Gambaran jelasnya terlihat pada gambar 4.19.

Gambar 4.20 Penghapusan record pertama

Apabila dalam satu *chunk* terjadi penghapusan *record* yang terakhir, maka akan dilakukan pengecekan apakah *chunk* tersebut berada di posisi akhir *file*. Jika ya, maka *chunk* tersebut dihapus dari *file*, dengan cara melakukan *truncate* sebesar ukuran *chunk*.

4.3.2.2 Fungsi pembentukan historical file

System call ioctl merupakan media komunikasi bagi *user* untuk melakukan *setting* atribut *historical* di *file* milik *Hostfs*. Disini *xpack* melakukan *trap* di fungsi *ioctl* milik *Hostfs* dengan memantau parameter yang di*passing*. *Setting* yang dapat dikerjakan oleh *user* yakni men*set* atribut *historical*, menghapus atribut *historical*, memperoleh informasi parameter-parameter atribut *historical*, dan men-*set* parameter-parameter atribut *historical*. Penjelasan proses lebih detil tentang *setting* atribut *historical* dapat dilihat di gambar 4.21, yang merupakan potongan kode milik *xpack* untuk men-*trap* parameter-parameter di fungsi *ioctl* yang berhubungan dengan setting atribut *historical*.

```
case XP_IOC_SETATTR:
case XP_IOC_CLRFLAGS:
        if (IS_RDONLY(hi))
                return -EROFS;
        if ((current->fsuid != hi->i_uid) && !capable(CAP_FOWNER))
                return -EACCES;
        if (cmd == XP_IOC_CLRFLAGS) {
                if (get_user(type, (int __user *) arg))
                        return -EFAULT;
                return xp_clear_attr(hi, type);
        } else {
                if (copy_from_user(&attr_u, (char*)arg, sizeof(attr_u)))
                        return -EFAULT;
                return xp_set_attr(hi, &attr_u);
        }
case XP_IOC_GETATTR:
        if (copy_from_user(&attr_u, (char*)arg, sizeof(attr_u)))
                return -EFAULT;
        res = xp_get_attr(hi, &attr_u);
        if (res)
                return res;
        if (copy_to_user((char*)arg, &attr_u, sizeof(attr_u)))
                return -EFAULT;
        return 0;
case XP_IOC_GETFLAGS:
        return put_user(xii->i_attr, (int __user *) arg);
```

Gambar 4.21 Potongan kode ioctl

4.3.2.3 Fungsi pembuatan versi

Pembentukan versi *file* terjadi hanya terdapat saat terjadi pemanggilan fungsi VFS dari objek *file* meliputi *write, aio_write, writev*. Selain itu juga dapat ditemukan pada fungsi VFS dari objek *inode*, yakni di operasi *rename*, dan *set_attr* (disini awal proses *truncate* dilakukan). Fungsi-fungsi tersebut di-*trap* oleh *xpack* dengan menambahkan fungsi pengecekan *policy historical* dari *file*, yang dapat menentukan pembentukan versi *file*. Ada dua macam *trigger* yang sudah didefinisikan sebelumnya, yakni *write trigger* dan *time trigger*. Untuk penjelasan *write trigger* dapat dilihat di gambar 4.21 yang merupakan potongan kodenya. Untuk penjelasan *time trigger*, dapat dilihat potongan kodenya di gambar 4.22.

```
/* already make trigger */
if (hf && hf->private_data)
        return 0;
if (!xii->i_hinode->i_size)
        return 0;
histidx = ATTR_IDX(XPA_HISTORICAL, &xii->i_dbidx);
hs = (xp_historical*)xii->i_attrdata[histidx];
/* no write retention */
if (!hs->max_write_cnt)
        return 0;
hs->cur_write_cnt++;
if (hs->cur_write_cnt < hs->max_write_cnt)
        return 0;

hs->cur_write_cnt = 0;
/* extend check time */
if (hs->interval_time && hs->check_time < now.tv_sec) {
        tm = ((now.tv_sec - hs->check_time + hs->interval_time - 1)
                        / hs->interval_time) * hs->interval_time;
        hs->check_time += tm;
}
xii->i_dirty |= 1UL << XPA_HISTORICAL;
if (hs_get_file(xii, histidx))
        return 0;
if (hf)
        hf->private_data = (void*)1;
```

Gambar 4.22 Potongan kode write trigger

```
if (!xii->i_hinode->i_size)
        return 0;
histidx = ATTR_IDX(XPA_HISTORICAL, &xii->i_dbidx);
hs = (xp_historical*)xii->i_attrdata[histidx];
/* no time retention */
if (!hs->interval_time)
        return 0;
if (hs->check_time >= now.tv_sec)
        return 0;
/* extend check time */
tm = ((now.tv_sec - hs->check_time + hs->interval_time - 1)
                / hs->interval_time) * hs->interval_time;
hs->check_time += tm;
hs->cur_write_cnt = 0;
xii->i_dirty |= 1UL << XPA_HISTORICAL;
if (hs_get_file(xii, histidx))
        return 0;
```

Gambar 4.23 Potongan kode time trigger

4.3.3 Fungsi manipulasi file versi

Manipulasi *file* versi dapat dilakukan oleh *user*, juga melalui *system call ioctl* seperti saat men-*setting* atribut *historical*. Disini *user* dapat memanggil *file-file* versi dari *current file*, mengembalikannya lagi seperti semula, men-*set* isi *file* versi sebagai isi *current file*, serta menghapus *file* versi dari *version set*. Pemanggilan sebenarnya adalah dengan membuat *virtual file* milik *Pseudofs* yang merujuk ke *file* versi yang tersimpan di *directory .xpack* tersebut. Ini disebabkan karena *directory .xpack* telah disembunyikan oleh *xpack*, sehingga *user* tidak dapat memanipulasi *file-file* penting milik *xpack*. Potongan kode di fungsi *ioctl* yang telah di-*trap* oleh *xpack* dapat dilihat di gambar 4.24.

```
/* historical file manipulation */
case XP_IOC_DELHISTFILE:
        histtype++;
case XP_IOC_SETHISTFILE:
        histtype++;
        if (IS_RDONLY(hi))
                return -EROFS;
case XP_IOC_PUTHISTFILE:
        histtype++;
case XP_IOC_GETHISTFILE:
        if ((current->fsuid != hi->i_uid) && !capable(CAP_FOWNER))
                return -EACCES;
        if (get_user(hfidx, (int __user *)arg))
                return -EFAULT;
        down_read(&xii->i_attrsem);
        res = hs_ctl_file(hf, histtype, hfidx);
        up_read(&xii->i_attrsem);
        return res;
```

Gambar 4.24 Potongan kode ioctl untuk manipulasi file versi

BAB V

UJI COBA DAN EVALUASI

5.1 LINGKUNGAN UJI COBA

Uji coba dan evaluasi dilakukan setelah perancangan dan pembuatan perangkat lunak selesai dilakukan. Hasil uji coba tersebut digunakan untuk evaluasi dan perbaikan *system* sesuai dengan batasan-batasan yang telah ditentukan sebelumnya.

Pengujian ini dapat dilakukan pada lingkungan *system* operasi berbasis Linux. Adapun yang perlu disiapkan sebelum pengujian adalah sebuah komputer yang dilengkapi dengan atribut-atribut sebagai berikut : *System* dengan distribusi Debian GNU/Linux versi 3.1, menggunakan *kernel* versi 2.6.8, dan *file system* yang digunakan adalah Reiserfs. *System* berjalan di spesifikasi *hardware* prosesor Intel Pentium II, dengan *memory* 64 Megabytes, dan *hard disk* dengan kapasitas 4 Gigabytes.

5.2 PELAKSANAAN UJI COBA

Uji coba dibagi menjadi 3 bagian (skenario). Yang **pertama** adalah *setting* awal untuk dapat menjalankan *XPackfs* di *file system* Reiserfs serta mendemonstrasikan secara sederhana penggunaan *historical file*. Untuk yang **kedua** adalah mencoba mengaplikasikan *historical file* ke salah satu *file* konfigurasi yang berada di */etc*. Nantinya *file* akan dikembalikan ke isi semula setelah *booting*, meski telah dilakukan beberapa perubahan/penulisan. Dan bagian

ketiga adalah mencoba melihat perubahan yang terjadi di masing-masing versi yang terbuat, dengan menggunakan program *diff*. Untuk yang **terakhir** dilakukan percobaan ke beberapa aplikasi editor untuk mengetahui keberhasilan dalam membuat versi file.

5.2.1 Uji coba I

Untuk seting awalnya, supaya *XPackfs* dapat berjalan di *native file system* Reiserfs, adalah :

1. Membuat *directory .xpack* di *root directory file system.*
2. *Reboot system* dengan *kernel* yang sudah terkompilasi statis *XPackfs*.
3. Memantau *boot message*, jika *XPackfs* berhasil berjalan, maka akan tampil seperti gambar 5.1.

```
TCP: Hash tables configured (established 4096 bind 8192)
NET: Registered protocol family 1
NET: Registered protocol family 17
NET: Registered protocol family 15
BIOS EDD facility v0.16 2004-Jun-25, 1 devices found
Filesystem Expansion Pack Initializing...
ReiserFS: hda1: found reiserfs format "3.6" with standard journal
ReiserFS: hda1: using ordered data mode
ReiserFS: hda1: journal params: device hda1, size 8192, journal first block 18,
max trans len 1024, max batch 900, max commit age 30, max trans age 30
ReiserFS: hda1: checking transaction log (hda1)
ReiserFS: hda1: Using r5 hash to sort names
Filesystem Expansion Pack started...
VFS: Mounted root (reiserfs filesystem) readonly.
Freeing unused kernel memory: 168k freed
Adding 208836k swap on /dev/hda3.  Priority:-1 extents:1
ReiserFS: hda2: found reiserfs format "3.6" with standard journal
ReiserFS: hda2: using ordered data mode
ReiserFS: hda2: journal params: device hda2, size 8192, journal first block 18,
max trans len 1024, max batch 900, max commit age 30, max trans age 30
ReiserFS: hda2: checking transaction log (hda2)
ReiserFS: hda2: Using r5 hash to sort names
eth0: link up, 100Mbps, full-duplex, lpa 0x45E1
```

Gambar 5.1 XPackfs berjalan diatas Reiserfs

Setelah berhasil menjalankan *xpack* di *Hostfs*, dilanjutkan dengan percobaan melakukan *setting* atribut *historical* ke *file* dengan tipe teks. Untuk itu diambil contoh *file /etc/modules*. Gambar 5.2 menunjukkan isi *file modules* sebelum

mengalami perubahan, serta pen-*setting*-an atribut *historical* ke *file modules*, dengan parameter *default* yang dimiliki utility *xpattr*.

```
nakula:/tmp# cat modules
# /etc/modules: kernel modules to load at boot time.
#
# This file should contain the names of kernel modules that are
# to be loaded at boot time, one per line.  Comments begin with
# a "#", and everything on the line after them are ignored.

nakula:/tmp# xpattr -s hist modules
historical flag         : Retained_Name
maximum files           : 10
minimum files           : 0
current files           : 0
maximum write trigger   : 1
maximum size of files   : 0 KB
current size of files   : 0 KB
checkpoint time         : -
interval time trigger   : 0 mins (0 hours)
maximum saved date      : -
minimum saved date      : -
nakula:/tmp#
```

Gambar 5.2 File modules di set atribut historical

Langkah selanjutnya dilakukan tiga kali perubahan terhadap isi *file modules*. Gambar 5.3 menunjukkan kondisi akhir isi *file modules* dan juga kondisi parameter-parameter *historical* dari *file modules* saat itu. Terlihat bahwa jumlah versi *file* milik *modules* sekarang adalah tiga, dikarenakan jumlah penulisan untuk men-*trigger* terjadinya versi *file* adalah 1 kali. Sehingga setiap kali terdapat perubahan/penulisan, maka akan menghasilkan *file* versi baru. Selama aplikasi memakai *file* deskriptor yang sama, maka setiap penulisan akan dianggap sebagai satu penulisan. Pemakaian sebuah *file* deskriptor dianggap sebagai satu *session.*

Untuk melihat *file-file* versi yang terbentuk, maka perlu dipanggil dengan menggunakan *utility hsfile*. Kemudian dipanggil *file-file* versi yang disimpan oleh *xpack* untuk dapat terlihat di *directory* tempat *file modules* berada. Gambar 5.4 memperlihatkan cara pemanggilan *file-file* versi milik *modules*.

```
nakula:/tmp# cat modules
# /etc/modules: kernel modules to load at boot time.
#
# This file should contain the names of kernel modules that are
# to be loaded at boot time, one per line.  Comments begin with
# a "#", and everything on the line after them are ignored.

snd-intel8x0
8139too
lp
usbcore
nakula:/tmp# xpattr -g hist modules
historical flag        : Retained_Name
maximum files          : 10
minimum files          : 0
current files          : 3
maximum write trigger  : 1
maximum size of files  : 0 KB
current size of files  : 1 KB
checkpoint time        : -
interval time trigger  : 0 mins (0 hours)
maximum saved date     : -
minimum saved date     : -
```

Gambar 5.3 File modules setelah 3 kali mengalami perubahan

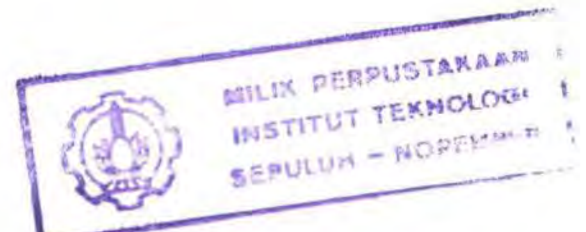
```
nakula:/tmp# ls
          modules
nakula:/tmp# hsfile -g 1 modules
nakula:/tmp# hsfile -g 2 modules
nakula:/tmp# hsfile -g 3 modules
nakula:/tmp# ls
          modules  modules.1  modules.2  modules.3
nakula:/tmp#
```

Gambar 5.4 Memanggil file-file versi milik modules

Untuk mengembalikan lagi *file-file* versi *modules*, *utility hsfile* dipanggil dengan perintah *put* beserta nomor indeks *file* dari *file modules*. Untuk jelasnya dapat terlihat di gambar 5.5.

```
nakula:/tmp# ls
          modules  modules.1  modules.2  modules.3
nakula:/tmp# hsfile -p 1 modules
nakula:/tmp# ls
          modules  modules.2  modules.3
nakula:/tmp# hsfile -p 2 modules
nakula:/tmp# ls
          modules  modules.3
nakula:/tmp# hsfile -p 3 modules
nakula:/tmp# ls
          modules
nakula:/tmp#
```

Gambar 5.5 Mengembalikan file-file versi milik modules


MILIK PERPUSTAKAAN
INSTITUT TEKNOLOGI
SEPULUH - NOPEMBER

Uji coba yang dilakukan pada tahap pertama berhasil menghasilkan *file* versi ketika terjadi perubahan *file*, dan mampu menampilkan *file-file* versi yang terbentuk untuk dapat diakses oleh aplikasi normal.

5.2.2 Uji coba II

Untuk uji coba kedua, dilakukan *setting* atribut *historical* ke *file* konfigurasi */etc/samba/smb.conf*. Parameter *historical* di-*set* dengan jumlah minimum *file* versi 1 dan maksimum *file* versi 1. Dengan begitu, *file* versi hanya menyimpan perubahan *file* yang pertama, untuk selanjutnya tidak ada pembuatan *file* versi, karena sudah dianggap melanggar *policy* minimum *file*. Gambar 5.6 menjelaskan penge-*set*-an atribut *historical* pada *file smb.conf*.

```
nakula:/etc/samba# xpattr -s hist -o maxhistcnt=1,minhistcnt=1 smb.conf
historical flag         : Retained_Name
maximum files           : 1
minimum files           : 1
current files           : 0
maximum write trigger   : 1
maximum size of files   : 0 KB
current size of files   : 0 KB
checkpoint time         : -
interval time trigger   : 0 mins (0 hours)
maximum saved date      : -
minimum saved date      : -
nakula:/etc/samba#
```

Gambar 5.6 Setting atribut historical di file smb.conf

Selanjutnya dibuat *script* sederhana untuk mengembalikan isi *file smb.conf* disaat *booting*, dan *script* tersebut diletakkan di *file /etc/init.d/bootmisc.sh* (gambar 5.7). *File bootmisc.sh* akan dieksekusi saat *booting system*.

```
#
#       Save kernel messages in /var/log/dmesg
#
if [ -x /bin/dmesg ] || [ -x /sbin/dmesg ]
then
        dmesg -s 524288 > /var/log/dmesg
elif [ -c /dev/klog ]
then
        dd if=/dev/klog of=/var/log/dmesg &
        dmesg_pid=$!
        sleep 1
        kill $dmesg_pid
fi

#
#       Remove ".clean" files.
#
rm -f /tmp/.clean /var/run/.clean /var/lock/.clean

# set /etc/samba/smb.conf back to origin
/usr/bin/hsfile -s 1 /etc/samba/smb.conf
                                                  76,1          94%
```

Gambar 5.7 Setting script di bootmisc.sh

Sebelum di-*reboot*, dilakukan perubahan di *file* konfigurasi *smb.conf* tersebut berkali-kali. Dan setelah melalui proses *reboot*, *file smb.conf* telah kembali ke isi semula, sebelum terjadi perubahan.

Uji coba yang kedua ini berhasil melakukan pengembalian isi *file* ke semula dengan memanfaatkan fasilitas atribut *historical file*.

5.2.3 Uji coba III

Uji coba ketiga adalah dengan melakukan beberapa kali perubahan ke *file* konfigurasi *modules*, dan mencoba membandingkan masing-masing *file* versi dengan program *diff*. Gambar 5.8 memperlihatkan data awal dari *file modules*.

```
nakula:/tmp# cat modules
# /etc/modules: kernel modules to load at boot time.
#
# This file should contain the names of kernel modules that are
# to be loaded at boot time, one per line.  Comments begin with
# a "#", and everything on the line after them are ignored.

nakula:/tmp#
```

Gambar 5.8 Isi file modules pada awalnya

Setelah melakukan 3 kali perubahan dengan menggunakan program *editor vi*, *file modules* menghasilkan 3 file versi. Dikarenakan *setting* parameter pada nilai maksimum *trigger* penulisan adalah 1, maka setiap kali terdapat penulisan *file*, akan men-*generate file* versi. Gambar 5.9 memperlihatkan isi *file* modules saat terakhir.

```
nakula:/tmp# cat modules
# /etc/modules: kernel modules to load at boot time.
#
# This file should contain the names of kernel modules that are
# to be loaded at boot time, one per line.  Comments begin with
# a "#", and everything on the line after them are ignored.

snd-intel8x0
8139too
lp
usbcore
nakula:/tmp#
```

Gambar 5.9 Isi file modules setelah mengalami perubahan

Selanjutnya dipanggil semua *file* versi milik *modules* ke *directory* tempat *current file* berada. *File-file* versi tersebut mempunyai nama *modules.1, modules.2, modules.3*. Semakin kecil nilai versinya maka semakin lama *file* versi tersebut tersimpan di *version set modules*. Gambar 5.10 memperlihatkan perbedaan antara *file modules* dengan *file* versi yang pertama. Gambar 5.11 memperlihatkan hasil perbedaan antara *file modules* dengan *file* versi yang kedua, dan gambar 5.12 memperlihatkan perbedaan antara *file modules* dengan *file* versi yang ketiga.

```
nakula:/tmp# diff -u modules modules.1
--- modules     2005-01-11 17:23:20.000000000 +0000
+++ modules.1   2005-01-11 17:23:06.000000000 +0000
@@ -4,7 +4,3 @@
 # to be loaded at boot time, one per line.  Comments begin with
 # a "#", and everything on the line after them are ignored.

-snd-intel8x0
-8139too
-lp
-usbcore
nakula:/tmp#
```

Gambar 5.10 Perbedaan current file dengan file versi 1

```
nakula:/tmp# diff -u modules modules.2
--- modules     2005-01-11 17:23:20.000000000 +0000
+++ modules.2   2005-01-11 17:23:16.000000000 +0000
@@ -5,6 +5,8 @@
 # a "#", and everything on the line after them are ignored.

 snd-intel8x0
+ide-cd
+ide-detect
 8139too
 lp
 usbcore
nakula:/tmp#
```

Gambar 5.11 Perbedaan current file dengan file versi 2

```
nakula:/tmp# diff -u modules modules.3
--- modules     2005-01-11 17:23:20.000000000 +0000
+++ modules.3   2005-01-11 17:23:20.000000000 +0000
@@ -5,6 +5,8 @@
 # a "#", and everything on the line after them are ignored.

 snd-intel8x0
+#ide-cd
+#ide-detect
 8139too
 lp
 usbcore
nakula:/tmp#
```

Gambar 5.12 Perbedaan current file dengan file versi 3

5.2.4 Uji coba IV

Uji coba keempat adalah dengan melakukan serangkaian percobaan oleh aplikasi *editor*. Beberapa aplikasi tersebut melakukan perubahan terhadap *file*

yang sudah diberi atribut *historical file*. Tabel 5.1 menjelaskan beberapa aplikasi yang dapat berjalan dengan atribut khusus *XPackfs*.

Tabel 5.1 Uji coba aplikasi historical file

Aplikasi	Tipe file akses	Status
KWord (ver. 1.3.4)	teks dokumen (.kwd)	sukses
KSpread (ver. 1.3.4)	spreadsheet (.ksp)	sukses
KPresenter (ver. 1.3.4)	presentation (.kpr)	sukses
KFormula (ver. 1.3.4)	teks dokumen (.kfo)	sukses
Write OpenOffice (ver. 1.1.2)	teks dokumen (.sxw)	sukses *
Calc OpenOffice (ver. 1.1.2)	spreadsheet (.sxc)	sukses *
Impress OpenOffice (ver. 1.1.2)	presentation (.sxi)	sukses *
Draw OpenOffice (ver. 1.1.2)	image (.sxd)	sukses *
Gimp (ver. 2.0.6)	image (.xcf, .jpeg, .gif, .png, .bmp)	sukses
Bluefish (ver. 0.13)	HTML (.htm, .html)	sukses
Mcedit (ver. 4.6.1)	teks	sukses
Vim (ver. 6.3)	teks	sukses
Nano (ver. 1.2.4)	teks	sukses
Source Navigator (ver. 5.1.4)	teks (.c, .cc, .java)	sukses
MS Word 2002	teks dokumen (.doc)	gagal
MS Excel 2002	spreadsheet (.xls)	gagal
MS Powerpoint 2002	presentation (.ppt)	gagal
Abiword (ver. 2.0.14)	teks dokumen (.abw)	gagal
Strip (ver. 2.1.5)	binary	gagal
Hexedit (ver. 1.2.2)	binary	sukses
Gcc (ver. 3.3.4)	binary	gagal

* = setiap penulisan membuat 2 file versi

5.3 EVALUASI UJI COBA

Dari uji coba yang telah dilakukan, terlihat bahwa *file system XPackfs* yang berjalan di *native file system* yang dimiliki Linux telah berjalan dengan baik. Komunikasi antara aplikasi *user* dengan *kernel* melalui *system call ioctl* telah berjalan dengan baik. Hal ini dapat disimpulkan dari keberhasilan skenario-skenario yang telah ditentukan sebelumnya.

Keberhasilan skenario-skenario tersebut ditunjukkan dengan fungsi *historical file* yang bekerja di *Hostfs*. Dan juga dengan keberhasilan komunikasi melalui *utility userspace* yang dikembangkan di Tugas Akhir ini, meliputi *xpattr*, *lsxpfl*, dan *hsfile*. Akses ke *database inode* maupun *database* atribut *file* juga bekerja dengan baik.

Namun terdapat beberapa masalah yang terjadi, yakni beberapa aplikasi (*editor* umumnya) mempunyai karakteristik yang berbeda dalam melakukan operasi akses *file*, dalam hal ini perubahan isi *file*. Ada beberapa karakteristik yang tidak mampu untuk ditelusuri oleh *XPackfs* dalam melakukan perubahan *file*, yaitu apabila aplikasi menggunakan *temporary file* untuk melakukan penulisan data, dimana setelah itu menghapus *file original* dan mengganti (*rename*) *temporary file* sebagai original *file*.

Masalah teknis lainnya yang belum diperbaiki adalah saat pemanggilan *file* versi, yakni dengan membuat *virtual file* (milik *Pseudofs*) yang berada di hirarki *Hostfs*. *Default*-nya jika *current file* terhapus, semua *file* versi yang terbuat akan dihapus juga dari hirarki. Namun untuk periode dan kondisi tertentu yang belum dapat diketahui, *file* versi tersebut tidak dapat dihapus dari hirarki *Hostfs*.

BAB VI

KESIMPULAN DAN SARAN

6.1 KESIMPULAN

Dari pelaksanaan pembuatan tugas akhir ini, dapat diambil beberapa kesimpulan yaitu :

- *Historical file system* yang berjalan di *native file system* Linux khususnya Ext2 dan Reiserfs telah dapat diimplementasikan.
- Penggunaan teknik injeksi fungsi VFS pada objek-objek milik *native file system* dapat digunakan sebagai alternatif dari teknik *stackable file system*.
- System sudah dapat bekerja secara transparan dalam pembentukan versi *file*, tanpa perlu modifikasi dari aplikasi yang ingin memanfaatkan fasilitas *versioning*.
- Beberapa aplikasi mempunyai perilaku yang berbeda saat melakukan perubahan *file*. Hal ini menyebabkan beberapa aplikasi tidak dapat berjalan pada model *historical file*.

6.2 SARAN

Sebagai pengembangan dari perangkat lunak yang telah dibuat ini, dapat dilakukan lagi peningkatan yang berupa :

- Manajemen penyimpanan informasi baik itu *database inode* atau *database* atribut perlu dioptimasi dengan menggunakan algoritma yang lebih baik untuk teknik pencariannya (misalnya menggunakan *B-tree*).
- Perlunya penambahan *field* di struktur data objek-objek VFS untuk menghindari bentrok dengan *Hostfs*. Dan juga perlunya kehadiran *XPackfs* dapat diketahui oleh *Hostfs*.
- Perlunya penyelidikan dan pembenahan lebih lanjut tentang perilaku *XPackfs* tersebut, karena terdapat masalah teknis yang belum dapat diketahui.

DAFTAR PUSTAKA

[1] *E. Zadok, J. Nieh*, FIST: A Language for Stackable File Systems, 2000.

[2] *E. Zadok, I. Badulescu, A. Shender.* Extending File Systems Using Stackable Templates, 1999.

[3] *K. Kumar, C. P. Wright, A. Himmer, E. Zadok.* A Versatile and User-Oriented Versioning File System, 2004.

[4] *D. P. Bovet, M. Cesati.* Understanding the Linux Kernel, 2nd Edition, O'Reilly, 2002.

[5] *S. D. Pate.* Unix Filesystem: Evolution, Design, and Implementation, Wiley, 2003.

[6] *J. Dike.* User Mode Linux HOWTO, http://user-mode-linux.sf.net/UserModeLinux-HOWTO.html.

[7] *I. Bowman.* Conceptual Architecture of the Linux Kernel, 1998.

[8] *I. Bowman, S. Siddiqi, M. C. Tanuan.* Concrete Architecture of the Linux Kernel, 1998.