Development and Implementation of a File System for Gannet Virtual Machine

Galih Kanigoro

A dissertation presented in part fulfillment of the requirements of the Degree of MSc in Information Technology at the University of Glasgow

September 2009

Acknowledgement

First and foremost I am grateful to Allah SWT, the one and only, who has led me through this journey of life.

I would like to express my sincere gratitude to Dr Wim Vanderbauwhede for providing me an opportunity to do this research. I am thankful for his advices and guidance during the development of the ideas of this dissertation.

I am very thankful to my parents and my sisters for their unconditional love, support, pray and cheerfulness. I am also thankful to Soraya Dian Insani, for her never ending support.

Abstract

This project is addressed to design and implement a file system to organize and store a set of service configuration information and all the data within Gannet system. Since the file system will operate on embedded system, the project development focuses on the process and space efficiency. Unlike other file systems which are implemented on top of kernel, this file system is designed to be able run without rely on kernel as Gannet system has no kernel within it. Therefore, the file system should have a kernel-like mechanism. The result of this project is a file system, called GannetFS , which successfully integrated to the Gannet system.

Table of Contents

Introduction

Digital electronic devices nowadays typically consist of one of more integrated circuit chips that are contain several components such as memory, hardware engine and processors which are referred to as system-on-chip (SoC). By using the concept of Soc, a digital device is able to perform multiple function system on a single chip. The common examples of these systems are mobile phones, video game consoles and network routers. The SoC designs become more complex rapidly as it is designed to handle the complex application (Pasrcha & Dutt, 2008). In order to accommodate the complexity, the efficiency on performing functionality and quality should be very optimal.

Unfortunately, the shared bus paradigm that used on SoC design has a power and performance bottleneck to accomplish complex application. Due to the considerable number of logic blocks and their connection, the bus-style interconnection within the traditional is no longer viable option. In order to response this problem, Network-on-Chip (NoC) architectures are introduced. The dedicated on-chip network allows the chip to optimize transmission channels to address performance, reliability and power issues (Kogel, 2006).

Furthermore, Network on-chip architectures have a networking algorithm which ensures the high optimization traffic management on chip. This condition encompasses the lack of traffic management capabilities on shared bus as it has a mutual dependency between all components. By using Network on-chip, the ad-hoc communication on shared bus could be substituted with allocation of the required communication service and the networking algorithm on Network on-chip is responsible to provide the required resources.

From architecture point of view, the separation of the communication service from the architectural resources can be thought as a virtualization of the actual communication architecture (Kogel, 2006), which decouples of mapping problem for communication and computation.

Gannet system was proposed to enable the design of large and complex reconfigurable SoC which connected over Network on-chip on high abstraction levels (Vanderbauwhede, 2007b). It consists of multiple processing cores which could be realized by software or hardware and each processing core provides a single specific service. Gannet system is addressed to be able implemented as an embedded system which can be defined as information processing systems that embedded into a larger product (Heath, 2003). The purpose of this project is to design and implement a file system for Gannet system to make Gannet system fully dynamic reconfigurable.

1.1 Statement of Problem

Gannet system was proposed to be a novel type of reconfigurable System-on-Chip (SoC) architecture which performs tasks through services interaction offered by heterogeneous multi-core SoCs. It considers processing core as a service in a similar way as services on network (Vanderbauwhede, 2007a). Set of services and their interaction defines the functionality and the behavior of the system.

Gannet system is reconfigurable on two levels (Vanderbauwhede, 2006b). The first level is by changing the task description. Task description defines the functionality of the system. Gannet system performs set of tasks by executing the task description. Therefore, changing on task description changes the functionality and behavior of Gannet system. The second level is by loading new service configuration information from the service library. This condition changes the functionality of a service. At the moment, current version of Gannet system only supports the first level of reconfigurable mechanism.

To enable a fully dynamic reconfigurable SoCs, a mechanism to maintain service libraries as global variables is required. This project is addressed to design and implement a file system for Gannet system. Hence, instead of using static library for each service, Gannet system can use shared object library stored on the file system. By accessing those service configuration from file system, the functionality of Gannet system is able to be changed on run time which makes system fully dynamic reconfigurable.

This dissertation consists of seven chapters. The second chapter reviews the existing file system, as well as an introduction and background knowledge about file system. The third chapter presents the requirements that must be satisfied on this project. The fourth chapter provides a brief overview of the design aspect regarding the file system project. The chapter five describes detailed of the implementation of file system as well as its integration to Gannet system. The chapter six provides a detailed discussion regarding the testing and evaluation about the project. The final chapter is the summaries of the report.

Survey

2.1 Gannet System Overview

Gannet is a distributed service-based-architecture for multi-core Systems-on-Chip (SoC) (Vanderbauwhede, 2006a). As it multi-core system based, Gannet has several core service nodes. Each node based on Intellectual Property (IP) concept represents a single particular service and connected by on-chip network. Vanderbauwhede (2006b) defines reconfigurable service nodes as a set of blocks which each function is depend on configuration information.

2.1.1 Gannet Architecture

Gannet system consists of core system, gateway module, service lookup module and service reconfigurable manager. Figure below describe the high level architecture of the Gannet System.

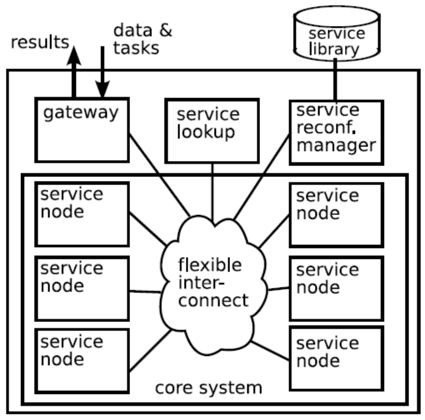

Figure 2.1 Service-based SoC architecture (Vanderbauwhede, 2007b)

As shown on diagram above, every nodes or modules are connected with flexible interconnection which represented as network on-chip. Exchange data between nodes or modules are carried out over the network on-chip based on packet-switching.

The first module on the Gannet system is Gateway. It is an interface between the Gannet system and another device so that they can perform an interaction and exchange data. The next module is service lookup module has responsibility mapping a service to its physical node address. As Gannet system is reconfigurable, its behaviour also becomes the subject of change (Vanderbauwhede, 2007b). The behaviour is established by composition of several service nodes. As the task description received by Gannet system changes, the composition of service nodes also changes. To mapping the physical node address of each service node, service lookup module is needed.

The biggest module on Gannet System is a core system which is responsible to a set of services and interacts with each other connected by on-chip network. The node system contains several service node which each of them represents single service. Each service node can interact one to another in order to build a composite service.

However, each service node itself consists of several modules, which are an Intellectual Property (IP) core, local RAM, a transceiver module and a service manager module. The existence those modules on a service node, allow the service node become independent and also allow the Gannet to perform process concurrently. The configuration of modules within the service node is shown on figure below.

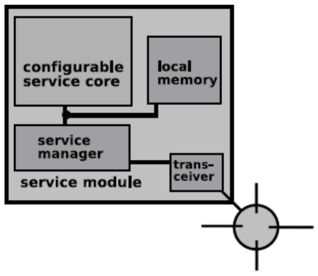

Figure 2.2 Service-based SoC Node (Vanderbauwhede, 2007a)

The transceiver module within the service node contains transmit and receive FIFOs buffer. Moreover, it allows custom implementation depending on the interconnect technology type. The service nodes exchange the data information as packets. The transceiver encapsulates the Gannet packet on a data link-layer frame. Encapsulated established by the deployment of the communication network on-chip. The way it encapsulates is very similar with OSI 7 - layer model but the way it works is very different. As the OSI focuses on quality of service and error control, on the other hand, network on-chip protocol focuses on the efficiency and the effectively the data is being transferred from one point to another endpoint (Kogel, 2006).

The service reconfigure manager manages the data required and produced by the service core. Service node behaves in a “functional” way as they only interact with the system through the service manager and are stateless (Vanderbauwhede, 2006a). Thus, every service will produce identical output for identical inputs and no side-effects. However, this functional behavior accommodates the enable of large-scale parallelism and self-managed distributed processing.

The service manager has a local memory for storing the task description and data required by the core. The memory allocation is independent and managed by the service manager. It also acts as a bridge which connects service node with the service manager in order to exchange the data. Therefore, service node is task-agnostic and the service manager is unaware of the nature of the service.

On Gannet system, tasks are executed based on request. The position of a subtask for a service in the task description determines the way its scheduling. This dataflow execution guarantees the fastest possible execution of a task by fully exploiting parallelism. Service manager plays a crucial role as all communication between the cores is handled by it.

Gannet machine use service manager, a generic circuit, as an interface between the core system and the network on-chip to provide management data facilities and rule-based mechanism for core to process a task and deliver the result. Service manager loads data required for task-level configuration at the boot time, then process exchange data independently without communicate with the microcontroller.

2.1.2 Gannet Framework

Gannet system are designed to be able perform concurrency offered by massively multi-core systems which mean computation will be executed in parallel (Vanderbauwhede, 2007a). In order to perform such type of computation, Gannet adopt a functional approach to task graph composition. A service provided by each IP core can be analogized as a function which has input and output. Functional composition means that all data are the results of calls of other services.

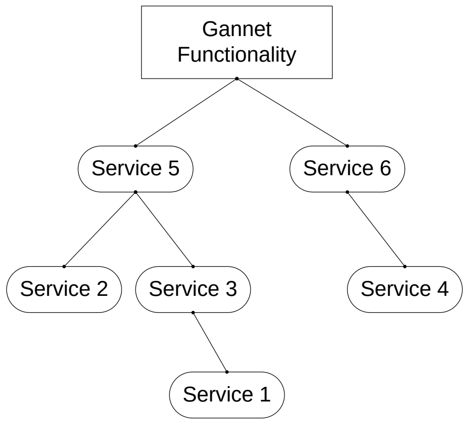

Figure 2.3 Service Node Composite

As shown from figure above, the functionality of Gannet system is composed by several service nodes with single node represent single service. The communication between nodes transmitted over the network on-chip, both either on virtual machine or physical and based on packet system. The operation sequences are mentioned as follow (Vanderbauwhede, 2007b):

- First of all, service managers receive code packets which contains byte-compiled subtask. Every subtask represents references to other subtasks.
- Service manager actives the subtask as the corresponding reference packet is received.
- The execution of this particular subtask results in dispatch of reference packets to other subtasks.
- Tasks without any references are executed by the service core and the results packets are returned.

2.2 File System Background Knowledge

A file system provides the mechanism for storing and accessing data. It consists of two types of entities. First is collection of files which storing the data. Another one is directory which organizes set of files inside. Hence, a file system provides a high level abstraction of file on a system. it provides not only a way to store and organize data as files, but also a set of function that can be run over files on file system.

This section will describe several techniques for storing file in storage and describe the structure of directory. Moreover, this section also describes the storage management of files and directories on a storage device.

2.2.1 Storage Management

The management of disk is a major concern on file system as files and directory stored on disk. There are two approaches for storing a file. First is dynamic allocation disk approach by using consecutive bytes of disk space. Every file has different size of storage space depending on its size. Unfortunately, it has major drawback. The problem arises when a file grows, as this approach has no support for enlarge the dynamic block, it needs to be allocated to a new disk space which has bigger size.

Secondly is fixed allocation storage by dividing disk into fixed-size blocks. However, the size of fixed blocks is a big issue on file system design. Because, if using small allocation unit, it means that a file will consist of number of blocks. Writing a file to disk will consume large overhead as there will be number of allocation block for a file. Furthermore, reading blocks requires a seek overhead and delay. Hence, reading and writing a file on small blocks will be slow.

On the other hand, if large allocation unit is used, the reading and writing file will be faster than small allocation. Since it needs few numbers of blocks, the overhead time produced will be less comparing to small allocation blocks. However, this approach is not space efficient as it has big possibility to waste disk space. Especially, on the last block as it usually not filled fully by data. Thus, the bigger size of allocation unit, the bigger disk space to be wasted.

2.2.2 File Allocation

Files are stored on storage media which support direct access. However, there is a different point of view how to allocate file on a storage media concerning on utility effectiveness and accessed time aspects. Methods of allocating storage space are explained as follows.

2.2.2.1 Contiguous Allocation

Contiguous allocation is an allocation mechanism to store data on a contiguous run of block. This mechanism has significant advantages. Firstly, it is very simple to implement as it keeps the track of blocks of file by using two attributes: the address of the first data block on the disk and the number of blocks used. Furthermore, as this approach only remembers two attributes to manage a file written on the disk, the overhead is quite low. Second advantage, the read performance is very good as it only needs single operation which is a seek operation to find the first block of a file. After that, no seek operation needed as rest of file's data blocks is stored on a contiguous run.

However, contiguous approach has a big disadvantage. Since, as the disk becomes fragmented, it will have number of files and holes. Eventually, the disk utilization will increase and either, the compactness of disk or free space reuse is required. Furthermore, in a creation of new file, it is required to have information regarding the size of file in order to locate a correct hole to store it in.

2.2.2.2 Linked List Allocation

Linked list allocation method uses linked list of disk blocks to store files. On each block, there is a word of pointer which pointed to the next one. The rest of block is reserved to store data. This mechanism has significant advantages. Firstly, it has no limitation of the maximum size a file can be stored on a file system. It is possible for a file system to store a very large file that consumes all data blocks on disk. Secondly, unlike the contiguous mechanism, there is no wasted space to disk fragmentation. Hence, this mechanism is space efficient.

However, this approach consume considerable amount of time for random access. Firstly, to obtain the n-th block, the file system has to start looking up from the beginning and read block by block at a time until n times read. Hence, the overhead on this operation is quite big. Also, the size of data stored on a block is not a power of two as the pointer consumes few bytes. This process could decrease the efficiency as most of programs read and write in blocks in size of power of two (Giampaolo, 1999).

2.2.2.3 Indexed Allocation

Linked allocation suffers from high overhead for random direct access as the pointers to the next data block are spread with the block itself all over the disk.

Indexed allocation has different approach to overcome this problem. It brings the entire pointer together in same location which is the index block. Thus, a file has an index block which contains array of address of data block.

This mechanism has an advantage as it enables random direct access as it simply access the index block to get a particular address of data block. Furthermore, this allocation method does not suffer from external fragmentation as linked allocation.

However, indexed allocation also has wasted space if the file is quite small. For instance, if a file only has one or two blocks, an entire index block must be allocated. Hence, there rest of index block is wasted.

2.3 RAM-Based File System

This sub section provides discussion about the previous works and their limitation. Since GannetFS file system will be implemented on RAM memory, the discussion is limited to RAM-based file system only. Although there are many other file systems based on RAM but, they provide other features like persistent RAM-based, MRAM-based, NonVolatileRAM-based and Hybrid RAM/Disk file systems. Those file systems are not discussed in here as they use different approaches and have different purposes.

2.3.1 RAM Disk

RAM disk is reserved memory which accessed through device interface. At initialization, RAM disk formats a region of RAM device to emulate a hard disk. As an area has formatted, RAM is then mounted as an ordinary disk file system. The advantage of this approach is that RAM disk creating a file system on a RAM device by following the existing file system interface.

However, as it follows the disk file system interface, RAM drive is still use cache mechanism. Therefore, a data can be maintained on two locations cached in the disk buffer and in the RAM drive. It leads to double memory consumption (Snyder, 1990). Furthermore, RAM drive still uses disk file system on its emulated drive despite having no mechanical limitation which leading to slower the performance.

The idea to reserve amount of region on RAM is suitable for GannetFS file system implementation as there is no kernel on Gannet Virtual Machine. Hence, GannetFS file system has to manage its own region of memory.

2.3.2 RAM File System

RAM file system (McKusick 1990) is built top of virtual file system (VFS) interface (Kleiman). Instead of allocating reserved fixed region of memory for used as a file system, RAM file system uses dynamically allocation depending on its usage. Furhermore, by using VFS for saving both, metadata and data itself, RAM file system overcomes the doubling memory consumption of RAM drive.

Even though, RAM file system has avoided disk-related interface, it relies on top of VFS, which has generic storage access routines. Hence, this approach leads RAM file system to lower performance.

GannetFS will follow the RAM file system concept. However, instead of implemented on top of VFS, GannetFS will have its own mechanism to handle the interaction. Furthermore, GannetFS avoid using dynamic allocation as in Gannet system there is no kernel to support that mechanism.

2.4 Literature Review

Having discussion regarding background knowledge about file system and the existing RAM-based file system, this sub section provides discussion regarding techniques from several file systems to represent files and directories.

2.4.1 Metadata Representation of Files

The design of metadata for Gannet file system is inherited from UNIX's metadata which is represented with inode data structure (Thompson 1978, McKusick et al. 1996). In recent years, inode data structure has evolved through many generations and it's slightly different compared to the original inode design (McKusick et al. 1984; Card et al. 1994).

The inode for *ext2* has 15 pointers used for locating the location of data blocks. The first 12 pointer point to the first 12 data blocks. After the direct data blocks are consumed, the 13th pointer points to a single indirect block, which point to another data block that contains pointer to data block. The 14th pointer points to a double indirect block, which in turn contain pointer to an indirect block. The 15th pointer points to a triple indirect block, which contain pointers to double indirect blocks.

Unix file system (UFS) and *ext2* have 15 pointers which pointing to the location of data blocks. The first 12 pointer is direct block which point to actual data block.

BeFS file system has a slight different implementation of inode since it does not implement the 15th pointer or triple indirect blocks inside the metadata. As solution to trade with triple indirect blocks, *BeFS* uses array of contiguous data blocks instead of a single data block (Giampaolo 1999). However, both types of metadata explained above are not suitable for Gannet file system since the size of metadata is 128 bytes which consumes considerable amount of space for RAM-based file system.

Another approach to save space consumption for metadata has presented by Conquest (Wang et al. 2006). It removes the nested indirect blocks from the inode, thus the data blocks are accessed through dynamically allocated index array which contains pointer to on data block. However, this approach only allows *Conquest* to store small files inside the RAM and have to store the larger files on static disk. Furthermore, to minimize the size of memory used for metadata, *Conquest* uses a stripped down version of standard metadata structure that used in disk. As it only contains fields that required conforming to POSIX, the size of *Conquest* inode is 53 bytes long. The major drawback of *Conquest* inode is only small files can be accommodated on RAM the file system as the metadata only contain direct pointer on dynamically indexed array.

Another scheme to minimize the size of metadata also presented by HeRMES (Miller et al. 2001). It uses the same version of standard metadata which used in disk, but before the inodes are stored in RAM, HeRMES uses compression techniques to minimize the size of inodes. The major drawback is the overhead occurs in every time the inodes are accessed.

According to Giampaolo (1999), generally an inode will store between 4 and 16 block references directly. There is a tradeoff between the size of the inode and how much data that inode can map. Storing a few block addresses directly in the inode simplifies finding file data as most engineering files tend to weigh under few kilobytes (Ousterhout et al. 1985, Satyanarayanan 1981). Furthermore, Gannet file system will operate on embedded environment which the average of most files tend to be smaller. Thus, in default configuration the inode for GannetFS file system will store 3 block direct references to address and an indirect block. GannetFS does not contain triple indirect block as files on embedded and RAM environment will not exceed the maximum size of double indirect block.

The structure of inode on GannetFS file system will be slightly different with the usual inode on disk media as it will use the striped down version of inode. As pointed by Edel et al (2003), the stripped down version offers a small size of inode without have any overhead process on for compressing it. The size of inode is strictly to have inode's size of power of two. The reason for this condition will be explained in the file system data storage mechanism section.

Another consideration on implementing file system is to determine the ratio between number of inode and number of data block. This ratio is very important since the disk could run out of space if there is no inode available even considerable amount of free data block is available or there is a numbers of free inode but no available data block for a file to write onto. Hence, the ratio between inode and data block determines the effectiveness of space consumption of a file system. Based on empirical experiment (Tanenbaum, 1997), number of inode for 1,44 Mbytes with 1 kb block size are 480. The inode of minix has 7 direct reference block, an indirect block and a double indirect block (Tanenbaum, 2006). By using mathematic formula, the number of inode required to obtain the same ratio for 65 Kbytes with 64 bytes block size is 360 nodes. This number is still rough prediction. Thus, the experiment on actual environment is needed to get the ideal ratio will be used as default number of Gannet inode.

Furthermore, to avoid complicated metadata management, Gannet file system uses the same approaches with *BeFS* and *Conquest* file system which uses the address of metadata as unique ID. By using address as the id for metadata, ensures that its ID is unique. Hence there is no duplication id among the file. Furthermore, it also improves the entries searching process.

The disadvantage of this design is the file system has limited size of files as there is a limitation on how large a data can be stored. However, this constrain is considered not significantly confine the performance of file system as files for embedded system is quite small. Furthermore, the size of data block can be configured to be bigger to allow storing large files.

2.4.2 Metadata Representation of a Directory

Unix directories are considered as files whose data blocks maintain list of entries. The list stores the names and inodes id for each file and directory reside within it. The structure design for the list determines the performance of traversal operation and fast random lookup to locate a file (Wang et al. 2006).

Thus, the mapping of name to inode numbers on directory is very important. There are several ways to maintain the name to inode number mapping. Unix-style file systems use a simple linear list to store the entries of a directory (McKusick et al. 1996). The major advantage is space effective as it is simple to implement. However, the disadvantage rise when there are considerable amount of files reside in a directory. This implementation works fine in small number of files, but will degraded significantly as the number of files increases since the time complexity is O(n).

Another approach to mapping the name and inode number is to use a B+trees as implemented on *BeFS* and also XFS (Sweeney 1996). The data structure stores pairs of key and value as name and inode number respectively, in a balanced tree structure. By using B+trees data structure, the time complexity can be reduced to O(Log n) to look up an item. Furthermore, *BeFS* and XFS also need B+tree data structure for indexing the file attributes.

Another approach is used a hashing table (Fagin et al. 1979) as directory representation. The structure is built by a hierarchy of hash tables, using file names as keys and inode numbers as values. By using hash table data structure, the time complexity can be reduced significantly to O(1). However, this approach has major drawbacks as there will be wasted space for unused hash entries.

GannetFS file system will operate on embedded environment where the number and size of data are small. The number of files in a directory will not exceed a hundred files. Thus, linear list is still feasible to be implemented on Gannet file system. Furthermore, the implementation of B+tree or Hash table in metadata of directory is too heavy if the purpose is only to improve the search time of look-up entries process as it consumes considerably amount of space. The others file system use those data structures also for store the file attributes which is not supported on Gannet file system. To improve the searching time of entries on directory, the system will first compare the length of file name with the target file. If the entry has the same length of the target file, then the system will compare the file name.

Design

3.1 Requirement

This section discusses the requirement regarding the project which based on the purpose of the project and the specific requirements for Gannet Virtual Machine system.

In order to build system right, the gathering of the requirements process is very important for the understanding the needs of the system. The main requirement capture phase has performed by interviewing with the project supervisor Dr. Wim Vanderbauwhede.

3.1.1 Functional Requirement

The purpose for this project is to design and implement file system for Gannet system. The file system must support file and directory operations. However, the file system does not required to be a complete file system which considers aspect and mechanism regarding the reliability, consistency checking and security.

The file system must be able to operate on Gannet system which has no kernel inside. As Gannet system is embedded system which operates on high constraint environment, the file system also must adapt with the similar environment. Hence, the file system should be effective on time processing and space consumption aspects.

3.1.2 Non Functional Requirement

As discussed above, Gannet system is embedded system that operates on high constraint environment. The constrain demands the system to be able have an interaction and communication with other devices. However, those devices are issued from different vendors and built on different platform. Hence, the file system's interfaces must compliance with range of vendors, platforms and also architectures. The file system also must be built on the same programming language as Gannet system which is C/C++.

The final section of the requirement gathering specification for both functional requirement and non-functional requirement is listed in appendix A. More discussion regarding the aspect to design the system can be found in a later section.

3.2 File System Design

Having requirement specification, the next step is designing upon a data structure. Since the file system is developed on C program language, instead of explaining the class, in this section I explain the design of data structures underneath GannetFS file system.
GannetFS file system is designed to be implemented on GannetVM platform. It is designed to be a reconfigurable file system that has standard POSIX file system interface to its user. As GannetVM does not have secondary storage device, instead of establishing file system on hard disk, GannetFS is built on Gannet's RAM memory region which reserved for this purpose. GannetFS file system is also designed to be able to operate without depending on system call from kernel since GannetVM platform does not have kernel or operating system. In Gannet system, all services are performed by service cores instead of kernel.

GannetFS file system is different with other file systems since GannetFS does not depend on any kernel. Meanwhile, other file systems are built on top of kernel or operating system. Other existing file systems are built on top of kernel and using system calls for accessing the device. Unlike the others, GannetFS file system is designed to operate on an environment where kernel or Operating System does not exist. Therefore, GannetFS file system should have a mechanism for accessing storage device. Furthermore, GannetFS must have a set of library to support the operation of GannetFS.

3.2.1 Storage Management

Storage allocation mechanism is needed to manage storage on GannetFS file system. The mechanism should be able to allocate a size of memory as requested by file system and to free those allocated memory when they are not used anymore. In order to build this mechanism, GannetFS file system divides a region of memory into fixed size of allocation units which are stored as a simple array data structure and uses stack data structure to manage the array.

The mechanism is quite straight forward. The stack contains relative address of unused blocks within array. When system requests blocks of memory, stack performs ‘pop’ operation which returns top most elements. These returned values are relative address of unused blocks. Then, blocks with corresponding relative addresses are allocated and returned to the system. On the other hand, while the system does not need particular blocks, it pushes those addresses to the stack so they are placed onto a stack and they can be used anymore. By using array - stack mechanism, GannetFS file system has its own storage allocation mechanism.

3.2.1.1 File System Layout

GannetFS file system is a logical entity with inodes and data blocks. From higher level point of view, GannetFS consist of an array of data blocks which is managed and marshaled by two stacks data structure.

Initial Design of Data Object Layout

The initial design to organize the data object on GannetFS file system is by using two types of arrays to accommodate the metadata file and data file, respectively. This array contains number of fixed allocated memory called block. Each type of array has a stack to manage the allocation of block as illustrated on diagram below.

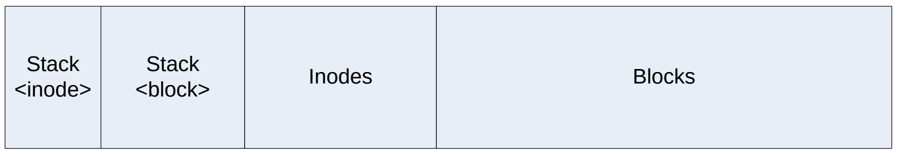

Figure 3.1 Illustration initial design of data object

Stack has function to manage the allocation for blocks by storing a set of free blocks. As the system needs a block to store a file or metadata, the stack returns an address of free block and pop the address out of the stack. If the system has freed the allocation of a block, the address of unused block is inserted to the stack. Hence, by using this mechanism, the system always knows the address of free blocks and avoids writing a new entry to non free blocks which causing a data loss.

In this approach, inode blocks have an array of inodes and a stack managing the allocation of inode. The size of blocks on this array follows the size inode, which in this case is 32 byte. Similar to array of inode, data blocks also have its own array and stack with larger size, which is 64 byte per block. The advantages of using different array for metadata and inode are easy to manage, simple and each array is independent from one to another.

However, this approach requires system to access the array of inode to access the metadata of a file and then accessing the array of data block to access the actual data. The overhead process is quite high as the system should access the different array. Whereas, updating file system metadata over the files dominating the operation on a file system (Ousterhout et al. 1985, Satyanarayanan 1981).

Design of Data Object Layout

This array contains number of fixed allocated memory called block. This is the main resource of GannetFS as inodes and blocks reside within this array. Data blocks array is divided into two zones.

First zone is inodes region which is an area on data block that is reserved only for inodes. However, the size of inode is usually smaller than size of block. Thus, to save space consumption, instead of allocating one inode per block, a block can be divided into numbers of chunks so that a block can be used to store numbers of inodes. Since the size of inode and block are designed to be power of two, no space being wasted on division process.

The second one is data blocks region which is an area on data block to put actual data in. This approach is quite straight forward as it uses storage allocation mechanism.

The division of block into several chunks seems like creating a new small size array within an array. This approach is exactly similar to the initial design, but in this approach the system only calls one array. Hence, the overhead process is kept as minimum as possible.

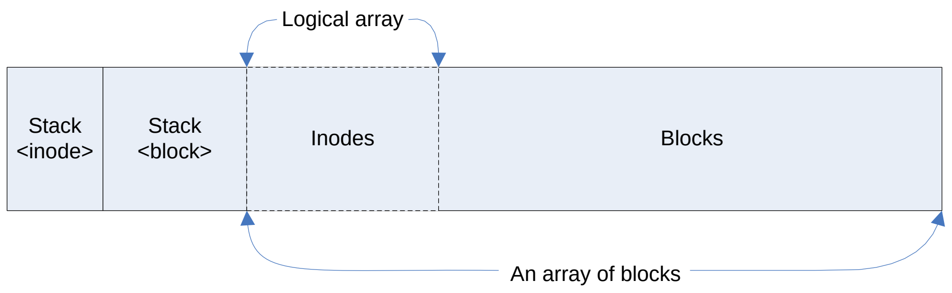

Figure 3.2 Illustration design of data object

The fixed partition between inodes and data blocks leads a file system designer to have a consideration. The designer has to determine the ratio between number of inodes and number of data blocks. This ratio is very important since the file system could run out of space if there is no inode available even considerable amount of free data block is available or there is a numbers of free inode but no available data block for a file to write onto. Hence, the ratio between inode and data block determines the effectiveness of space consumption of a file system.

In order to determine ratio between number of inodes and number of data blocks, a file system designer should have a picture regarding the environment of the file system. If the file system tends to operate with small number of large file, the designer can use few numbers of inodes. On the other hand, if the file system is projected to work on environment with large numbers of small sized file, the designer should use large number of inode.

3.2.2 File Allocation

As discussed on previous chapter, GannetFS file system is designed to use multilevel indexed file allocation. Therefore, GannetFS uses metadata as representation files that reside within file system. The metadata stores all information about a file. The design of metadata for GannetFS file system is inherited from UNIX's metadata which is represented with inode data structure (Thompson 1978, McKusick et al. 1996). In recent years, inode data structure has evolved through many generations and it has slightly different compared to the original inode design (McKusick et al. 1984; Card et al. 1994). However, GannetFS inode data structure is very different with others inode as GannetFS inode is designed to be space efficient.

3.2.2.1 Inode Data Structure Design

GannetFS inode data structure represents an object within file system. This data structure maintains all the information about a file. It records information such as the size, description and actual address of a file. The last one is the most important information about where the data lies on memory, as it is then possible to locate all the blocks of the file. GannetFS inode is shown as figure below.

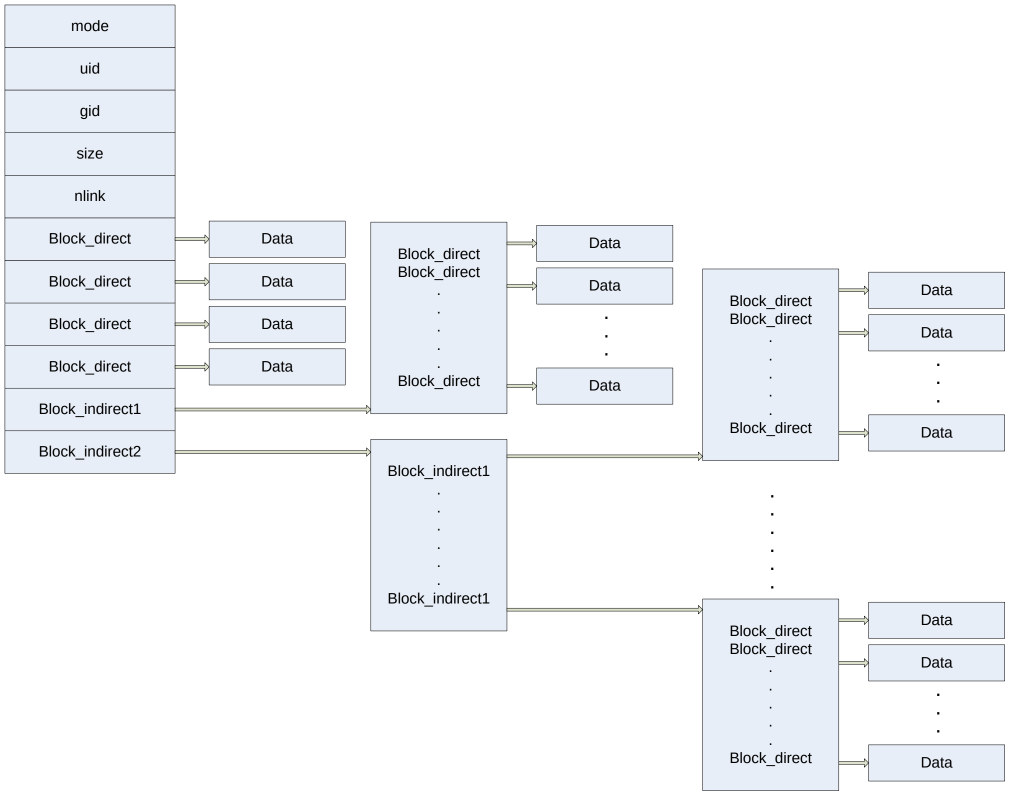

Figure 3.3 GannetFS inode structure.

The structure of GannetFS inode is a stripped down version of inode on UNIX. It only contains five basic fields. The first field is mode which stores information about the type of file. There are six types of file on actual file system. But according to project requirement, GannetFS file system only required to support only two types of file. The first type is a regular or ordinary file which represented as a stream of bits contained on one or more blocks. It does not have any special internal structure as it only treats them as a stream of bits. The second one is directory type which contains hierarchical internal structure. Instead of maintaining a set of bit, directory file maintain set of directory entry. This directory entry could be a file or another sub directory.

GannetFS file system treats these two types as the same file by considering them as stream of bytes. However, the difference between them is on how interfaces treat them.

This field also contains information regarding the file access permission. The file permission is required by POSIX 1003.1 specification. There are three different privileges for accessing file, namely user, group, and outside. For each privilege, there are three permission types, namely read, write and execute. This mechanism makes data accessing in file system more secure as there is an authorization process for accessing files. This mechanism has strong relationship with the next two fields, which are uid and gid fields that represent user id and group id of a file ownership, respectively.

At the moment, Gannet system has two different types of file. The first type is a task description file which contains set of operation assignment for a service. This file only should be executed by an authorized service only. The second one is data file. This file contains data to be processed or result from service operation. Therefore, this type of file is never to be executed. File access violation can happen when unauthorized service executes a task description file. A violation also happen when a service execute data file. Any violation on accessing those types of files may cause Gannet system to crash.

In order to avoid any kind of violation on Gannet platform, permission and privilege mechanism should be used. A service cores represent a user, while set of services core represents a group in GannetFS file system term. This representation of user and group allows GannetVM to have different privilege level among its service cores. By setting up the privilege bit on inode field mode, GannetFS applies authorization checking process. Therefore, only an authorized service can access certain file. Then, the permission of task description file should be set for execution only and permission of data file should be set for read and write only. Hence, violation of accessing those files could be avoided.

The next fields are size and nlink which store information regarding size of file in byte and number of entries that referenced to the corresponding inode, respectively.

The last three data members are data block management which provides the connection between the file and the location where those bytes reside on ram memory. It provides a way to map from a logical file position to a file system block at some location on memory storage. Block_direct is an array of pointers to the data blocks in a file. It contain pointer to the actual data block on memory storage. Therefore, it allows inode to keep track the list of blocks on actual memory that belong to a particular file. From high level perspective, a file appears as a contiguous stream of bytes, but on the low level actually the file data is spread over the memory.

For somewhat larger file, block_indirect1 can be used. This field contains a single indirect pointer that points to a single indirect block of pointers that point to the actual data block. For bigger file, singe indirect block, block_indirect1, can be exhausted. To accommodate this kind of file, double indirect block block_indirect2 should be used as it is a pointer to a block of pointers to pointers to data block. In other words, it contains two levels of pointer before reaching the actual data block. Figure 3.3 illustrated how direct and indirect blocks reach the data blocks.

However, first and higher level index approach has an issue regarding the performance aspect. The higher the index level of data block, the more time required for accessing data block. Direct block has time performance O(1) as it contains the actual data blocks. A single indirect block has performance OLog(n) as it points to an index block which contains the addresses of actual data. Then, the double indirect block has the worst performance, O (nLogn), since for accessing a block would require reading in all index blocks to follow the pointer chain until the target block finally could be reached. However, the performance of single and double indirect block is not worse than linked list allocation.

3.2.2.2 Directory Entry Data Structure Design

GannetFS file system directories are considered as files whose data blocks maintain the list of entries. The list stores the names and inodes id for each file and directory reside within it. This data structure holds important role on file system as it converts a name file which is human readable to an inode data structure which is machine readable. This data structure has six fields as shown on figure below.

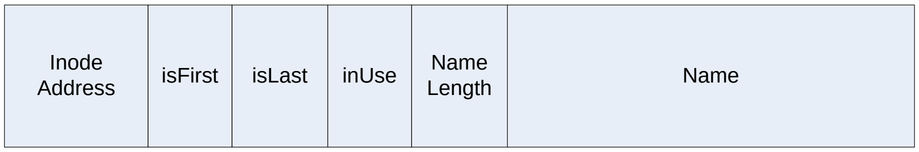

Figure 3.4 Structure of GannetFS directory entry.

The first field contains the address of associated inode. The associated inode refers to a file or another directory. So, it allows user to access the contents of the directory and traverse the file system hierarchy. Furthermore, by using absolute address to identify the associate inode, it allows system to obtain the inode in faster way as it no needs to perform lookup process. The next two fields are first and last flags that indicate that this directory entry is the first or last on directory a block, respectively.

The fourth field is also Boolean data type that indicates that this directory entry is currently used or unused. This field is used on directory entry deletion process. Instead of erasing a directory entry, GannetFS simply unset this field which means current directory entry is 'erased'. The last two fields are namelen and name which store file name length and actual file name.

A GannetFS directory entry is a file that contains static 4 byte of inode address, 4 byte of flags and namelen, and 24 bytes for the file name. The design limited disk partition for directory structure to 32 byte and file names to 24 characters. The use of fixed-length of directory entries, in this case 32 bytes, is a trade off involving speed and storage. GannetFS directory is very simple and space efficient for looking up and storing entries. However, there is wasted on storage, if the name of file is less than 20 characters.

It is different with directory entry on other UNIX which breaks data blocks into variable-length directory entry in order to allow name of files to be of nearly arbitrary length. Thus, the size of directory entry structure varies depending on the filename length. Therefore, there is no space wasted for small file name. However, as an entry is deleted, the entry's space from deleted entry cannot be reused for new entries if the length of file name longer than the deleted one. Hence, the size of directory tends to grow despite there are number of entries deleted.

However, the usage of fixed length on GannetFS overcomes that problem. Since the size is fixed, the entry's space from deleted entry can be used for new entry. Moreover, to minimize the wasted space, the maximum limit for entry's file name is designed to accommodate the mean file name length. Therefore, this approach provides more space efficiency.

3.2.2.3 Translator Data Structure Design

A common request on file system is to look up a file name in a directory. This request commonly stated using name path. However, the system cannot understand the path name as system accessing files and directories by accessing their inodes. Moreover, GannetFS does not support relative address. As the user of GannetFS is service cores on Gannet system which is not human, there is no need to establish a relative address. Hence, a mechanism to resolve an absolute path name into an inode is required. GannetFS uses translator structure to resolve the path name required.

Managing directory and path names is an important aspect on file system. Most of system calls use path names as input parameter. Hence, translator structure is required by file system to resolve the path name into a directory tree and locate the associate inode.

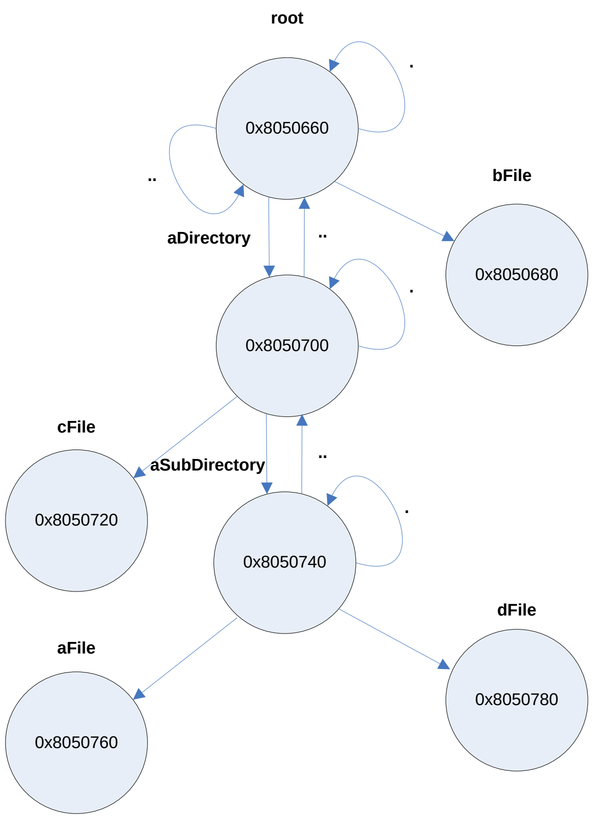

Figure 3.5 GannetFS file system tree

Certain files are distinguished as directory and contain pointers to files that may itself be directory as discussed on Inode data structure section. The directories and files are organized as a hierarchy tree structure. Figure3.5 shows a small tree. Circle represents an inode with its absolute address inside. Each arrow represents a name in directory. For instance, the /aDirectory with entry **.** , which points to itself and entry **..** ,which points to its parent, root. It also contains name cFile, which references to a file and name aSubDirectory which references to its children directory.

3.2.2.4 File Descriptor Table Design

As discussed on previous sub section, most of operations on file system involve looking up an entry associated with the path name. For that purpose, it is convenient to have an id rather than use the original path name. POSIX convention stated that the file id should be identified as a small positive integer called file descriptor.

GannetFS file system has to maintain an opened file. It deals with an issue to indicate the next byte to be read or written. Therefore, file system should have a file pointer to keep track the last accessed byte on a file. Thus, a mechanism to map a files descriptor and file pointer is required to maintain an opened file. In GannetFS file system file descriptor table, each file descriptors is mapped to a stat structure.

Furthermore, the usage of file descriptor table enables a file to be opened by several users. In order to make it simpler, I use a stat structure as the main pointer since it has a file pointer within and other basic information about a file. Since it is only has single file pointer, file system only support on operation per opened file, either read() or write() operation.

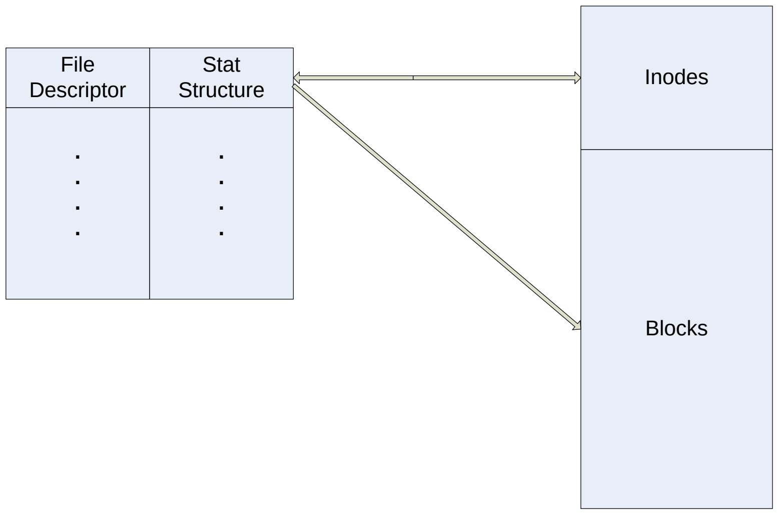

Figure 3.6 File Descriptors Table.

The relationship between file descriptors table and opened files, which are represented by inodes and data blocks is shown as above. This relation between file descriptor and a file structure is established by the open() and creat() functions. Those functions assign a file descriptor for an existing file or new file.

In this table, file is represented by stat data structure which maintains basic information about the file.

3.2.2.5 Stat Data Structure

GannetFS file system uses stat data structure to maintain characteristic of file. It means, stat stores all information about a file. Stat data structure uses stripped down version of POSIX's stat structure to save more space.

The first field is flag which is set on how the file was opened. There are two types of flag that is supported, namely read only and write only. File cannot open for both read and write purpose. Next field is file's mode which represent file's type and permission.

Inode field represents a pointer to an inode that associated to a file. Then, offset field records the offset within the file being reading or reading. The default is zero and will be increased by each subsequent read or write operation.

3.2.3 File System Programming Interface Design

Having discussion regarding the file system data structure, the following section describes how the file system interacts with users or modules. This mechanism also called as an interface. It provides communication mechanisms between modules or systems. Through this interface, the collaboration between modules and objects takes place. The requirement stated that the file system must be platform independent. Thus, a good design interface of file system is crucial for this project.

However, there is a POSIX which describes the contract between the application and its library on the operating system. It is an international standard with set of definitions and assertions which can be used to verify compliance. A conforming POSIX application gains the confidence level that a system is a platform independent

(Lewine 1991). Since POSIX also provides set of interfaces between file system and its operating system, the interfaces for GannetFS file system simple reuse the existing interface from POSIX. The detailed section of GannetFS file system API is listed in appendix C.

Implementation

Having performed a thorough design for the domain, the implementation of GannetFS file system is quite straight forward. This section will now outline the implementation of all data structure discussed on previous chapter and the integration process between GannetFS file system and Gannet Virtual Machine. Implementation of GannetFS file system was performed in default GannetFS file system configuration which has inode size 32 byte and block size 64 byte.

4.1 File System Data Structures Implementation

This sub section provides brief explanation and description regarding the implementation of data structures mentioned on design process. Firstly it will discuss regarding inode implementation with explanation about several issues on compression technique and how indexed allocation that used by GannetFS file system overcome the file limitation. The next subsection, presents discussion regarding the implementation of entries data structure. Then, it is followed by disccusion about file descriptor table and translator data structure for path translation implementation.

4.1.1 Inode Implementation

GannetFS is aimed to be a configurable file system. However, the changes on size of inode only affects on size of direct data blocks. In default configuration where the size of inode is 32 byte, an inode maintains 3 direct data blocks. As size of inode is incremented up to 64 byte, an inode will maintain 11 data block direct and other fields will remain the same. However, if a file system designer need to change the configuration of other fields, for instance number of indirect block in inode configuration, he should change the code manually.

On storage allocation mechanism, GannetFS inode is dynamically allocated which allocates block space to a file as needed. As at the creation of an inode, array of data blocks is not allocated. Those data blocks are being allocated only when there is data to be stored on data blocks. Hence, file system is efficient in term of space usage.

Direct block represents a list of blocks of a file. In this design, inode can store first 4 blocks addresses on inode itself directly to simplify finding file data. This allows the first 192 byte of data to be located directly by using simple indexed lookup. Providing enough direct blocks in the inode to map the data improves the performance. However, there is a trade-off between the size of the inode and how much data on inode can map.

However, for file which has size greater than 192 byte, direct block cannot accommodate the entire file. In order to overcome this constraint, GannetFS inode can use single indirect block, block_indirect1. When an inode uses an indirect block, it stores the block address of the indirect block instead of block addresses of the data. Indirect block contains pointers to the blocks that make up the data of the file. It does not contain metadata about the file, only pointers to the blocks of the file. Thus, with one block address, indirect block can access a much larger number of data blocks.

In GannetFS file system, data block addresses are 4 byte (32 bit). Thus, given a file system block size of 64 bytes and a block address size of 4 byte, an indirect block can refer to 16 blocks. Therefore, by using single indirect block, GannetFS can store file with maximum size 1024 byte.

4.1.2 Directory Entry Implementation

Mapping of name which is human readable to inode on directory holds an important role on a file system. As discussed on Literature Review section, there are several mechanisms to mapping a file name to an inode. Gannet file system will operate on embedded environment where the number and size of data are small. The number of files in a directory will not exceed a hundred files. Thus, linear list is still feasible to be implemented on Gannet file system. Furthermore, the implementation of B+tree or Hash table in metadata of directory is too heavy if the purpose is only to improve the search time of look-up entries process as it consumes considerably amount of space.

GannetFS directory entries are designed to have fixed size regardless the length of file name is long or short. The implementation of directory entry allocation on data block has the same approach to how inodes are allocated on data blocks. The allocation mechanism divides data blocks into numbers of parts. In this implementation, data blocks able to store two directory entries.

On directory entry addition operation, GannetFS first check the size field on the inode. If the size is zero, it means this is the first directory entry to be allocated. So, it simply add directory entry on a data block, set the flag first, last and inuse, and also increment size field on inode with size of a directory entry. If the directory entry is not the first directory entry on particular directory, it simply looks up the chain of directory entry to find directory entry with inuse flag is unset. If a directory entry not inuse were found, it simply overwrites current directory entry with the new one and set the inuse flag. However, if there is no unused directory entry were found, new directory entry is written next to the last directory entry on particular chain, set the last and inuse flag, and increment the size of associated inode.

On directory entry deletion operation, it simply unset the inuse flag on corresponding directory entry so that, this particular chunk can be used for new directory entry. Hence, there is no erasing directory entry on this operation. The next operation is directory entry lookup. File system starts the look up at the beginning of the directory and comparing each directory entry one by one. To increase the speed of searching, instead of directly compare the name file one by one, it simply compares the file name length. In case the lengths are the same, then it compares the string. If the comparison is match, the operation is success and terminated. If not match, compares the next directory entry.

4.1.3 File Descriptor Table Implementation

The implementation of file descriptor table is actually created by using an array indexed by file descriptor number. File descriptor is used to locate associated file as it is presented. File descriptor also represents slot on file descriptor table. A mechanism for slot allocation on file descriptor table is managed using stack mechanism as seen many times on previous chapter. The stack contains free slot index number. Once a file is opened or created, as the stack is not empty, it returns an element which is a file descriptor for associated file. Then, stat structure is created to represent the corresponding file and put onto the file descriptor table at index stated by file descriptor.

The stat structure remains on file descriptor table as long as the file opens and also maintains the read or write pointer at associated file. As the file is closed, the file descriptor is pushed onto the stack to be reused and the stat structure is erased from the file descriptor table.

4.1.4 Translator and Name Path Translation

When a file is opened, it is a must to take the file name and then locate its disk blocks. Consider the path name is /aDirectory/aSubDirectory/aFile. GannetFS does not support relative address. As the user of GannetFS is service cores on Gannet system which is not human, there is no need to establish a relative address. Hence, the starting directory is always root directory. To find the file as mentioned above, the file system looks up the first component of the path, aDirectory, in root directory to get the inode of the file /aDirectory/. Based on this inode, the system locates the next component, which is aSubDirectory. Once the system finds the entry for aSubDirectory, an inode for directory /aDirecotry/aSubDirectory can be accessed. Based on this inode, the aFile file can be opened. Then the inode is kept on file descriptor table until the file is closed. Figure below shows the illustration to search particular file.

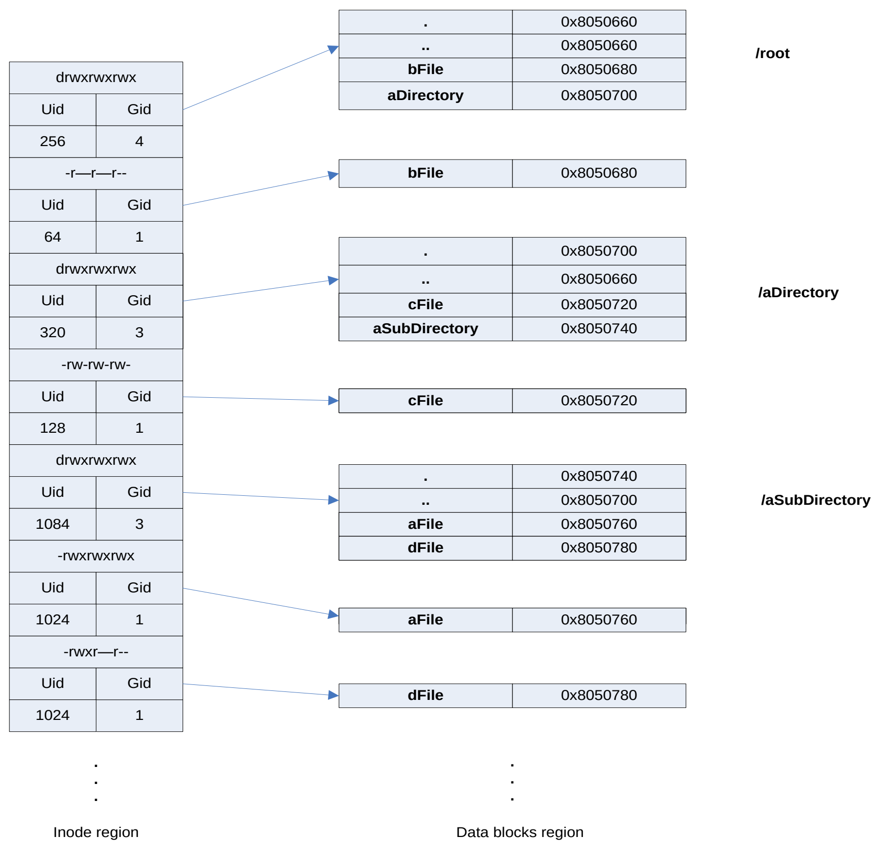

Figure 4.1 Internal structure of GannetFS file system

4.2 File System Operations Implementation

This sub section discuss the implementation of GannetFS file system API's. However, the discussion is not presented on detailed explanation. This sub section provides short discussion regarding the file operation and directory operation on GannetFS file system.

4.2.1 File Operations Implementation

GannetFS file system provides full POSIX compliant API for files operation. There are six interfaces over files operation, namely creat(), open(), close(), read(), write() and lseek().

According POSIX, function creat() and open are similar. The open() function allows creating a new file, so does the creat() function that become a subset of the possible uses of open function for creating a file. However, open() function can open the existing file but not the creat() function. Opening or creating a file has three steps. First step is obtaining the associated inode or allocating a new inode if the file is new. The next step is obtaining the directory entry or creating the directory entry if the file is new. The last step is allocating a file descriptor for the file. The file descriptor becomes an identity for the opened file.

As a file has been opened, this file can be read or written. Once read function is called, the first step is validating the file flag of the associated file. In order for a file to be able for reading, the flag must be marked as read only. Then, GannetFS accessing the sequence of data blocks which starts from the offset pointed by file pointer. The process terminates when all bytes have been read or EOF has been hit. As the read process is finish, the file pointer position is updated to the new offset position. Write function has similar steps as read function. The main difference among them is write function needs allocation new data blocks. However, write function also can terminate if there is no available block to be written into.

The next function is lseek which is very simple as it updates the file pointer to a new given value. As function close is called, the file is closed. The file descriptor is pushed back onto the stack to be reused and the stat structure is erased from the file descriptor table.

4.2.2 Directory Operations Implementation

GannetFS file system provides full POSIX compliant interfaces for directory operation. There are three interfaces over files operation, namely mkdir(), opendir(), closedir() and readdir(). GannetFS treats directory as a file. Therefore, those interface is similar to creat(), open(), close(), and read() function on files operation, respectively.

mkdir() function is similar to creat() function on file operation. However, mkdir() function does not allocate a slot on file descriptor table. As opendir() is called, a slot on file descriptor table is allocated. Unlike open() function on file operation, if the directory is not exist, instead of creating new directory, opendir() function terminates with error. Another difference between them, instead of returning a file descriptor, opendir() function returns a file DIR pointer which is used by readdir() to obtain an directory entry. Readdir() returns the next directory entry form the data block on each call. Instead of reading data byte per byte on data blocks, Readdir() reads directory entry one by one. The closedir() function frees a slot on file descriptor table that allocated by opendir() and closes the directory.

4.3 Integration to Gannet Virtual Machine

As discussed on section 2.1.1 , Gannet system consists of services. The interaction between services determines its functionality and behavior. This architecture of Gannet system facilitates decoupling of processing, thus enables parallelism on data processing. Looking deeper into low level, Gannet system is actually a big ‘loop’ that ensure each service has some portion of cycles to perform the task concurrently.

Gannet system needs data and task description from its client in order to perform a task. The data and task description are received by Gannet system via Gateway as it communicates with outside world trough this module. At the moment, as the data and task description are received, they are then stored on temporary file system which is built from bytecode queue. Since its bytecode queue, the task description must be parsed into bytecode before is put onto the queue. The task description file consists of list of Gannet’s words which is a representation of unsigned integer which has 4 bytes size. Bytecode itself is constructed from a list of words. Once the bytecode is stored on queue, all service can access it.

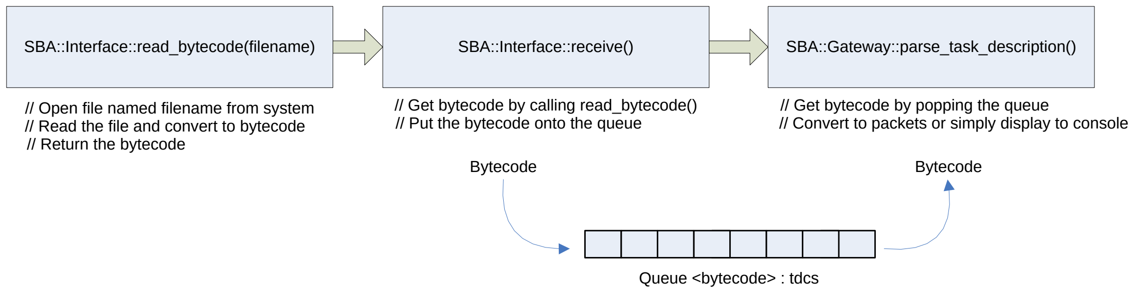
SBA::Interface::read_bytecode(filename)
SBA::Interface::receive()
SBA::Gateway::parse_task_description()
// Open file named filename from system
// Read the file and convert to bytecode
// Return the bytecode
// Get bytecode by calling read_bytecode()
// Put the bytecode onto the queue
// Get bytecode by popping the queue
// Convert to packets or simply display to console
Bytecode
Bytecode
Queue <bytecode> : tdcs

Figure 4.2 . Illustration relation between Interface and Gateway

According to illustration on Figure 4.2 , tdcs queue acts as file system for Gannet Virtual Machine. It stores numbers of bytecode and allows other class to access the bytecode from queue.

However, the queue is very different compare to the actual file system. A queue only allows data to be stored from the rear and accessed from the front. Meanwhile, a file system provides mechanisms to create, manipulate, store and retrieve data. Furthermore, from higher level perspective a file system offers organization and management information on storage medium, in this case RAM. Hence, in order to increase the performance of Gannet Virtual Machine on managing and organizing its data, the queue based file system should be replaced by a file system. For this purpose, a file system called Gannet File System has been built to be implemented on Gannet Virtual Machine.

4.3.1 Modification

The aim of this integration is to demonstrate that file system is working properly on Gannet Virtual Machine. The integration of GannetFS file system into Gannet system, replaces the temporary file system. The file system is implemented on Gannet architecture as module which can be accessed by all services. Hence, after receiving task description and data, instead of storing them onto queue, Gateway stores them onto GannetFS file system as shown figure below.

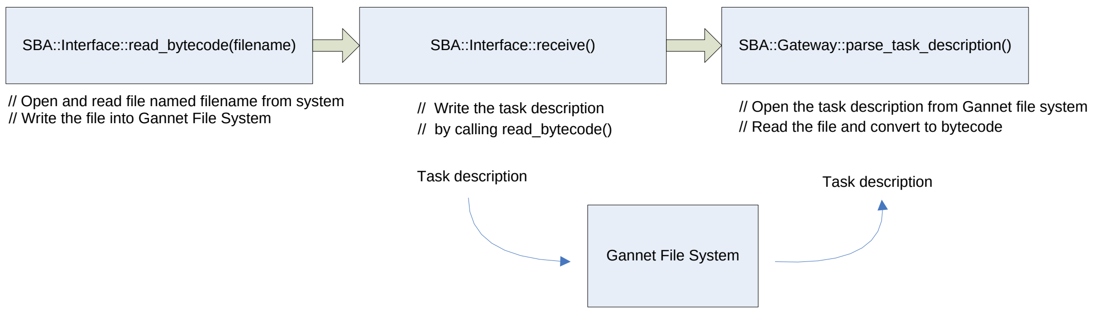
SBA::Interface::read_bytecode(filename)
SBA::Interface::receive()
SBA::Gateway::parse_task_description()
// Open and read file named filename from system
// Write the file into Gannet File System
// Write the task description
// by calling read_bytecode()
// Open the task description from Gannet file system
// Read the file and convert to bytecode
Task description
Gannet File System
Task description

Figure 4.3 Illustration Interface and Gateway Class after Gannet File system is integrated

Testing and Evaluation

5.1 Testing

After implementation phase has been done, the system must be verified to ensure that the system meets its specification. To verify GannetFS file system, Software testing approach were chosen for system checking and analysis.

5.1.1 Software Testing

Software testing is a dynamic technique of verification to ensure the quality and performance of the system. It examines the output of the system and its behavior to check that the system works as expected. In this project, two different type of testing were performed.

The first one is defect testing. It is intended to reveal defect in the system so that inconsistency between the system and specification could be encountered. Unit testing, which is subset of defect testing, was carried out during the implementation phase. It provides a way to verify the individual units of code works properly. All tests are written on executable code where the input and the expected output are specified and automatically checked. UnitTest++ was implemented as automated unit testing during implementation and integration phase.

The second one is validation testing. The purpose of this type of testing is to demonstrate that the system meets its requirements. GannetFS file system used set of test cases to ensure it works properly. Several test cases were derived from POSIX file system API test bench which focus on system stress testing. While other test cases were designed to ensure the system works properly on the real operational environment.

Test Case	Description	Result
Test00	Creates, writes to, verifies and then deletes a 6000-byte file. This test fails if the write and read function calls fail. This test also fails if the data written to the file does not match the data read from the file.	**Passed**
Test01	Creates, writes to, verifies, and then deletes a 2048-byte file. This test fails if the mkdir, write, or seek function calls fail. This test also fails if the data written to the file does not match the data read from the file.	**Passed**
Test02	Creates, copies, and then deletes a file. This test fails if the mkdir, creat, cp, or remove function calls fail. This test also fails if the data written to the file does not match the data read from the file.	**Passed**
Test03	Creates, moves, and then deletes a file. This test fails if the mkdir, creat, cp, or remove function calls fail. This test also fails if the data written to the file does not match the data read from the file.	**Passed**
Test04	Creates 17 empty files in a directory. Write the data on those files. This test fails if it cannot create 16 empty files in the directory. This test also fails if the file system can open 17 files at the same time.	**Passed**
Test05	Creates single directories with names that have a lengths varying from MAX_NAME_PATH-5 to MAX_NAME_PATH+5 characters. This test fails if a directory with a name longer than MAX_NAME_PATH characters is successfully created. The test also fails if it cannot create a directory with a name that contains	**Passed**

	MAX_NAME_PATH or fewer characters.	
Test06	Attempts to create a file in a directory that does not exist. This test fails if it successfully creates the file.	**Passed**
Test07	Creates subdirectories within a directory and then creates one file in each subdirectory until 6 level sub dir. This test fails if it cannot create a subdirectory or file.	**Passed**
Test08	Attempts to create a file that has the same name as an existing directory and has the same name but different character. This test fails if it successfully creates a file with the same name as an existing directory. This test also fails if it fails to create new file with the same name but different character.	**Passed**
Test09	Attempts to create a directory that has the same name as an existing file has the same name but different character. This test fails if it successfully creates a directory with the same name as an existing file. This test also fails if it fails to create new directory with the same name but different character.	**Passed**
Test10	Writes data to and reads data to multiple files from the outside using the lseek functions. This test fails if the data written does not match the data read.	**Passed**

Table 5.1 Test Cases

Table shown above gives an overview regarding the test cases that have been done during the testing. The detailed test cases is listed on Appendix C.

5.1.2 Integration Testing

After GannetFS passed software testing, it gained the confidence level that GannetFS has met the requirement and it works properly. The next phase is integration which is a process to implement GannetFS file system into Gannet Virtual Machine. Integration testing checks that GannetFS actually works together with Gannet Virtual Machine. As discussed on previous chapter, integration of GannetFS does not add new feature to Gannet Virtual Machine, but it provides alternative storage mechanism which is complementary with the old system. Furthermore, C++ #ifdef, else, and endif# technique was applied during the integration to localized the changes.

The testing for integration is quite straight forward. The integration only changes the way Gannet Virtual Machine stores the data and has no affect on output of the system. Hence, the test can be declared as success if the output of the system between Gannet Virtual Machine with GannetFS integrated and Gannet Virtual Machine without GannetFS is identical.

5.2 Evaluation

Unlike testing which is aimed to find the defect, evaluation concerns on find any problems interacting with the system. However, it is also can be used to check whether the system has been met the expectation. As this project does not have user interface to be evaluated, the evaluation focused on the performance of the system. There is three aspects to be examined on this sub section. First is Random Access Memory evaluation. Then it is followed by examining the Read Only Memory. The last sub section describes briefly about the performance of file system's operations.

5.2.1 Random Access Memory Evaluation

This section evaluates the utilization aspect of GannetFS file system. As GannetFS file system is configurable, its performance is different from one configuration to other configuration. Diagram below shows the difference performance.

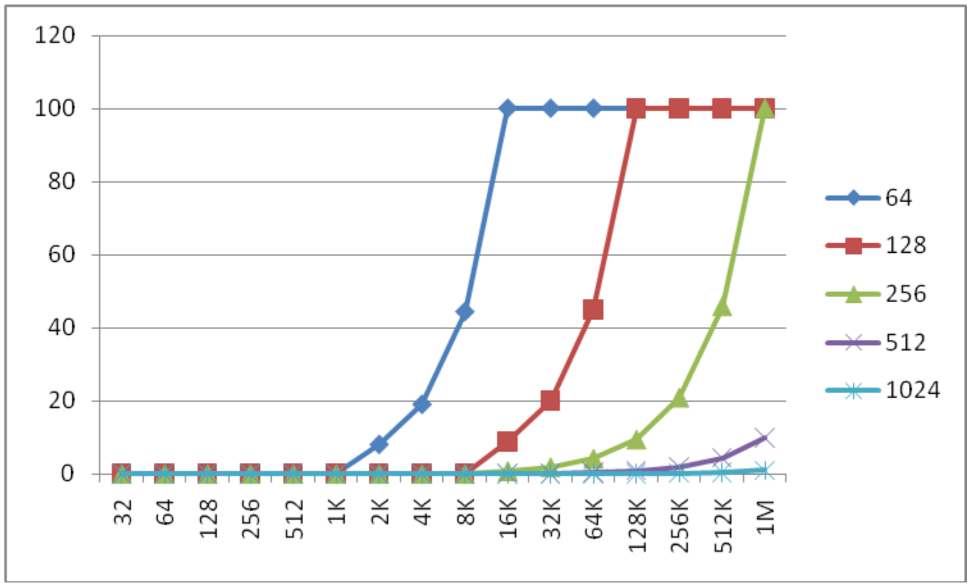

Figure 5.1 Block Allocation Overhead

Figure above shows the percentage of overhead process over different size of file. However, there is tradeoff between space efficiency and time efficiency. Having a large allocation unit tends to waste space as even a small file, a byte of file for instance, will consume a single unit. However, the larger its size, the faster time required to access each block as there is few blocks to proceed.

On the other side, having a small allocation unit means that a file will have many blocks. Reading and writing operations quite slow as accessing each block requires a processing time. Especially for writing operation, as there is a high overhead to allocate blocks before writing data to the blocks.

The size of inode also has influence to the performance of read and write operations. As discussed on section 2.1 , GannetFS file system is configurable. If the size of inode is increased, the size of array of direct block on inode also increases. Hence, inode can maintain more direct data blocks. This condition decreases the overhead which resulted by indirect block.

5.2.2 Read Only Memory Evaluation

In order to implement GannetFS to a system, system designer needs to know the space required for GannetFS. So that, an amount of memory can be reserved to implement GannetFS file system. Based on its layout, data structures of GannetFS can be calculated as follows.

	(Size)		(Type)		(Amount)		total	
Size of inode stack:	1	byte	(char)	X	16	=	16	byte
Size of block stack:	4	byte	(int)	X	112	=	448	byte
Size of FilDesc stack:	1	byte	(char)	X	16	=	16	byte
Array of data block:	64	byte	(block)	X	128	=	8192	byte
FilDesc table :	12	byte	(stat)	X	16	=	192	byte
							8864	byte

At default configuration, GannetFS has size as 8864 bytes. Furthermore, system designer also needs to have information regarding the size of compiled source code of GannetFS. Following table shows size of compiled GannetFS files system code without operation calls.

Compiled file	File size
FileSystem.o	19244 byte
GannetFileSystem.o	10088 byte

Table 5.2 Size of compiled code

After GannetFS has been implemented on a system, the actual needed space is revealed by observing the difference size between Gannet Virtual Machine with GannetFS integrated and Gannet Virtual Machine without GannetFS integrated.

	Size of GannetVM
GannetFS integrated	240.639 byte
GannetFS not integrated	234.942 byte

Table 5.3 Size of GannetVM with and without GannetFS

5.2.3 Performance Evaluation

Performance testing is carried out to observe system behavior under a certain workload. The purpose of this test is to compare and benchmark a system with others to find which performs better. To have ideal comparison between GannetFS and others file system, a benchmark environment is required which needs to have identical specification machine to carry out benchmark testing so that a fair time measurement could be obtained. Since the prerequisite is not feasible on this project, I decided to use another approach which measure the cycles of cpu instead of time measurement.

This method records how many cycle required performing an operation. However, the cycle measurement is not accurate as it returns different value for identical operation. Hence, sampling method is required to increase the accuracy of cycle measurement. On this project, five samples per operation were taken then those samples were calculated to get the mid values. The following tables show the cycle measurement of GannetFS.

Operation	**Cycles**
Creat	6722
Open (File exist)	14827
Close	1513
remove	6150
mkdir	4805
rmdir	5062
opendir	1882
readdir	1665
closedir	1807
cp (1216 byte)	20214
mw (1216 byte)	31074

Table 5.4 (a) Result of Test Performance

The first table shows operation that its performance is not affected by the size of file. However, there is anomaly on creat and open operations. Creat operation intends to create a new file and put stat structure of associate file on file descriptor table. While open allows creating new file as creat operation do and also allows putting an existing file into file descriptor table. In this performance test, the last operation was used which is only put a file into the file descriptor table. It was expected that open function should perform faster than creat function. However, the result shows that open operation took more time than create operation. Those results were different as expectation. As investigation took place, it was found that there was a flaw on process copying inode from translator to stat structure before putting it on file descriptor table.

File size	Read Cycles	Write Cycles
64 byte	2054	2933
128 byte	2434	3458
192 byte	2527	3506
256 byte	2987	3952
320 byte	3033	4662
1216 byte	6152	9859
1280 byte	6772	24733
2240 byte	11398	30618

Table 5.4 (b). Result of Test Performance

Second table shows results of read and write process over different size of file. There is no anomaly on these results. Since writing operation performs block allocation for each blocks before writing the data in, it took more cycles than reading operation. Furthermore, the results show that accessing data on indirect block takes more cycles than accessing data from direct data block. The maximum data size for direct, single indirect and double indirect blocks is 192, 1216 and 16356 byte, respectively. The result shows that the cycles increase significantly when GannetFS writing file from 192 byte to 256 byte. This is caused by overhead on single indirect block allocation. File with size 192 byte is written on direct data block while 256 byte file is written on single indirect data block. The more significant increment is when GannetFS transferring allocation block from single indirect data block to double indirect data blocks.

Conclusion

This project was undertaken to design and implement a file system for Gannet Virtual Machine. Different to other file systems which are implemented on top of kernel, GannetFS was designed to operate without rely on a kernel since Gannet Virtual Machine does not have kernel within it. Therefore, a mechanism to have direct access to data storage was required as a part of GannetFS file system.

Since GannetFS file system operates on embedded environment, where the constraint is quite tight, the time and space usage efficiency are main concerns on this project. Moreover, as embedded system, Gannet Virtual Machine connected to different type of platforms. Hence, GannetFS file system has been designed and implemented with fully POSIX compliant.

Due to time constraint on project development, current project has designed and implemented GannetFS file system without considering aspect on security, consistency and reliability. Further work will need to improve current GannetFS file system with enhancement on security, consistency and reliability aspect. Implementation of permission and privilege aspect on GannetFS file system should be carried out for future work.

This project has implemented a GannetFS file system for current Gannet Virtual Machine to allow Gannet Virtual Machine manages and organizes data. This implementation makes a significant progress to Gannet Virtual Machine that is aimed to emerge as a new Operating System.

References

Card, R., Ts'o, T., Tweedie, S. 1994. Design and implementation of the second extended
File system. In Proceedings of the 1st Dutch International Symposium on Linux.

Edel, N.K., Miller, E.L., Brandt,K.S., Brandt, S.A. 2003. Measuring the Compressibility of Metadata and Small Files for Disk/NVRAM Hybrid Storage Systems. *Technical Report UCSC-CRL-03-04.*

Giampaolo, D. 1999. Practical File System Design: The Be File System. *Morgan Kaufmann Publisher, Inc.*

Fagin, R.,Nievergelt, J., Pippenger,N., Strong,H.R. 1979. Extensible hashing—A fast access method for dynamic files. *ACM Transaction Database System.*

Heath, S. 2003. Embedded System Design, Second Edition. Newnes.

Kogel, T et al.,2006. Integrated System-Level Modeling of Network-on-Chip enabled Multi-Processor Platforms. Springer.

Lewine, Donald. A., 1991. POSIX Programmer's Guide: Writing Portable UNIX Programs with the POSIX.1 Standard. California: O'Reilly.

McKusick, M. K., Joy, W. N., Leffler, S. J., Fabry, R. S. 1984. A fast file system for UNIX. *ACM Transaction on Computing System*.

McKusick, M.K., Bostic, K., Karels, M.J.,Quarterman, J.S. 1996. The Design and Implementation of the 4.4BSD Operation System. *Addison-Wesley Longman, Inc.*

Miller,E. L.,Brandt, S. A., AND Long,D.D.E. 2001. HerMES: High-performance reliable MRAM enabled storage. *In Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems.* Schloss Elmau, Germany.

Ousterhout. J.K., Da Costa, H., Harrison, D., Kunze, J.A., Kupfer, M., Thompson, J.G. 1985. A Trace-Driven Analysis of the Unix 4.2 BSD File System. *Proceedings of the 10th Symposium on Operating System Principles.*

Pasrcha, S., Dutt, N., 2008. On-Chip Communication Architectures: System on Chip Interconnet. Morgan Kaufman: California.

Satyanarayanan, M.1981. A Study of File Sizes and Functional Lifetimes. *Proceedings of the 8th Symposium on Operating System Principles.*

Silberschats, A., Galvin, P.B., Gagne, Greg. 2002. Operating System Concepts. John Willey & Sons, inc.

Snyder, Peter. 1992. Tmpfs: A Virtual Memory File System. White Paper. Sun Microsystems inc.

Sweeney, A., Doucette, D., Hu, W., Anderson, C., Nishimoto, M., Peck, G. 1996. Scalability in the XFS file system. *In Proceedings of the USENIX Annual Technical Conference.* San Digeo, CA.

Tanenbaum, A.S., Woodhull,A.S. 1997.Operating Systems: Design and Implementation (2nd Edition). *Prentice Hall.*

Tanenbaum, A.S. 2006. Operating Systems: Design and Implementation (3rd Edition). Prentice Hall.

Thompson, K. 1978. Unix Implementation. *Bell System Technology.*

Vanderbauwhede, W.,2006a.Gannet: a functional task description language for a service-based soc architecture. In Proceesings of the Symposium on Trends in Functional Programming 2006.

Vanderbauwhede, W., 2006b The Gannet Service-based SoC: A Service-level Reconfigurable Architecture. In Proceedings of 1st NASA/ESA Conference on Adaptive Hardware and Systems (AHS-2006), pp 255–261.

Vanderbauwhede, W., 2007a. Separation of Data flow and Control flow in Reconfigurable Multi-core SoCs using the Gannet Service-based Architecture. In Adaptive Hardware and Systems, 2007. AHS 2007. Second NASA/ESA Conference on, pp 326–333.

Vanderbauwhede, W., 2007b. Gannet: a Scheme for Task-level Reconfiguration of Service-based Systems-on-Chip. In Scheme and Functional Programming, SFP2007. Eigth ACM/SIGPLAN Workshop on, pp 129–137.

Wang, A.A., Kuenning, G., Reiher, P., Popek, G. 2006. The Conquest File System: Better Performance Through a Disk/Persisten-RAM Hybrid Design. *ACM Transaction on Storage.*

Wang, X., 2008. Development and Implementation of a TCP/IP Stream Socket Interface for Gannet Virtual Machine. Master Dissertation.

APPENDICES

A. Requirement Specification

Functional Requirement

- The file system must operate in user space without rely on kernel.
- The file system must be able to create new file.
- The file system must be able to read a file.
- The file system must be able to write a file.
- The file system must be able to remove a file.
- The file system must be able to create new directory.
- The file system must be able to remove a directory.
- The file system should support random access on a file.
- The file system should support streaming feature.
- The file system should be able to perform copy operation.
- The file system should be able to perform move operation.

Non-Functional Requirement

- The file system's interface must comply to POSIX standard.
- The file system must be developed and implemented using C/C++ programming language.

B. File System API

Application Program Interfaces listed below, taken from POSIX standard interface (Lewine 1991).

1. open() : Opens a file if the file exists or creates a new file does not exist.

Arguments:

int open(const char *path, mode_t oflag);

Path	: Pointer to path of the file to open.
oflag	: Symbolic flags (Permission bits if a file is created)

Returns:

A file descriptor or -1 on error.

Description:

The open() function establishes a connection between a file and a file descriptor. The file descriptor is a small integer that is used by I/O functions to reference the file. The path argument points to the pathname for the file. The oflag argument is the bitwise inclusive OR of the values of symbolic constants. The programmer must specify exactly one of the following three symbols:

O_RDONLY	: Open for reading only.
O_WRONLY	: Open for writing only.

2. creat()—Creates a new file or rewrites an existing one.

Arguments:

int creat(const char *path, mode_t mode);

path	: Pointer to path of the file to be created.
mode	: Permission bits for the new file.

Returns:

A file descriptor or -1 on error.

Description:

It opens a file for writing. If the file does not exist, it is created with the permission bits set from mode and the owner and group IDs are set from the effective user and group ID of the calling process. The file descriptor returned by creat() may be used only for writing.

3. read() : Reads from a file.

Arguments:

size_t read(int fildes, void *buf, unsigned int nbyte);

fildes	:File descriptor open for reading.
buf	: Pointer to the place where the data should be read.

nbyte	: Maximum number of bytes to read.

Returns:
The number of bytes actually read or -1 on error.
Description:
The read() function reads nbyte bytes from the file associated with fildes into the buffer pointed to by buf. The read() function returns the number of bytes actually read and placed in the buffer. This will be less than nbyte if t⍰⍰he number of bytes left in the file is less than nbyte.

4. write() : Writes to a file.
Arguments:
ssize_t write(int fildes, const void *buf, unsigned int nbyte);

fildes	: File descriptor open for writing.
buf	: Pointer to the data to be written.
nbyte	: Number of bytes to write.

Returns:
The number of bytes written, or -1 to indicate an error.
Description:
The write() function writes nbyte bytes from the array pointed to by buf into the file associated with fildes. If nbyte is zero and the file is a regular file, the write() function returns zero and has no other effect.

The write() function returns the number of bytes written. This number will be less than nbyte if there is an error. It will never be greater than nbyte.

5. close() : Closes a file.
Arguments:
int close(int fildes);

fildes : The file descriptor to close.
Returns:
Zero on success and -1 on failure.
Description:
The close() function deallocates the file descriptor named by fildes and makes it available for reuse.

6. lseek() : Repositions read/write file offset.
Arguments:
off_t lseek(int fildes, off_t offset, int whence);

fildes	: File descriptor to be repositioned.
offset	: New offset.

whence : One of the following codes
Returns:
The new offset. In case of error, ((off_t)-1) is returned.
Description:
The lseek() function sets the file offset for the file description associated with fildes as follows:

- SEEK_SET Set offset to offset.
- SEEK_CUR Add offset to current position.
- SEEK_END Add offset to current file size.

7. mkdir() : Makes a directory.
Arguments:
int mkdir(const char *path, mode_t mode);

path : Pointer to name of the directory to create.
mode : Directory permission bits.
Returns:
Zero on success and -1 on failure.
If an error occurs, a code is stored in errno to identify the error.
Description:
The mkdir() function creates a new directory named path. The permission bits are set from mode.

8. rmdir() : Removes a directory.
Arguments:
int rmdir(const char *path);

path : Pointer to the path name of the directory to remove.
Returns:
Zero on success and -1 on failure.
Description:
It removes an exist directory.

9. opendir() : Opens a directory.
Arguments:
DIR *opendir(const char *dirname);

dirname :Pointer to the name of the directory to read.
Returns:
A pointer for use with readdir() and closedir() or, if an error took place, NULL.
Description:
The opendir() function opens a directory stream corresponding to the directory named in the dirname argument. The stream is positioned at the first entry.

10. readdir() : Reads a directory.

Arguments:

struct dirent *readdir(DIR *dirp);

dirp Pointer returned by opendir().

Returns:

A pointer to an object of type structure dirent or, in case of error, NULL.

Description:

The readdir() function returns a pointer to a structure dirent representing the next directory entry from the directory stream pointed to by dirp. On end-of-file, NULL is returned.

11. rewinddir() : Resets the readdir() pointer.

Arguments:

void rewinddir(DIR *dirp);

dirp Pointer returned by opendir().

Returns:

No value is returned.

Description:

The rewinddir() function resets the position associated with the directory stream pointed to by dirp. It also causes the directory stream to refer to the current state of the directory.

12. closedir(): Ends directory read operation.

Arguments:

int closedir(DIR *dirp);

dirp : Pointer returned by opendir().

Returns:

Zero on success and -1 on failure.

Description:

The directory stream associated with dirp is closed. The value in dirp may not be usable after a call to closedir().

13. remove() : Removes a file from a directory.

Arguments:

int remove(const char *filename);

filename : Pointer to filename to delete.

Returns:

Zero on success and non-zero on failure.
Description:
The remove() function comes from Standard C and has the same effect as unlink(); namely, the string pointed to by filename can no longer be used to access a file. Use the rmdir() function to delete directories.

14. stat() : Gets information about a file.
Arguments:
int stat(const char *path, struct stat *buf);

path	: Path name of file to research.
buf	: Pointer to an object of type struct stat where the file information will be written.

Returns:
Zero on success and -1 on failure.
Description:
The path argument points to a pathname for a file and requires all directories listed in path must be searchable. The stat() function obtains information about the named file and writes it to the area pointed to by buf.

15. unlink() : Removes a directory entry.
Arguments:
int unlink(const char *path);

path	: Pointer to path name of file to delete.

Returns:
Zero on success and -1 on failure.
Description:
The unlink() function removes the link named by path and decrements the link count of the file referenced by the link.

C. Testing

TestCase00

Creates, writes to, verifies and then deletes a 6000-byte file. This test fails if the write and read function calls fail. This test also fails if the data written to the file does not match the data read from the file.

1. Reads a file named “testing.txt” from disk drive using C++ std library. The data is stored on a first buffer.
2. Creates a file named “head-gfs.txt” on GannetFS file system.
3. Write the data from first buffer to “head-gfs.txt” file.
4. Closes “head-gfs.txt” file.
5. Reopens “head-gfs.txt” file.
6. Reads the file and stores the data on second buffer.
7. Verifies the first and second buffers using memcmp() function.
8. Write the data from second buffer to disk drive named “buffout_big.txt” using C++ std library.
9. Removes “head-gfs.txt” file from GannetFS file system.
10. Compares “testing.txt” with “buffout_big.txt” using diff function.

TestCase01

Creates, writes to, verifies, and then deletes a 2048-byte file. This test fails if the **mkdir**, **write**, or **seek** function calls fail. This test also fails if the data written to the file does not match the data read from the file.

1. Reads a file named “testing.txt” from disk drive using C++ std library. The data is stored on a first buffer.
2. Creates a directory named “/test-one”.
3. Creates a file named “/test-one/testFile-one” on GannetFS file system.
4. Write the data from first buffer to “/test-one/testFile-one” file.
5. Closes “/test-one/testFile-one” file.
6. Reopens “/test-one/testFile-one” file.
7. Reads the file and stores the data on second buffer.
8. Verifies the first and second buffers using memcmp() function.
9. Write the data from second buffer to disk drive named “buffout_big.txt” using C++ std library.
10. Removes “/test-one/testFile-one” file from GannetFS file system.
11. Removes directory named “/test-one”.

TestCase02

Creates, copies, and then deletes a file. This test fails if the **mkdir**, **creat**, **cp**, or **remove** function calls fail. This test also fails if the data written to the file does not match the data read from the file.

1. Reads a file named "head.txt" from disk drive using C++ std library. The data is stored on a first buffer.
2. Creates a directory named "/test-two".
3. Creates a file named "/test-two/testFile-two" on GannetFS file system.
4. Write the data from first buffer to "/test-two/testFile-two" file.
5. Closes "/test-two/testFile-two" file.
6. Copies "/test-two/testFile-two" to "/test-two/testFile-copy"
7. Opens "/test-two/testFile-copy" file
8. Reads "/test-two/testFile-copy" and store the data on second buffer
9. Verifies the first and second buffers using memcmp() function.
10. Write the data from second buffer to disk drive named "buffout_big_2.txt" using C++ std library.
11. Removes "/test-two/testFile-two" file from GannetFS file system.
12. Removes "/test-two/testFile-copy" files from GannetFS file system.
13. Removes directory named "/test-two".

TestCase03

Creates, moves, and then deletes a file. This test fails if the **mkdir**, **creat**, **mv**, or **remove** function calls fail. This test also fails if the data written to the file does not match the data read from the file.

1. Reads a file named "head.txt" from disk drive using C++ std library. The data is stored on a first buffer.
2. Creates a directory named "/test-three".
3. Creates a file named "/test-three/testFile-three" on GannetFS file system.
4. Write the data from first buffer to "/test-three/testFile-three" file.
5. Closes "/test-three/testFile-three" file.
6. moves "/test-three/testFile-three" to "/test-three/testFile-moved"
7. Opens "/test-three/testFile-moved" file
8. Reads "/test-three/testFile-moved" and store the data on second buffer
9. Verifies the first and second buffers using memcmp() function.
10. Write the data from second buffer to disk drive named "buffout_big_2.txt" using C++ std library.
11. Removes "/test-three/testFile-moved" files from GannetFS file system.
12. Removes directory named "/test-three".

TestCase04

This test case creates 17 empty files in a directory. Write the data on those files. This test fails if it cannot create 16 empty files in the directory. This test also fails if the file system can open 17 files at the same time.

1. Reads a file named "-test.breadtxt" from disk drive using C++ std library. The data is stored on a first buffer.
2. Creates a directory named "/test-four".
3. Creates 17 files under directory "/test-four"..
4. Writes data from first buffer to those files.
5. Closes and removes those files.

TestCase05

This test case creates single directories with names that have a lengths varying from MAX_NAME_PATH-5 to MAX_NAME_PATH+5 characters. This test fails if a directory with a name longer than MAX_NAME_PATH characters is successfully created. The test also fails if it cannot create a directory with a name that contains MAX_NAME_PATH or fewer characters.

1. Creates a directory named "test-five".
2. Creates a directory under "test-five" named "abcdefghijkelmnopq"
3. Creates a directory under "test-five" named "abcdefghijkelmnopqrs"
4. Creates a directory under "test-five" named "abcdefghijkelmnopqrst"
5. Creates a directory under "test-five" named "abcdefghijkelmnopqrstuv"
6. Creates a directory under "test-five" named "abcdefghijkelmnopqrstuvwx"
7. Performs listdir for directory "test-five".

TestCase06

This test attempts to create a file in a directory that does not exist. This test fails if it successfully creates the file.

1. Creates a directory named "test-six".
2. Creates a file named "test-six/file"
3. Creates a file named "test-six/file1"
4. Creates a file named "test-six/file2"
5. Creates a file named "test-six-1/file3"
6. Creates a file named "test-siz/file4"
7. Creates a file named "test-siy/file5"
8. Closes those files.
9. Performs listdir for directory "test-six".

TestCase07

Creates subdirectories within a directory and then creates one file in each subdirectory until 6 level sub dir. This test fails if it cannot create a subdirectory or file.

1. Creates a directory named “test-seven”.
2. Creates a file named “File” under directory “test-seven”.
3. Creates a directory named “test-seven-a” under directory “test-seven”.
4. Creates a file named “aFile” under directory “test-seven-a”.
5. Creates a directory named “test-seven-b” under directory “test-seven-a”.
6. Creates a file named “bFile” under directory “test-seven-b”.
7. Closes all files.

TestCase08

This test case attempts to create a file that has the same name as an existing directory and has the same name but different character. This test fails if it successfully creates a file with the same name as an existing directory. This test also fails if it fails to create new file with the same name but different character.

1. Creates a directory named “existingDir”.
2. Creates a directory named “existingDir”.
3. Creates a directory named “existingdir”.

TestCase09

This test case attempts to create a directory that has the same name as an existing file has the same name but different character. This test fails if it successfully creates a directory with the same name as an existing file. This test also fails if it fails to create new directory with the same name but different character.

1. Creates a file named “existingFile”.
2. Creates a file named “existingFile”.
3. Creates a file named “existingfile”.

TestCase10

Writes data to and reads data to multiple files from the outside using the **seek** functions. This test fails if the data written does not match the data read.

1. Reads a file named “byte.txt” from disk drive using C++ std library. The data is stored on a first buffer.
2. Reads a file named “byte-1.txt” from disk drive using C++ std library. The data is stored on a second buffer.
3. Reads a file named “byte-2.txt” from disk drive using C++ std library. The data is stored on a third buffer.
4. Creates a file named “dummy-1-gfs10.txt” on GannetFS file system.
5. Write the data from second buffer to “dummy-1-gfs10.txt” file.
6. Closes “dummy-1-gfs10.txt” file.
7. Opens file named “dummy-1-gfs10.txt”.
8. Seeks to the end of file.
9. Writes the data from third buffer to “dummy-1-gfs10.txt” file.
10. Seeks to the beginning of file.
11. Reads the “dummy-1-gfs10.txt” and stores the data to the fourth buffer.
12. Verifies the first and the fourth buffers using memcmp() function.
13. Write the data from second buffer to disk drive named “buffout.txt” using C++ std library.
14. Compares “testing.txt” with “buffout_big.txt” using diff function.