A Lean USB File System for Bare Machine Applications

Songjie Liang
Department of Computer and Information Sciences
Towson University
Towson, Maryland, USA
jeffliang1@gmail.com

Ramesh K. Karne
Department of Computer and Information Sciences
Towson University
Towson, Maryland, USA
rkarne@towson.edu

Alexander L. Wijesinha
Department of Computer and Information Sciences
Towson University
Towson, Maryland, USA
awijesinha@towson.edu

Abstract

USB file management systems for mass storage are supported on many platforms and commonly used today by a variety of applications. These file systems vary depending upon the underlying operating system and environment. We consider the design of a USB file system that is independent of any operating system or application environment. Specifically, we describe the design and development of a lean USB file system for supporting bare PC applications that run with no OS or kernel support of any kind, and give details of its architecture, design, and implementation. Our design leverages the design characteristics and inherent features of USB mass storage. Bare machine USB file management systems enable computer applications to access mass storage at run-time without the operating system and environmental dependencies of conventional file management systems.

Keywords – File System, FAT32, USB, Bare Machine Computing, Application Object.

1 INTRODUCTION

File systems provide higher level abstractions for use in building computer applications. They are typically closely tied to the underlying operating system (OS). File systems are complex and depend upon mass storage technology and the services provided by the OS. We address the question of how to make the popular USB file system independent of its OS environment. Such a file system can then be used by computer applications that run on a bare machine with no underlying OS or kernel. It can also serve as a possible first step towards developing customized file systems for supporting secure applications that cannot be readily exploited by conventional attacks. Our approach takes advantage of some inherent features of a USB system such as its similarity to a memory system enabling it to be addressed as a linear block device. However, designing and developing such file systems for a bare machine is different from USB mass storage management for an embedded system, which requires some form of embedded kernel or lean OS.

We provide details of a lean USB file system for bare machines. We discuss several related architectural issues and then describe the design of the bare file system, including disk map, root directory, sequencing operations, tasks, and user interface. The file system runs on an Intel x86 based bare PC. For comparison purposes, essentially the same system, but with minimal system calls, was implemented on a Linux OS.

The rest of the paper is organized as follows. Section II considers related work, and Section III describes the lean file management system. Section IV discusses architecture and issues, Section V narrates some design details, Section VI shows implementation, Section VII outlines novel features of our approach, Section VIII deals with functional operation and testing, and Section IX presents the conclusion.

2 RELATED WORK

We implemented a lean file system on a bare PC with no kernel or OS running on the machine. Bare PC applications [6] use the Bare Machine Computing (BMC) or dispersed OS computing paradigm [7], wherein self-supporting applications run on a bare PC. The application is written in C++ and runs as an application object (AO) [7] by using its own interfaces to the hardware [8] and device drivers. Using the BMC paradigm, we have built a variety of applications including Web servers, email servers, SIP servers and VoIP clients. For example, bare Web servers and clusters are described in [6,13].

While the BMC concept resembles approaches that reduce OS overhead and/or use lean kernels such as Exokernel [3], IO-Lite [11], Palacios and Kitten [9], there are significant differences such as the lack of centralized code in the form of a kernel to manage system resources. Recently, there has been considerable interest to use flash memory in mass storage devices. The Umbrella file system [5] demonstrates a versatile file system that uses a hard disk and flash memory. This system also illustrates how to integrate two different types of storage devices. Other research has dealt with adding cache systems at a driver level to gain performance improvements [1]. The design and implementation of a FAT32 file system for high performance clusters is discussed in [2]. As the capacity of flash drives continues to increase and their cost is reducing, they are expected to become an important component of future mass storage systems.

3 FILE MANAGEMENT SYSTEM

A lean file management system is designed and implemented to demonstrate our approach. We have built two functionally equivalent compatible file systems: one runs on the Linux OS and the other one runs on a bare PC. The code that runs on a Linux system uses minimal system calls. The code that runs on a bare PC uses no OS or kernel. In the bare file system, there are minimal file commands including create a directory, delete a directory, list a directory, create a file, delete a file, open a file, close a file, read a file, write to a file, and copy a file. These file commands are implemented in C/C++. The commands were tested on both the bare PC and Linux systems.

The file system is based on FAT32 [10]. Fig.1 shows the FAT32 disk map. In the bare file system, the master boot record (MBR) contains the bare boot code that enables the PC to boot from the USB. The USB needs to be properly partitioned and made bootable by installing the appropriate configuration on the device. A Linux based "**parted**" tool is used to install the appropriate partition on the USB device. The disk map including the contents in the MBR and the reserved sectors are different depending on the USB vendor. Thus, the bare file system needs to consider such variations when designing the bare file system application. In Fig.1, FAT1 is the file allocation table entry for the device, FAT2 is the backup for FAT1, the root directory structure has the 32 byte file records for each file entry, and the rest of the device is used to store data files. The bare file system designer needs to know details of the disk map and its individual records and fields.

In addition, bare machine applications also contain their own bare USB device driver. The bare USB driver is written based on the USB specification [12], enhancer host controller interface specification [4], and universal serial bus mass storage class specification [15]. The details of the bare USB driver are not described in this paper since our focus here is on the file management system. However, it is an essential part of the application object (AO) that contains the file system application.

Our lean USB file system can be used on a bare PC environment without any OS or kernel, and it is also usable in a Microsoft Windows environment. The version that runs on Linux can also be run on Windows. Using FAT32 as the basis for the file system enables it to be compatible with Microsoft Windows and other OSs. We now discuss the architecture, design and implementation of the bare file system.

4 ARCHITECTURE

The architecture of a file system that can run on a bare PC and is also compatible with the Microsoft Windows file system is based on the bare machine computing paradigm. This paradigm requires a unique style of programming that requires the programmer to be aware that a bare application must run without an OS/kernel or environment support. In this approach, an end user application is modeled as a single AO. It may consist of a set of applications a user may want to use at a given point in time. For example, a user may use a file system and upload a file through a Browser and also edit a file to cut and paste some data. All of the above tasks can be modeled as a single self controlled AO. AO is self-controlled, self-executed and self-managed. As there is no OS or kernel in the system, an AO programmer controls the operating environment including: boot, loader, static and dynamic memory, thread or process management, interrupts and I/O control. In particular, there is a single AO that implements the file management system. However, when this AO executes, it runs as a single thread of execution with no interrupts. The file management system consists of direct hardware interfaces to the USB driver, which are known as the bare hardware API. This hardware API is accessible to the AO programmer to design and implement the necessary file functions for a given application.

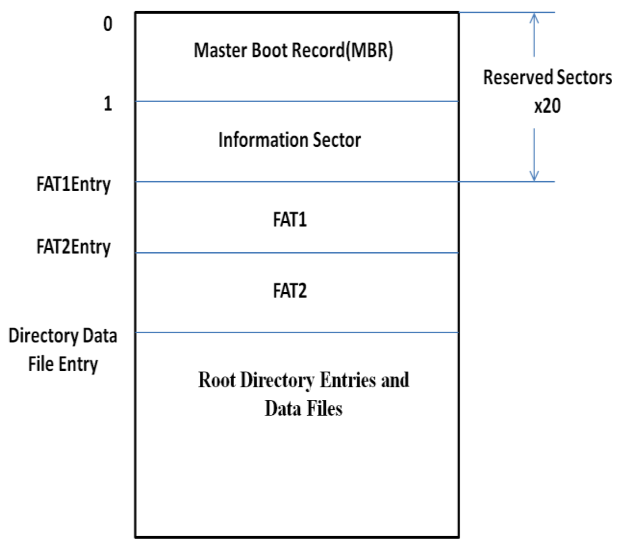

Figure 1. Bare File System Disk Map

For example in case of a bare Web server, files are stored on the USB and accessed using the bare hardware API. The Web server AO object can create a hash table with file names as key attributes and maintain file indexes through this hashing mechanism. The file management system hardware API described in this paper can thus be used by an AO programmer to support a bare PC application such as a bare Web server.

The file management system architecture is based on the disk map shown in Fig. 1 and the FAT32 specification and hardware API for the USB. The bare USB driver provides this hardware API to access the device directly within its C/C++ code. Several architectural issues that arise when building a bare file management system are described below.

4.1 Formatting

There are many file formats such as FAT16, FAT32 and NTFS. Each one provides different formats and requires a different organization with its own structures. Some OS-based file systems support many formats. The bare file system supports only the FAT32 format. Support of this single format enables the design and implementation of the bare file system to be simpler. Furthermore, it is also easier to build and demonstrate a working bare file system based on FAT32.

4.2 Boot Record

The boot record on the USB (0th sector) provides a variety of parameters that are needed to identify the file structure and its layout on the USB. In a bare PC without any OS or kernel, it is necessary to deal with variations in the boot record parameters. We followed the Microsoft Windows boot format in order to address this issue. Some parameters such as bytes per sector, reserved sectors, number of fats, total sectors, and fat size have to be read from the boot sector to locate the disk map as shown in Fig.1.

4.3 USB Vendors and Types

Each USB, depending on its size, version and vendor, provides a variety of options and storage configurations. The bare file system generalizes these options by identifying key parameters that are essential to the operation of mass storage devices. These parameters have to be read at boot time when a USB device is used for booting. Some BIOS (basic input output system) calls are used to identify the vendor and device type and its address.

4.4 USB Standard

The USB standard 2.0 is used to implement the bare file system and driver. This standard is very detailed (about 1000 pages) and continues to be updated. While the bare device driver is compatible with the standard, it only implements essential features in order to reduce complexity.

4.5 SCSI Commands

There are numerous small computer system interface (SCSI) commands that are provided in the standard. Each USB vendor implements only a subset of these commands. We have chosen a minimal subset of commands for implementation. Some of the commands implemented in our driver include: Test Unit Ready, Sense, Read (10), Write (10). The device driver internally implements these commands via USB commands. The commands are processed by the host controller in the bare machine.

4.6 USB Driver

Most USB drivers are developed for a given operating system or kernel. We have built a bare USB driver from scratch to make it independent of any OS or kernel environments.

4.7 USB Plug-and-Play

The plug-and-play features of the USB architecture require initialization and configuration/re-configuration of the USB every time it is inserted or removed. This requires special thread handling within the bare file management system.

4.8 USB Monitoring

USB operations require monitoring and polling of the USB with the Test Unit Ready command. The designer needs to decide how often this command needs to be done (it does not appear to be specified in the standard). We invoke this command before we do any read or write operation.

4.9 USB Operation and Sequencing

As the USB standard is complex and there are a large number of variables in the USB specification, it is challenging to build a bare USB driver and file system that runs without any OS or kernel. We used a Beagle Analyzer [14] and reverse-engineered the driver using a Windows analyzer trace.

4.10 Scalability

Several design issues arise when building a file system that can work with both small and large size USBs. To address varying USB capacities, it is necessary to make appropriate adjustments to cluster sizes and block sizes, and to the layout of the disk map. The current bare file system has only been tested with 2 GB USBs. This design needs to be enhanced in future to address the issue of scalability.

4.11 Enhanced FAT32 Requirements

The FAT32 standard allows large file names, grouping capabilities and clustered file systems. We have designed the lean bare system with only a minimal set of features. For example, the directory entry for a given file is limited to 32 bytes. The extension of a directory entry is not possible at present.

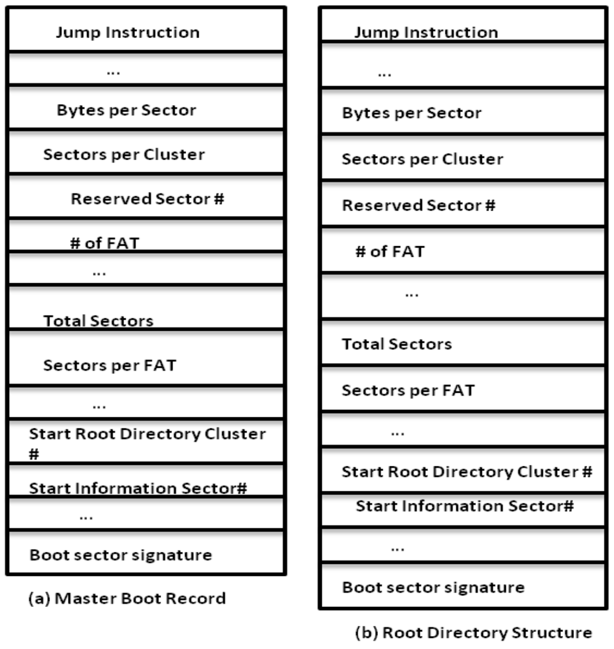

Figure 2. MBR/Root Directory Structure

5 DESIGN

The lean USB file management system for bare machines supports a given bare machine application. The key design elements of the system are discussed below.

5.1 Disk Map

Fig. 1 shows an example disk map layout for a USB of size 2 GB. During the initialization, a boot sector is read and its parameters as shown in Fig. 2a are parsed. Based on those parameter values, the entry points for FAT1, FAT2, root directory and data are calculated as shown below.

FAT1Entry = Number of Reserved Sectors

FAT2Entry = FAT1Entry + Sectors per FAT

DirectoryDataFileEntry = FAT1Entry +

Number of FATs * Sectors per FAT

Once these basic values are calculated, the starting point for user files can be computed. For each USB, the disk map layout may be different. Also, the disk map must be updated whenever the file system gets updated during USB operation. The master boot record contains the parameters needed to compute the disk map.

5.2 Root Directory

The root directory structure consists of a 32-byte data structure for each file resident on the mass storage USB device. When the size of the file system increases, it may consist of several linked data structures. Fig. 2b shows the structure of a root directory and its fields. Many of these fields require dynamic updates during operation. In this case, caching techniques are needed to improve performance.

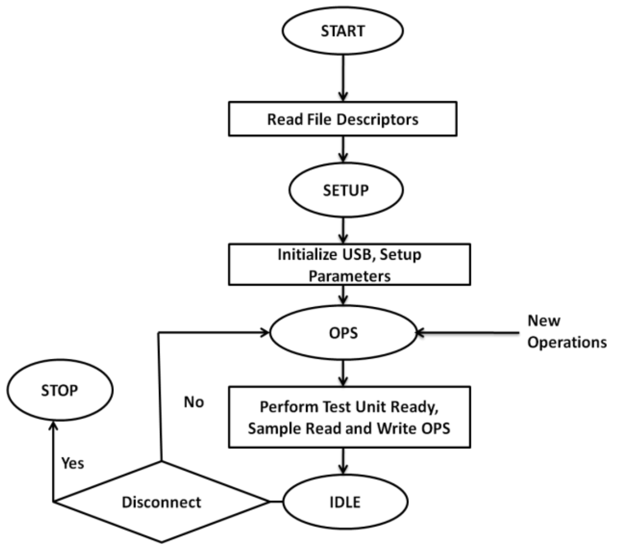
START
Read File Descriptors
SETUP
Initialize USB, Setup Parameters
OPS
New Operations
STOP
No
Perform Test Unit Ready, Sample Read and Write OPS
Yes
Disconnect
IDLE

Figure 3. Control and operations

5.3 Plug-and-play

The plug-and-play nature of the USB mass storage device requires that it have special handling capabilities. For instance, when the USB device is plugged, it needs to be immediately detected, initialized, configured and setup for operation. Similarly, when the device is detached, it needs to be disconnected and appropriate store-back techniques are needed.

5.4 Control

Fig. 3 shows the operation sequence to control a USB device. Every time a USB is plugged-in, numerous descriptors are read and their relevant information is stored by the system. These descriptors such as device, configuration, interface and endpoint are essential USB entities for controlling the device. The mass storage USB device also has the characteristics of a bulk-transfer device [15]. For example, bulk transfer devices use IN and OUT endpoints to communicate between the driver and the controller. The bulk transfer uses SCSI commands that are encapsulated in USB commands as specified in [15].

5.5 USB Operations

There are many USB low-level device operations described in the standard [12]. Some operations such as set address, clear feature, test unit ready, sense, read, and write are implemented in the driver. Such operations are invoked using high-level calls from the file system. The mass storage class document [15] specifies internal device operation details that are needed to design a USB driver. However, designing a bare USB driver is different from designing a conventional driver since it has to work as part of a bare application that runs with no OS support.

5.6 Bare PC System

Previous work has dealt with the design and implementation of bare PC applications and systems including [6, 7, 8, 13]. The lean USB bare file management system can be integrated with many bare PC applications. Since the bare file system is independent of any conventional OS or kernel, it is designed to be part of the bare PC application that it supports. For example as noted previously, a bare PC Web server can have its own file system as part of the server application code that runs in user mode. In essence, a given bare application is intertwined with the bare file system itself i.e., there is no special component that implements a bare file system independently of its application.

5.7 Tasks

Bare PC systems have a task structure that determines how execution of a bare application is managed. This task structure is also part of the bare application program. The present file system includes two types of tasks: the Main task and USB file tasks. The Main task, which is present in all bare applications, runs continually whenever no other task is running. It gives control to the USB file task as needed, and the latter returns control to the Main task when a given operation is complete. In a system with n USB ports, there will be n USB tasks i.e., each port has its own task to manage its operation. The Main task also detects the plug-and-play actions of each port. The simplicity of the task structure enables it to be easily customized for a given application. The bare PC application programmer controls execution flow and task management from within the application itself.

5.8 Bare PC USB Driver

The bare PC USB driver implements low-level operations as noted in item 5 above. We do not discuss details of driver design and implementation since they are not within the scope of this paper.

5.9 User API

File system commands implemented in our system are mentioned in Section 3. Commands such as Open(), Read() and Write() use low-level USB driver commands such as Test Unit Ready(), ReadOp(), and WriteOp(). These USB driver commands can also be directly accessed by bare application programmers to build their own file management applications. This provides a complete control of file system and USB driver control to an AO programmer. All file and driver commands run in user mode. These commands can be easily adapted to work with other pervasive USB devices.

5.10 Integrated application

As noted earlier, each bare PC application is a single entity that integrates all of its components within the application itself. Since the bare file system is also integrated into a given application, control of the file system all the way down to low-level USB commands is completely controlled by the code written by the bare PC application programmer.

6 IMPLEMENTION

The implementation of the lean bare file management system is done using C/C++. Due to the simplicity of the system, the code size is very small. Initially, the code was written for and tested in a Linux environment. However, this code does not use the existing Linux file system. The same code was then modified enabling it to run on a bare PC by removing appropriate system calls and OS-related constructs. In the bare PC code, a direct hardware API is substituted to make it run on it. The Linux and bare systems were validated and used for testing as described in the next section. State transition diagrams were used to implement USB operations and their sequencing. The task structure that runs in the bare PC file system is similar to that used for bare Web servers [6], and runs on any Intel-based CPU that is IA32 compatible. It does not use a hard disk, but uses the BIOS to boot the system. The file system, boot code and the application are contained on the same USB. A bootable USB along with its application is generated by a special tool designed for bare PC applications.

7 NOVEL FEATURES

Since a bare application is self-supporting, it can be carried on a mass storage device such as a USB and run on any bare PC or bare machine. The mass storage device carries its own boot, loader and applications along with the file management system. An application programmer has total control of the application since it is independent of any external software such as an OS or kernel i.e., the bare system constitutes a single programming environment with no dependencies on any other vendor or software. This approach can be used to enhance the operation of bare email servers and bare Web servers by integrating the bare file system with these applications.

The lean USB file management system proposed here can be easily adapted for other pervasive devices including cell phones, smartphones, and other mobile devices. It is also possible to design a lean API for a file management system that can be used on a variety of such devices.

8 FUNCTIONAL OPERATION AND TESTING

The file management system was tested using the equivalent Linux and bare PC versions. A Total Phase (Beagle) analyzer [14] was used to capture functional traces to validate the systems. The internal timings were also measured to identify any basic performance issues. We found that the performance of the lean file management systems implemented on Linux and a bare PC are similar. For example on Linux, an open file command takes about 25.7 milliseconds as measured using the Beagle analyzer. Most of this time is spent in the driver and only 894 microseconds were spent in the file system. The results for the bare PC system were not significantly different. Thus, the efficiencies gained due to the bare PC approach were limited by the reduced speed of the USB controller when performing read and write operations. Since a USB mass storage device (I/O-bound) is much slower than a CPU, improvements in the file management systems are hard to measure using a USB-based file management system. More work is needed to compare these file systems using special benchmarks over a longer period of time to evaluate the overall system performance.

9 CONCLUSION

We demonstrated the feasibility of building a lean USB file management system that runs on a bare PC. We described the architecture, design and implementation of such a system, and performed preliminary tests using equivalent bare and Linux-based systems. The bare USB file system can be easily adapted to work with pervasive USB devices. Furthermore, the lean file system can be integrated with existing bare applications such as Web servers and email servers. A performance comparison between equivalent bare and Linux file systems was not conducted since preliminary tests showed that the USB device controllers are much slower than the CPU, and the performance gains in bare PC file systems were limited by the slow device controller speeds. Further studies are needed to evaluate OS-based and bare PC USB file systems.

10 REFERENCES

[1] Y. H. Chang, P. Y. Hsu, Y. F. Lu, and T. W. Kuo "A Driver-Layer Caching Policy for Removable Storage Devices", ACM Transactions on Storage, Vol. 7, No. 1, Article 1, June 2011, p1:1-1:23.

[2] M. Choi, H. Park, and J. Jeon, "Design and Implementation of a FAT File System for Reduced Cluster Switchign Overhead", 2008 International Conference on Multimedia and Ubiquitous Engineering.

[3] D. R. Engler and M.F. Kaashoek, "Exterminate all operating system abstractions", Fifth Workshop on Hot Topics in operating Systems, USENIX, Orcas Island, WA, May 1995, p. 78.

[4] Intel Corporation, Enhanced Host Controller Interface Specification for Universal Serial Bus, March 2002, Rev 1, http://www.intel.com/technology/usb/download/ehci-r10.pdf

[5] J. A. Garrison and A. L. N. Reddy, "Umbrella File System: Storage Management across Heterogeneous Devices", Vol. 5, No. 1, Article 3, March 2009, p3:1-3:24.

[6] L. He, R. K. Karne, and A. L. Wijesinha, "The Design and Performance of a Bare PC Web Server", International Journal of Computers and Their Applications, IJCA, Vol. 15, No. 2, June 2008, pp. 100-112.

[7] R. K. Karne, K. V. Jaganathan, and T. Ahmed, "DOSC: Dispersed Operating System Computing", OOPSLA '05, 20th Annual ACM Conference on Object Oriented Programming,Systems, Languages, and Applications, Onward Track, ACM, San Diego, CA, October 2005, pp. 55-61.

[8] R. K. Karne, K. V. Jaganathan, and T. Ahmed, "How to run C++ Applications on a bare PC", SNPD 2005, Proceedings of NPD 2005, 6th ACIS International Conference, IEEE, May 2005, pp. 50-55.

[9] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, R. Brightwell, "Palacios and Kitten: New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing", Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010), April, 2010.

[10] Microsoft Corp, "FAT32 File System Specification", http://microsoft.com/whdc/system/platform/firmware/fatgn.rnspx , 2000.

[11] V. S. Pai, P. Druschel, and Zwaenepoel. "IO-Lite: A Unified I/O Buffering and Caching System", ACM Transactions on Computer Systems, Vol.18 (1), ACM, Feb. 2000, pp. 37-66.

[12] Perisoft Corp, Universal Serial Bus Specification 2.0, http://www.perisoft.net/engineer/usb_20.pdf.

[13] B. Rawal, R. K. Karne, and A. L. Wijesinha. "Mini Web Server Clusters for HTTP Request Splitting", 2011, IEEE International Conference on High Performance, Computing and Communications, Banff, Canada, p94-100.

[14] Total Phase Inc., USB Analyzers, Beagle, http://www.totalphase.com.

[15] Universal Serial Bus Mass Storage Class, Bulk Only Transport, Revision 1.0, 1999, http://www.usb.org.