Implementing a USB File System for Bare PC Applications

William Thompson, Ramesh K. Karne, Sonjie Liang, Alexander L. Wijesinha, Hamdan Alabsi, and Hojin Chang
Department of Computer and Information Sciences
Towson University
Towson, MD 21252, U.S.
e-mail: {wvthompson, rkarne, sliang, awijesinha, halabs1, hchang}@towson.edu

Abstract—**Bare machine computing applications including Web servers, Webmail servers, SIP servers and SQLite require a file system that can also be used with an OS such as Windows or Linux. However, conventional file systems are OS-dependent and cannot be used with bare PC applications, which run without any OS or kernel support. This paper describes the implementation of a novel FAT-32 based USB file system for a bare PC, and provides details of its internal structures and the file API. Implementing a bare machine file system is challenging because it does not use any standard system libraries and requires integrating the USB driver and FAT32 file system with the bare PC application. The file system can be used with any existing or future bare PC application.**

Keywords- bare machine computing; bare PC applications; FAT32; file system; USB.

I. INTRODUCTION

File systems provide a means for organizing and retrieving the data needed by many computer applications. Typically, they are closely tied to the underlying operating system (OS) and mass storage technology. Bare machine file systems are, in contrast, independent of any OS or platform. Such a file system can be used with computer applications that run on a bare machine with no OS, and also in a conventional OS environment. The file system can serve as a basis to support future bare machine database management systems, big data systems, and Web and mobile applications that eliminate OS overhead and cost. Furthermore, it can be used in bare machine security applications that provide protection from attacks targeting OS vulnerabilities. In earlier work [14], a lean USB file system for a bare PC was described and relevant design issues were discussed. This paper focuses on the implementation and internals of a bare machine USB file system. It also defines a file API for bare PC applications.

The file system depends on the USB architecture [17], USB Mass Storage Specification [21], USB Enhanced Host Controller Interface Specification [6], FAT32 standard [15], and the bare machine computing paradigm. The file system is stored on a USB along with its application. The USB layout is similar to a memory layout providing a linear block addressing (LBA) scheme. That is, a USB address map is similar to a memory map. However, a USB is accessed with sector numbers that are directly mapped to memory addresses. It uses small computer system interface (SCSI) commands that are encapsulated in USB commands. Thus, a bare PC USB driver that works with this file system is needed [12]. The FAT32 standard is complex and has a variety of options that are needed for an OS based system as it is required to work with many application environments. The FAT32 options implemented in this system and the file API are designed for bare PC applications.

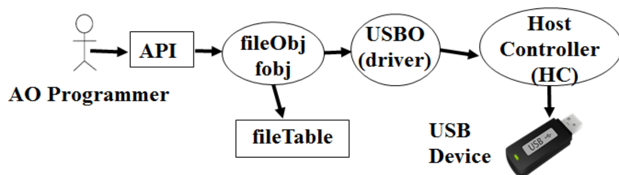


Figure 1. Bare machine USB file system.

Bare PC applications are based on the Bare Machine Computing (BMC) or dispersed OS computing paradigm [10]. This paradigm differs from a conventional approach as there is no underlying OS to manage resources. This means that the application programmer also has to deal with system programming aspects. Resident mass storage is not used in a bare PC, so applications are stored in a portable device such as a USB drive or in the cloud. The application is written primarily in C/C++ (with some assembly code) and runs as an application object (AO). An AO includes its own interfaces to the hardware [11] and the necessary OS-independent device drivers. Bare PC applications include Web servers [9], split servers [18], server clusters [19], email servers [5], SIP servers and user agents [1], and peer-to-peer VoIP systems [8].

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 describes the file API for bare PC applications. Section 4 gives details of file system internals. Section 5 presents functional operation. Section 6 contains the conclusion..

II. RELATED WORK

There are many approaches to reduce OS overhead, use lean kernels, or build a high-performance OS such as Exokernel [4], IO-Lite [16], and Palacios and Kitten [13]. While the BMC paradigm somewhat resembles these approaches, there is a significant difference in that bare machine applications run without any centralized code in the form of an OS or kernel. Flash memory has been used for mass storage devices as in the Umbrella file system [7], which also integrates two different types of storage devices. In [2], it is shown how to improve performance by adding cache systems at a driver level. In [3], a FAT32 file system for high performance clusters is implemented.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-474-9

```
h= createFile(fn, saddr, size, attr)
deletFile (h)
resizeFile (h, size)
flushFile (h)
flushAll ()
```

Figure 2. File API functions.

In [14], the design of a lean USB file system for bare PC applications was discussed and an initial version of the file system was built and tested. That work showed the feasibility of developing a file system without any OS support. However, the file system was not easy to modify or use with existing bare PC applications. This paper describes the implementation of an enhanced USB file system with a simple file API for bare PC applications.

III. FILE API

In a bare PC application, code for data and file systems reside on the same USB. In addition to the application, the USB has the boot code and loader in a separate executable, which enables the bare PC to be booted from the USB. The application suite (consisting of one or more end-user applications) is a self-contained application object (AO) [11] that encapsulates all the needed code for execution as a single entity. For example, a Webmail server, SQLite database and the file system can all be part of one AO. Since no centralized kernel or OS runs in the machine, the AO programmer controls the execution of the application on the machine. When an AO runs, no other applications are running in the machine. After the AO runs, no trace of its execution remains.

An overview of the USB file system for bare PC applications is shown in Figure 1. The simple API for the file system consists of five functions to support bare PC applications. These are (1) createFile(), (2) deleteFile(), (3) resizeFile(), (4) flushFile() and (5) flushAll(). These functions provide all the necessary interfaces to create and use files in bare PC applications. The fileObj (class) uses a fileTable data structure to manage and control the file system. A given API call in turn interfaces with the USBO object, which is the bare PC device driver for the USB [12]. This device driver has many interfaces to communicate directly with the host controller (HC). The HC interfaces with USB device using low-level USB commands.

Figure 2 lists the file API functions, and Figure 3 shows an example of their usage. The parameters for the createFile() function are file name (fn), memory address pointer (saddr), file size (size) and file attributes (attr); it

```
char *ptr;
char *readArray;
FileObj fobj;
h = fobj.createFile(fileName,
    &startAddress, &fileSize, attr);
ptr = (char *)startAddress;
for(i = 0; i < fileSize; i++)
     ptr[i] = 0; //write to file
for(i = 0; i < fileSize; i++)
     readArray[i] = ptr[i]; //read from
file
fobj.flushFile(h);
```

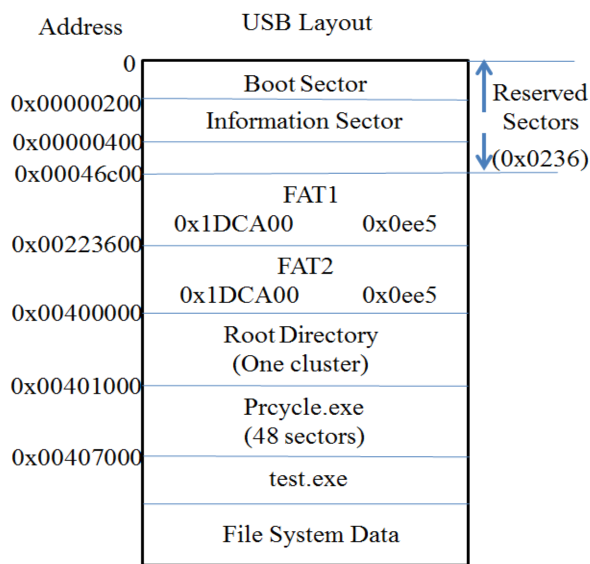
Figure 3. File API Usage.

```
GetReservedSectors() 0xe - 0xf (0x0236)
GetNumOfFats() 0x10 (02)
GetNumOfSectorsPerFat() 0x24 - 0x27 (0x0ee5)
GetSectorsPerCluster() 0x0d (08)
GetNumOfSectorsInPart() 0x20 -0x23 (0x003bafff)
GetClusterOfStartRootDir() 0x2c - 0x2f (02)
GetNumOfClustersInRootDir() (third entry in FAT, 04)
GetFATEntryPoint()
GetDirectoryEntryPoint()
```

Figure 4. USB parameters

returns a file handle (h). The file handle is the index value of the file in the fileTable structure, which has all the control information of a file. This approach considerably simplifies file system design as it can be used as a direct index into the fileTable without the need for searching. The deleteFile(h) function uses the file handle to delete a file. When a file is deleted, it simply makes a mark in the fileTable structure and its related structures such as the root directory and FAT table. The resizeFile() function is used to increase or decrease a previously allocated file size. Thus, an AO programmer needs to keep track of the growth of a file from within the application. The flushFile() function will update the USB mass storage device from its related data structures and memory data. An AO programmer has to call this function periodically or at the end of the program to write files to persistent storage. The flushAll() interface is used to flush all files and related structures onto the USB drive. Note that the programmer gets a file address, uses it as standard memory (similar to memory mapped files), and manages the memory to read and write to a file. There is no need for a read and write API in this file system. All standard file IO operations are reduced to the list shown in Figure 2.

A significant difference between the bare PC file system and a conventional OS-based file system is that an AO programmer directly controls the USB device through the API. That is, a user program directly communicates with the hardware without using an OS, kernel or intermediary software. For instance, the createFile() function invokes the



f

Figure 5. USB layout.

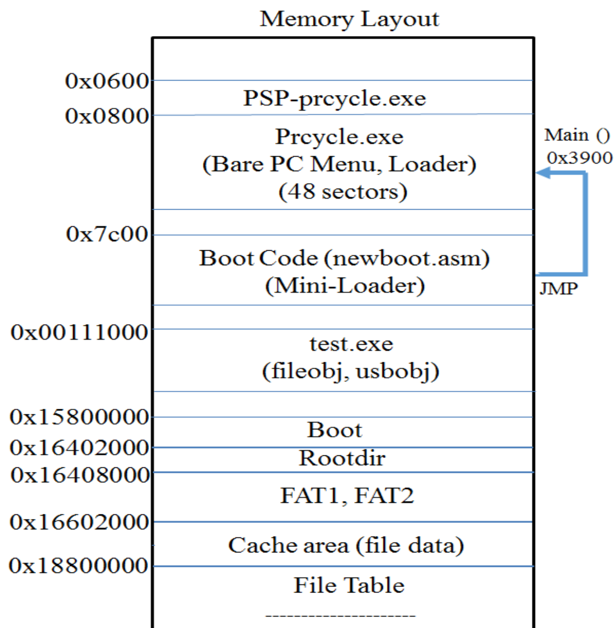


Figure 6. Memory map

ileObj function, which in turn invokes the USBO function. The latter then calls the HC low-level functions. In this approach, an API call runs as a single thread of execution without the intervention of any other tasks. Thus, writing a bare PC application is different from writing conventional programs as there is no kernel or centralized program running in the hardware to control the application. These applications are designed to run as self-controlled, self-managed and self-executable entities. In addition, the application code does not depend on any external software or modules since it is created as a single monolithic executable.

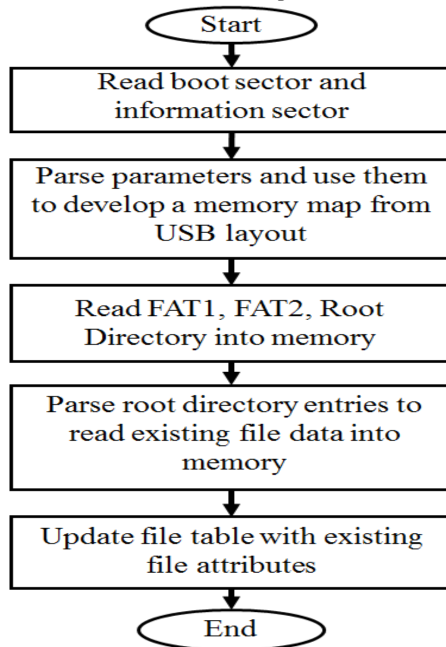


Figure 7. Initialization.

4	4	4	4	4	4	4	4 Bytes
File Index (h)	File Size	Starting Cluster	# of Cluster	Start Addr.	End Addr.	Start Sect or #	Attr.
0	1	2	3	4	5	6	7
File Name - 64 byte							

Figure 8. File Table Entry (FTE)

IV. FILE SYSTEM INTERNALS

Building a USB file system for bare PC applications is challenging. The system involves several components and interfaces, and it is necessary to map the USB specifications to work with the memory layout in a bare PC application and the bare machine programming paradigm. Details of file system internals are provided in this section to illustrate the approach.

A. USB Parameters

Each USB has its own parameters depending on the vendor, size and other attributes. Some parameters shown in Figure 4 are used for identification and laying out the USB memory map. These parameters are analogous to a schema in a database system and are located in the 0th sector.

B. USB and Memory Layout

Figure 5 displays the USB layout for a typical file system with 2GB mass storage. The boot sector contains many parameters as shown in Figure 4. The reserved sectors parameter is used to calculate the start address of FAT1 table. The number of sectors per FAT defines the size of FAT1 and FAT2 tables, which are contiguous. The root directory entry follows the FAT2 table as shown in Figure 5.

The number of clusters in the root directory and number of sectors per cluster defines the starting point for the files stored in the USB. The root directory has 32 byte structures for each file on the USB. These 32 byte structures describe the characteristics of a FAT32 file system. The layout in Figure 5 shows two files prcycle.exe and test.exe. The first file is the entry point of a program after boot and the second one is the application. Other mass storage files created by the application are located after test.exe. The bare PC file system has to manage the FAT tables, root directory and file system data.

The USB layout and its entry points are used to map these sectors to physical memory. A memory map is then drawn as shown in Figure 6. During the boot process, the BIOS will load the boot sector at 0x7c00 and boot up the machine. This code will run and load prcycle.exe using a mini-loader. When prcycle.exe runs, it provides a menu to load and run the application (test.exe). The original boot, root directory and FATs as well as other existing files and

```
usbo.ResetUSBPluggedIn()
usbo.ReadUSBDesc()
usbo.SetupUSB()
usbo.ClearFeature()
usbo.WriteOp()
usbo.ReadOp()
```

Figure 9. USB operations.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-474-9

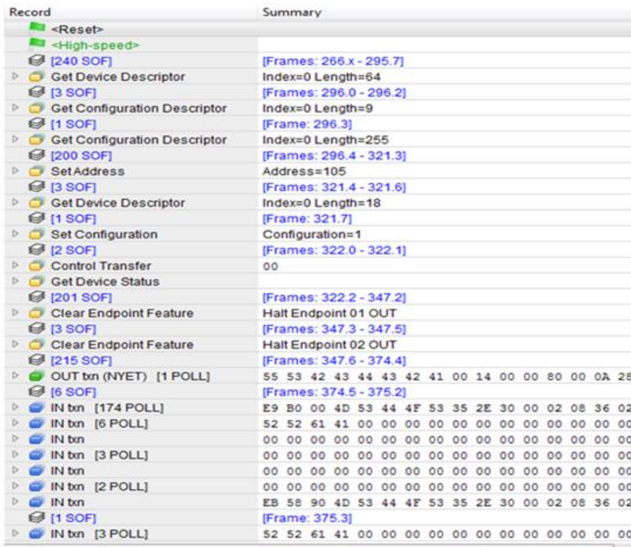


Figure 10. Analyzer trace.

data in the USB are also stored in memory to manage them as memory mapped files. The cache area stores all the user file data and provides direct access to the application program. In this system, the USB and memory maps are controlled by the application and not by middleware.

C. *Initialization*

The Figure 7 illustrates the initialization process after the bare PC starts. During initialization, existing files from the USB are read into memory and file table attributes are populated. In addition, FAT tables and other relevant parameters are read and stored in the system. If the file data size is larger than the available memory, then partial data is read as needed and the file tables are updated appropriately. A contiguous memory allocation strategy is used to manage real memory. Because the file handle serves as a direct index to the file table, the file management system is simplified.

D. *File Table Entry (FTE)*

The FTE is a 96-byte structure as shown in Figure 8. The file name is limited to 64 bytes including name and type. 32-byte control fields are used to store the file control information needed to manage files. These attributes are derived from the root directory, FAT tables and memory map. The file index is the first entry in the FTE and it indicates the index of the file table. The index is also used as a file handle to be returned to the user for file control.

E. *File Operations*

The five file operations in the bare PC system use the data structures file table and device driver interfaces.

Name	Date modified	Type	Size
PRCYCLE.EXE	9/17/2015 3:52 P...	Application	23 KB
test.exe	9/17/2015 3:52 P...	Application	217 KB
test1.txt	9/17/2015 3:52 P...	Text Docu...	98 KB
test2.txt	9/17/2015 3:52 P...	Text Docu...	69 KB
testing 123456.txt	9/17/2015 3:52 P...	Text Docu...	49 KB
This is a long filename.txt	9/17/2015 3:52 P...	Text Docu...	98 KB

Figure 11. Windows trace.

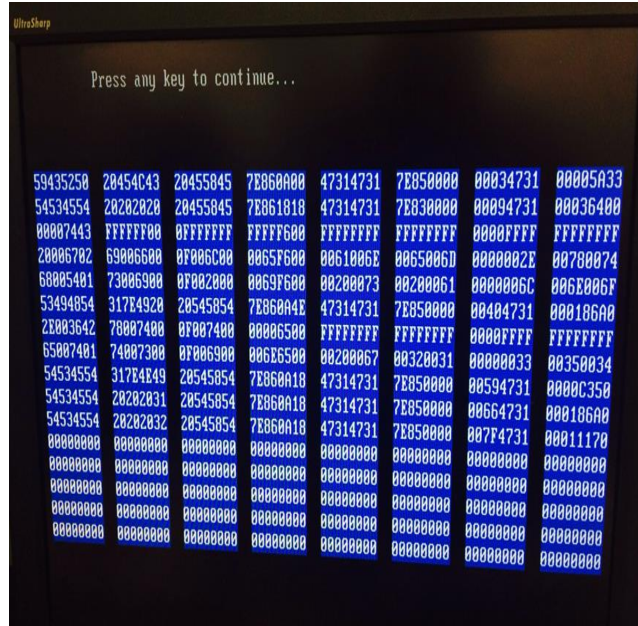


Figure 12. Bare PC root directory.

The file system only covers a single directory structure. When createFile() is called, it first checks the file table for any existing file using the file name. If this file does not exist, a new file is created with the given file name and requested file size. Then an entry is made in the file table, memory is assigned, and the root directory and FAT entries are created for the file. When flushFile() is called, it updates the USB and the call returns the file handle, which is an index into the file table. Similarly, deleteFile() will delete the file from the file table and flushAll() will update the USB with all the USB data fields. The resizeFile() interface simply uses the same entry with a different memory pointer and keeps the data "as is" unless the size is reduced. When the size is reduced, the extra memory is reset. All API calls and their internals are visible to the programmer.

F. *File Name*

The file system supports both short and long file names. At present, long file names are limited to 64 characters by design since they introduce difficulties when creating the root directory and file table entries. The FAT32 root directory structure also results in complexity that affects file system implementation.

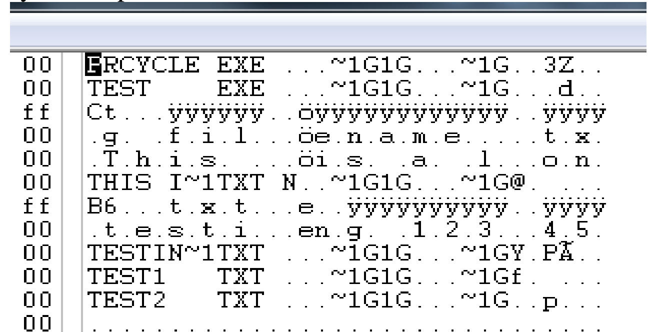


Figure 13. USB root directory.

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-474-9

G. *System Interfaces*

The The USB file system runs as a separate task in the bare PC AO. The AO has one main task, one receive task and many application tasks such as server threads. The main task enables plug-and-play when the USB drive is plugged into the system. Each USB slot in the PC is managed as a separate task. Tasks and threads are synonymous in bare PC applications as threads are implemented as tasks in the system. Each event in the system is treated as a single thread of execution without interruption. Thus, each file operation runs as one thread of execution. There is no need for concurrency control and related mechanisms in a bare PC application. The files generated in the bare PC system can be read on any OS that supports FAT32 such as Windows, Linux or Mac.

V. OPERATION

The file system is written in C/C++, while the device driver code is written in C and MASM. The MASM code is 27 lines and provides two functions that read and write to control registers in the host controller. The fileObj code is 4262 lines including comments (30% of the code), and one class definition. State transition diagrams are used to implement USB operations and their sequencing. For example, some of the state transitions occurring during the initialization process are shown in Figure 7. The fileObj in turn invokes the USB device driver calls shown in Figure 9.

File operations can be done anywhere in the bare PC application. The task structure that runs in the bare PC file system is similar to that used for bare Web servers [9], and runs on any Intel-based CPU that is IA32 compatible. Bare PC applications do not use a hard disk; instead, the BIOS is used to boot the system. The file system, boot code and application are stored on the same USB. A bootable USB along with its application is generated by a special tool designed for bare PC applications. The USB file system was integrated with the bare PC Web server for functional testing.

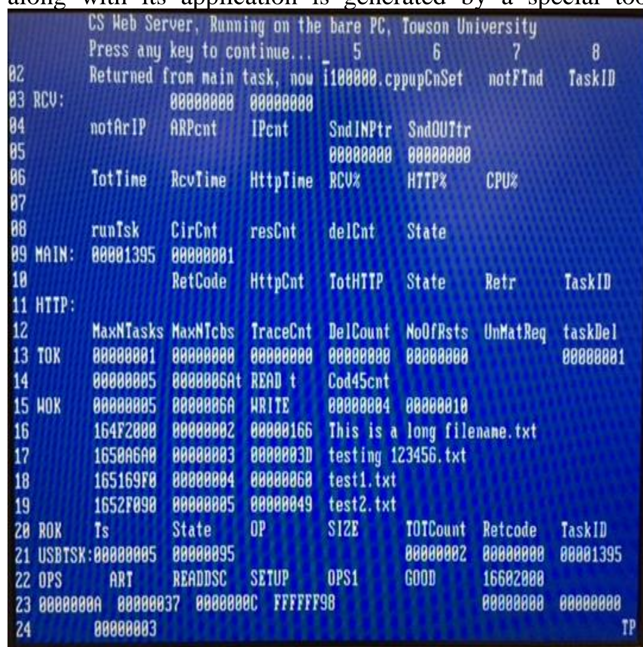


Figure 14. Bare PC screen shot.

The operation of the bare PC file system is demonstrated by having two existing files (prcycle.exe and test.exe) on the USB along with the boot code. Small and large files are created by the application with file sizes varying up to 100K. To demonstrate file operations, four files were created and tested as described here in addition to the two files prcycle.exe and test.exe on the USB (after the program runs, there a total of six files on the USB). The data were read from the files and also written to them using the file API. A USB analyzer [20] was used to test and validate the file system and the driver. Figure 10 shows a sample trace from the analyzer that illustrates reset, read descriptors, set configuration and clear. These low level USB commands are directly controlled by the programmer (they are a part of the bare PC application).

Figure 11 shows the six files that exist on the USB displayed on the screen of a Windows PC. The four created files can be read from the Windows PC. Figure 12 shows the file system in the bare PC root directory in memory. This directory is used to update the files until they are flushed. Figure 13 shows the root directory entries on the USB after the program is complete. Figure 14 is a screen shot on the bare PC showing the four files (short and long) created successfully by the system. The bare PC screen is divided into 25 rows and 8 columns to display text using video memory. This display is used by the programmer to print functional data, and for debugging. The programmer controls writing to the display directly from the bare PC application, with no interrupts used for display operations.

VI. CONCLUSION

We described the implementation of a novel bare machine USB file system designed for applications that run without the support of any OS environment/platform, lean kernel or embedded software. We also presented a file API for bare PC applications. The file system enables a programmer to build and control an entire application from the top down to its USB data storage level without the need for an OS or intermediary system. This implementation can be used as a basis for extending bare PC file system capabilities in the future. The file system can be integrated with bare PC applications such as Web servers, Webmail/email servers, SIP servers and VoIP clients.

REFERENCES

[1] A. Alexander, A. L.Wijesinha, and R. Karne, “Implementing a VoIP SIP server and a user agent on a bare PC”, 2nd International Conference on Future Computational Technologies and Applications (Future Computing), 2010, pp. 8-13.

[2] 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.

[3] M. Choi, H. Park, and J. Jeon, “Design and implementation of a FAT file system for reduced cluster switching overhead”,

International Conference on Multimedia and Ubiquitous Engineering, 2008, pp. 355-360.

[4] D. R. Engler and M.F. Kaashoek, “Exterminate all operating system abstractions”, Fifth Workshop on Hot Topics in Operating Systems, USENIX, 1995, p. 78.

[5] G. H. Ford, R. K. Karne, A. L. Wijesinha, and P. Appiah-Kubi, “The design and implementation of a bare PC email server”, 33rd IEEE Computer Software and Applications Conference (COMPSAC), 2009, pp. 480-485.

[6] 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 [retrieved: April 8, 2016]

[7] J. A. Garrison and A. L. N. Reddy, “Umbrella file system: Storage management across heterogeneous devices”, ACM Transactions on Storage (TOS), Vol. 5, No. 1, Article 3, March 2009.

[8] G. Khaksari, A. Wijesinha, R. Karne, L. He, and S. Girumala., “A peer-to-peer bare PC VoIP application”, IEEE Consumer Communications and Networking Conference (CCNC) 2007, pp. 803-807.

[9] 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.

[10] R. K. Karne, K. V. Jaganathan, N. Rosa, and T. Ahmed, “DOSC: dispersed operating system computing”, 20th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2005, pp. 55-61.

[11] R. K. Karne, K. V. Jaganathan, and T. Ahmed, “How to run C++ applications on a bare PC”, 6th ACIS Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD) 2005.

[12] R. K. Karne, S. Liang, A. L. Wijesinha, and P. Appiah-Kubi, “A bare PC mass storage USB device driver”, International Journal of Computers and Their Applications, Vol 20, No. 1, March 2013, pp. 32-45.

[13] J. Lange et al., “Palacios and Kitten: New high performance operating systems for scalable virtualized and native supercomputing”, 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2010, pp. 1-12.

[14] S. Liang, R. Karne, and A. L. Wijesinha, “A lean USB file system for bare machine applications”, 21st Conference on Software Engineering and Data Engineering (SEDE), 2012, pp. 191-196.

[15] Microsoft Corp, “FAT32 file system specification”, http://microsoft.com/whdc/system/platform/firmware/fatgn.rnspx, 2000. [retrieved: April 8, 2016]

[16] V. S. Pai, P. Druschel, and W. Zwaenepoel. “IO-Lite: A unified i/o buffering and caching system”, ACM Transactions on Computer Systems, Vol.18 (1), Feb. 2000, pp. 37-66.

[17] Perisoft Corp, Universal serial bus specification 2.0, http://www.perisoft.net/engineer/usb_20.pdf. [retrieved: April 8, 2016]

[18] B. Rawal, R. Karne, and A. L. Wijesinha, “Splitting HTTP requests on two servers”, 3rd Conference on Communication Systems and Networks (COMSNETS), 2011, pp. 1-8.

[19] B. Rawal, R. K. Karne, and A. L. Wijesinha. “Mini Web server clusters for HTTP request splitting”, IEEE Conference on High Performance, Computing and Communications (HPCC), 2011, pp. 94-100.

[20] Total Phase Inc., USB analyzers, Beagle, http://www.totalphase.com. [retrieved: April 8, 2016]

[21] Universal serial bus mass storage class, bulk only transport, revision 1.0, 1999, http://www.usb.org [retrieved: April 8, 2016]

Copyright (c) IARIA, 2016. ISBN: 978-1-61208-474-9