A Mass Storage System for Bare PC Applications Using USBs

William V. Thompson, Hamdan Alabsi, Ramesh K. Karne, Sonjie Liang, Alexander L. Wijesinha, Rasha Almajed, and Hojin Chang
Department of Computer & Information Sciences
Towson University
Towson, MD
(wvthompson, halabsi, rkarne, sliang, awijesinha, ralmajed, hchang) @towson.edu

***Abstract*—Bare machine applications eliminate the overhead and the security vulnerabilities that are due to operating systems. This paper describes a mass storage system for bare PC applications that uses USBs. It is implemented by extending a scalable FAT32 USB file system for a bare PC. First, details of the bare PC file system including the file API, file system internals, and file operations are given. Then the architecture of the mass storage system and its design and implementation are presented. A mass storage system based on this architecture is built by using four USBs on a desktop PC. Capabilities of the mass storage system are demonstrated by storing conventional files and SQLite database files on multiple USBs. Experiments to measure raw versus conventional file system performance show a 12% improvement for writes and a 33% improvement for reads with 30 MB files. This work is a first step towards building mass storage systems to support future bare machine big data and mobile applications with improved security and performance.**

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

I. INTRODUCTION

Mass storage systems are used with Web servers, database systems, clients, and numerous applications. Media for mass storage include hard disk drives, optical drives, memory cards, and USB flash drives. A mass storage system for a conventional application typically requires the support of an operating system (OS) or kernel. This paper considers a mass storage system for bare PC applications using USBs.

Bare PC applications are based on the BMC (Bare Machine Computing) paradigm, which eliminates the vulnerabilities and overhead of an OS. In the BMC paradigm, no OS, kernel, or middleware is required to communicate between an application and the hardware, i.e., an application contains everything it needs to run on a bare machine or bare PC. The BMC application uses direct interfaces to communicate with the hardware.

Mass storage systems are associated with file systems that manage both conventional data files and raw data files. 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. The mass storage system for bare PC applications uses a file system that is independent of any OS or platform. Such a file system can also be used by OS-based applications.

We first describe the bare PC file system including the file API [1]. We then show how the file system can be used as a basis for the architecture, design, and implementation of a mass storage system for bare PC applications using USBs. USB flash drives are ideal for file systems and mass storage in bare machine computing as they are inexpensive and able to store increasing amounts of data. We also demonstrate the capabilities of a bare PC mass storage system consisting of four USBs on a desktop PC by storing conventional files and SQLite database files on multiple USBs. We lastly present experimental results using the bare PC mass storage system to compare raw versus conventional file performance for both writes and reads. The bare PC file system and mass storage system can be used to support future bare machine database management systems and big data applications, or Web servers and mobile applications.

The rest of the paper is organized as follows. Section II gives a brief overview of BMC applications and related work. Section III gives details of the bare PC file system and its operation. Section IV describes the system architecture for the mass storage system, and Section V presents its design and implementation. Section VI illustrates functional operation, and presents the experimental performance results. Section VII concludes the paper and suggests possibilities for future research.

II. BMC APPLICATIONS AND RELATED WORK

A BMC application suite consists of an application, the necessary protocols and drivers, and the code to boot and load the application. A hard disk is not used in a bare PC, so all the code must be stored on a removable device such as a secured USB flash drive. Currently, BMC applications run on any x86-based bare PC. Bare PC applications include a Webserver [2][3], Webmail and email servers [4][5], split protocol servers [6], server clusters [7], SIP servers and user agents [8], and peer-to-peer VoIP systems [9].

Many approaches have been used to reduce OS overhead or build high-performance systems. Some use lean kernels, while others move OS-functionality into applications. Examples include Exokernel [10], IO-Lite [11], OS-Kit [12], and Palacios and Kitten [13].

The BMC paradigm [14] eliminates the OS and uses direct interfaces to the hardware to run applications on a bare PC. This approach to computing differs from a conventional approach as there is no underlying OS to manage resources, i.e., the application programmer manages memory and schedules tasks. The application is written primarily in C/C++ (with some assembly code) and runs as an AO (Application Object) that includes its own interfaces to the hardware [16] and the necessary OS-independent device drivers.

There are many types of file system specifications such as FAT32 [17], NTFS [18] and exFAT [19]. The BMC file system currently uses FAT32 as it is simple and easy to implement. The FAT32 file system has been used for building high performance clusters [20]. The Umbrella file system, which also integrates two different types of storage devices, is an example of a mass storage system that uses USB flash drives [21]. Driver-level caching can be used to improve file system for removable storage devices [22]. However, removable storage media can be exploited through the OS [23]. Bare PC USB file systems and mass storage systems have no OS-related vulnerabilities. Since there is no OS, a USB device driver needs to be integrated with the bare PC application [24].

SQLite is a lean database management system [25]. It is self-contained, self-configured, and stand-alone (i.e., it does not require a separate client and a server). SQLite is included in Web browsers, mobile devices and embedded systems. It requires an OS and the amalgamated version has about 130K lines of code. SQLite has been transformed to run on a bare PC with no OS or kernel [26].

III. BARE PC FILE SYSTEM

The material in this section previously appeared in [1]. The bare PC USB file system depends on the USB architecture [27], USB Mass Storage Specification [28], USB Enhanced Host Controller Interface Specification [29], FAT32 standard [17], and the BMC paradigm [14]. The file system is stored on a USB along with its application. The USB layout is similar to a memory layout providing a LBA (Linear Block Addressing) 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 SCSI (small Computer System Interface) commands [30] that are encapsulated in USB commands. Thus, a bare PC USB driver that works with this file system is needed [24]. 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.

In [31], 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. However, the file system was not easy to modify or use with existing bare PC applications. The rest of this section describes the implementation of an enhanced USB file system with a simple file API for bare PC applications.

A. File API

In a bare PC application, code for data and the file system reside on the same USB. In addition to the application as noted above, 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 AO 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 Fig. 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 [24]. 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.

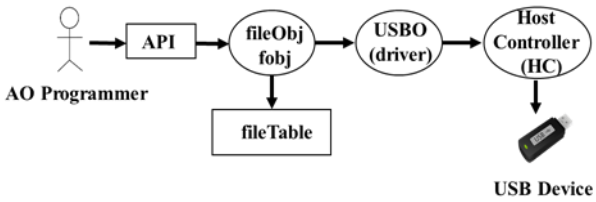

Figure 1. Bare machine USB file system.

Fig. 2 lists the file API functions, and Fig. 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 returns a file handle (h).

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

Figure 2. File API functions.

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 (File Allocation Table).

```
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.

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 Fig. 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 fileObj 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.

B. 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.

1) USB Parameters: Each USB has its own parameters depending on the vendor, size and other attributes. Some parameters shown in Fig. 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.

```
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.

2) USB and Memory Layout: Fig. 5 displays the USB layout for a typical file system with 2GB mass storage.The boot sector contains many parameters as shown in Fig. 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 Fig. 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 Fig. 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.

3) Memory Map: 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 Fig. 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 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.

4) Initialization: Fig. 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.

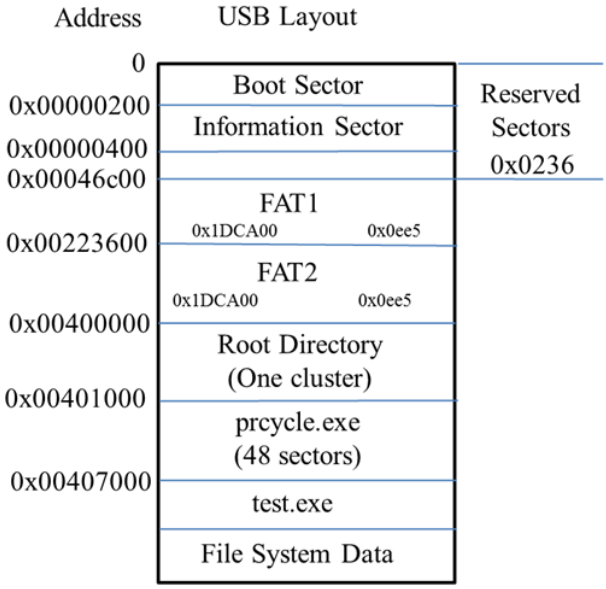

Figure 5. USB layout.

5) File Table Entry (FTE): The FTE is a 96-byte structure as shown in Fig. 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.

6) File Operations: The five file operations in the bare PC system use the data structures file table and device driver interfaces. 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.

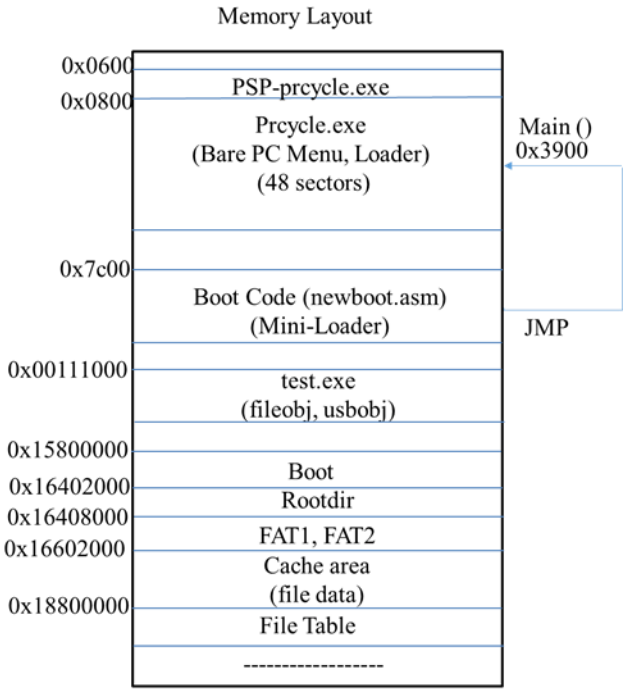

Figure 6. Memory map.

7) 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.

8) System Interfaces: 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.

C. 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 Fig. 7. The fileObj in turn invokes the USB device driver calls shown in Fig. 9.

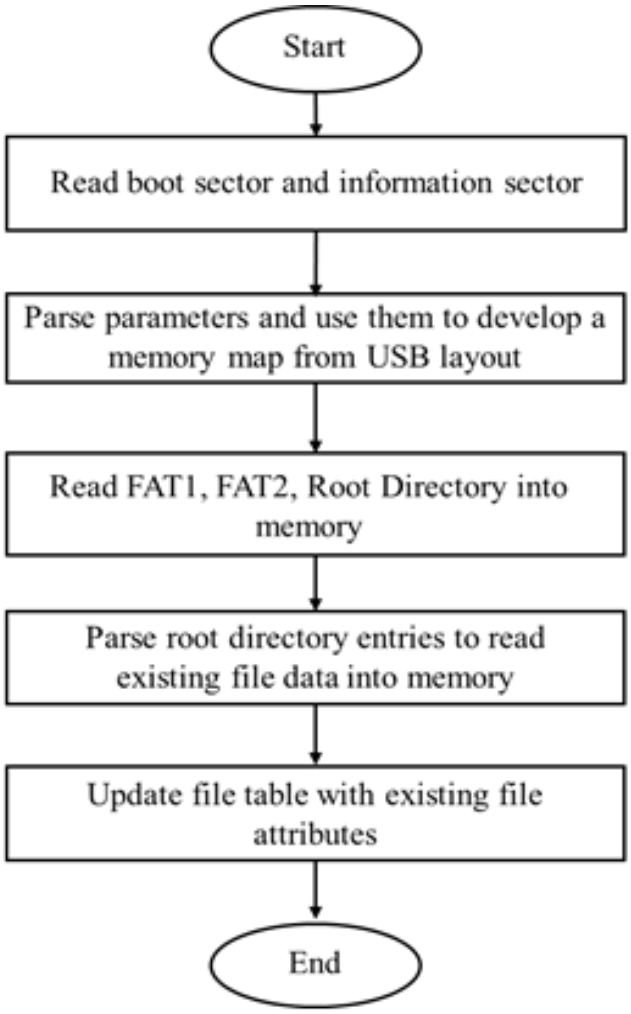

Figure 7. Initialization.

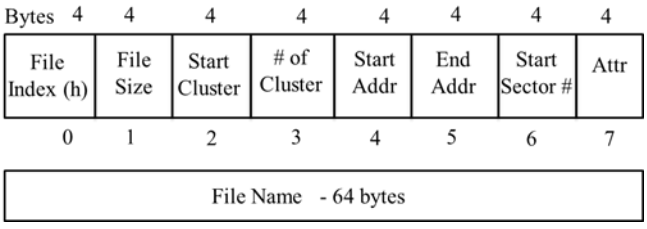

Figure 8. File Table Entry (FTE).

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 the bare Web servers, 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.

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

Figure 9. USB operations.

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 is 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 [32] was used to test and validate the file system and the driver. Fig. 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).

Fig. 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. Fig. 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. Fig. 13 shows the root directory entries on the USB after the program is complete. Fig. 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.

IV. ARCHITECTURE

We now describe a mass storage system for BMC applications by integrating the bare PC file system and the SQLite database system. This mass storage system can then be adapted for use with existing BMC applications. Fig. 15 shows the system architecture for the mass storage system.

A USB flash drive is a complete file system by itself, which consists of boot, FAT, root directory and file data. As main memories are getting cheaper and larger, it is becoming feasible to map high-capacity flash drives into main memory. Such memory maps enable easier implementation and high

performance while avoiding the need for an intricate memory management system.

The file system resident on the USB flash drive is represented by a 32-byte data structure capturing its file attributes. This is similar to a 32-byte structure in a FAT32 root directory structure. Conventional and SQLite files can all be represented with the same data structure.

Record	Summary
<Reset>	
<High-speed>	
[240 SOF]	[Frames: 266.x - 295.7]
Get Device Descriptor	Index=0 Length=64
[3 SOF]	[Frames: 296.0 - 296.2]
Get Configuration Descriptor	Index=0 Length=9
[1 SOF]	[Frame: 296.3]
Get Configuration Descriptor	Index=0 Length=255
[200 SOF]	[Frames: 296.4 - 321.3]
Set Address	Address=105
[3 SOF]	[Frames: 321.4 - 321.6]
Get Device Descriptor	Index=0 Length=18
[1 SOF]	[Frame: 321.7]
Set Configuration	Configuration=1
[2 SOF]	[Frames: 322.0 - 322.1]
Control Transfer	00
Get Device Status	
[201 SOF]	[Frames: 322.2 - 347.2]
Clear Endpoint Feature	Halt Endpoint 01 OUT
[3 SOF]	[Frames: 347.3 - 347.5]
Clear Endpoint Feature	Halt Endpoint 02 OUT
[215 SOF]	[Frames: 347.6 - 374.4]
OUT txn (NYET) [1 POLL]	55 53 42 43 44 43 42 41 00 1
[6 SOF]	[Frames: 374.5 - 375.2]
IN txn [174 POLL]	E9 B0 00 4D 53 44 4F 53 35 2
IN txn [6 POLL]	52 52 61 41 00 00 00 00 00 0
IN txn	00 00 00 00 00 00 00 00 00 0
IN txn [3 POLL]	00 00 00 00 00 00 00 00 00 0
IN txn	00 00 00 00 00 00 00 00 00 0
IN txn [2 POLL]	00 00 00 00 00 00 00 00 00 0
IN txn	EB 58 90 4D 53 44 4F 53 35 2
[1 SOF]	[Frame: 375.3]
IN txn [3 POLL]	52 52 61 41 00 00 00 00 00 0

Figure 10. Analyzer trace.

The **"file index"** field is the entry point used as an index into the root directory. The **"file size"** shows the number of bytes in the file. The**"start cluster"** and **"# of clusters"** show the starting point of the cluster on the physical media and the number of clusters needed for the entire file. Usually, a cluster is defined as 8 sectors, but a larger value can be used for larger files. The **"start addr"** and **"end addr"** fields define the start and end of the physical memory map in the system. In BMC systems, all memory is physical memory, which avoids virtual memory and paging overhead. The **"start sector#"** identifies the LBA needed to access the flash drive. The LBA scheme on the USB provides a convenient way to address it, which is similar to addressing main memory. However, the USB device needs to use SCSI (small computer system interfaces) [30] commands for access. The **"file attr"** field defines the file permissions needed to store the file in the system. Each file in the system needs one 32-byte record, which provides all the control information needed to manage the mass storage system.

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.

As a mass storage system has high capacity (order of terabytes or more), a large number of flash drives is needed. Although a desktop usually provides only a dozen USB ports, this limit can be increased to 127 ports by using a USB HUB. Each device is addressed by a unique **"portno"** in the range 1 to 127. The system architecture allows creation of a separate task for managing each port resulting in a **"taskindex"** from 0-126 (=portno–1). In a BMC application, one can define as many tasks as needed for managing ports; alternatively, a single task can be used to manage multiple USBs.

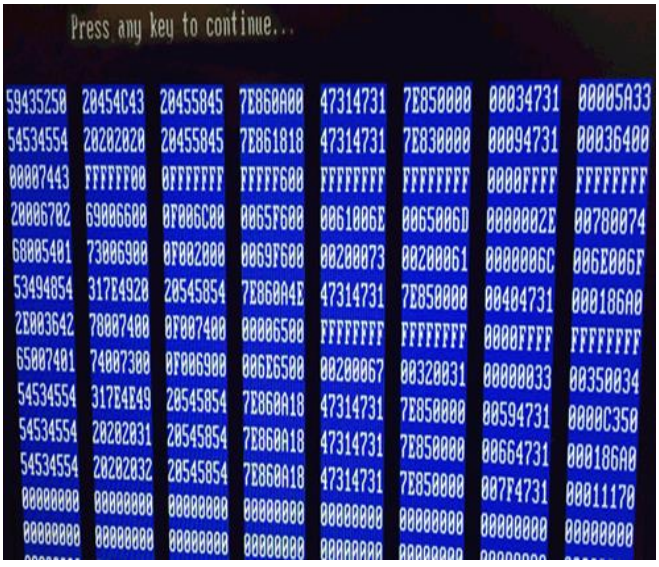

Figure 12. Bare PC root directory.

Mass storage needed for BMC applications can use up to 4 GB of real memory as this is the limit for a 32-bit address

space. If more storage is required, then the resident memory can be swapped in and out of persistent storage on to flash drives, without the need for virtual memory or paging. Raw file structures in temporary storage on the flash drives can be then used for swap space. We can also use a 64-bit CPU that can provide larger main memory capacity and a larger address space. The BMC architecture for the mass storage system is scalable and simple, and can be extended to meet the needs of new applications and future advances in technology. The design and implementation of the mass storage system based on the bare PC file system are detailed in the next two sections.

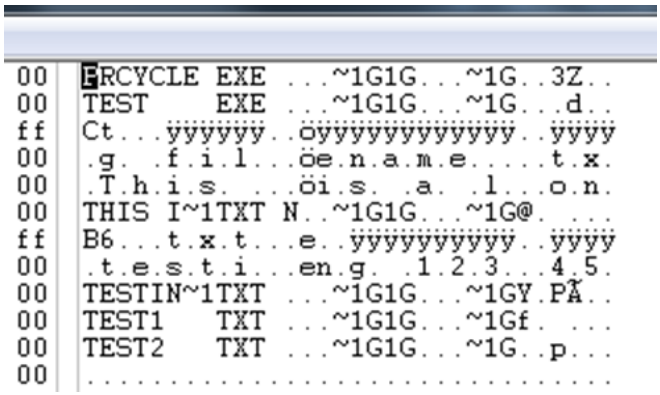

Figure 13. USB root directory.

V. DESIGN AND IMPLEMENTATION

The BMC mass storage system requires mechanisms to integrate the file system with the bare PC application. We integrated the SQLite file system with the application by using a bridge and interfaces from C to C++. The multiple USBs in a desktop need to host and manage multiple file systems in memory. The plug-play feature of USBs requires task structures that can detect the activity of flash drives and provide appropriate functionality. As the USB driver is now part of the application suite, timing-related and device-related knowledge must be integrated with the application. The device driver has to be managed by a separate file system task as it requires internal transactions and setup to perform USB operations. When SQLite is included as part of the mass storage system, it requires special handling of database functions to include the user interface and background operations.

A. Bridge between C/C++

The BMC code is written as object-oriented C++ programs with some C and Microsoft assembly code. The API to address hardware takes a path from a C++ function to a C function and then to an assembly function as needed. The C functions are used in C++ by defining them in **"extern"** blocks. This is a normal operation where C++ can call C code as is acceptable to go from strict type checking to no type checking. The SQLite code is written in C and it requires to communicate to the bare PC application code for the file system, device driver, and other function calls. Calling C++ from C to C++ violates object-oriented principles and weakens the strict type checking of C++. The bridge shown in Fig. 16 enables communication from C to C++. There are variety of ways to implement this bridge [33]. The C to C++ bridge can be summarized in four steps. In Step 1, capture the C++ member function address and store it in shared memory. In Step 2, define a C function header and a **"typedef"** for a dummy function. In Step 3, implement the C function, where the dummy function address is derived from the member function address in C++, which is stored in shared memory. In Step 4, simply define a C prototype where it is needed and call the C function. Notice that the **"typedef"** function signature must be the same as the C function call. We have defined many such functions in SQLite database to call the bare PC C++ functions to achieve the necessary integration.

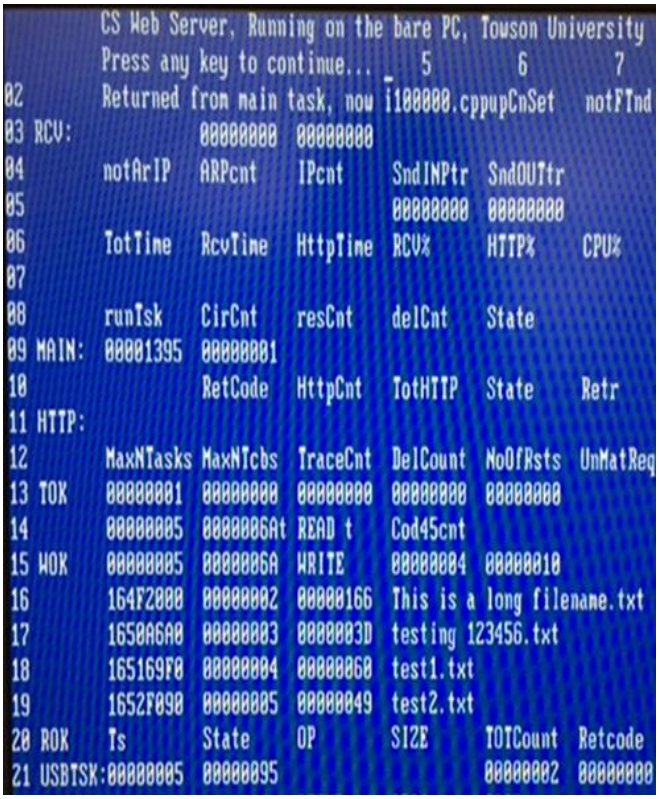

Figure 14. Bare PC screen shot.

B. USB Operation Flow Diagram

The USB operation flow diagram is shown in Fig. 17. Every time a USB is plugged in, it goes through a sequence of operations including: reset, read descriptors to capture device parameters, setup, clear feature to enable its end points, test unit ready, and read/write. The control flow for each USB has to go through these steps before it can be used for read and write operations. Note that some of these operations are SCSI commands encapsulated in the USB commands. The order of these operations are very important to make the USB operational. In addition, there are some built-in delays needed for reset operation. Determining these delay values and adjusting them as needed in the bare PC USB device driver proved to be somewhat challenging. Fig. 18 shows the design

and implementation of the approach used for managing the USB ports. There are two USB controllers in the Optiplex 960 desktop system. One controller provides four ports in the front of the machine, which are used for testing this architecture. The second controller's ports are in the back of the machine. A single task is designed to manage these four ports. These port numbers vary from 3, 4, 5, and 6. Their task indexes are one less than the port numbers. Each USB has its own file system that is resident on the flash drive. The control program is designed to check each port for its operation and functionality.

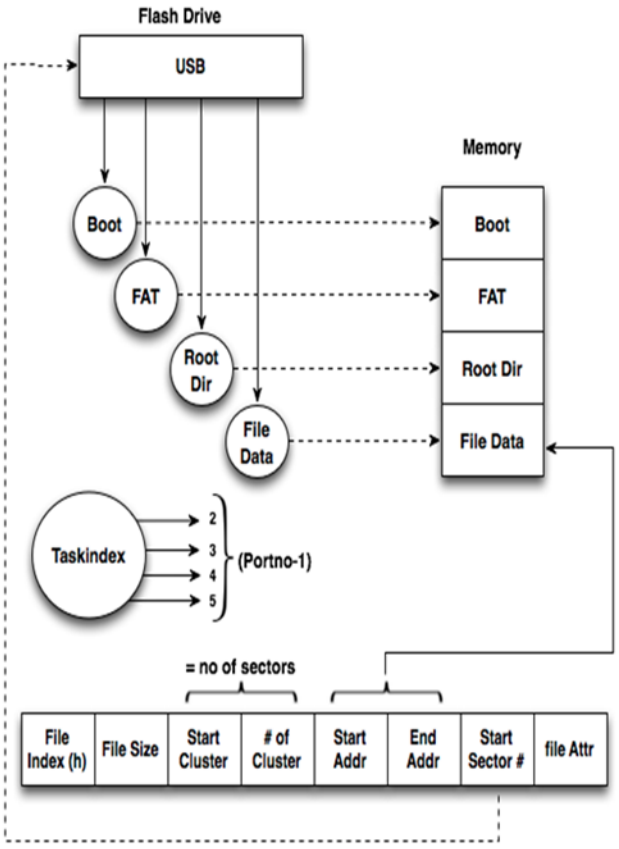

Figure 15. System architecture.

We found that the USB controller behaves differently depending on whether there is only one USB in operation or many of them plugged in. In the latter case, it requires a special reset known as mass storage reset. This is in addition to the operations as shown in Fig. 17.

A mass storage requires a sequence of USB operations test unit ready, read data, write data, clear feature, and sense data. The single USB task will go through each device and perform read or write operations as needed by an application. This task will stay in the loop until it is terminated by the user.

C. Task Structure

The task structure shown in Fig. 19 illustrates the integration of the mass storage system with the bare PC Web server application, which requires HTTP tasks. A Web server also requires resource files that are sent to clients. It may also use the SQLite database to provide dynamic content to clients. A USB task provides all USB interfaces to the user (it could be "n" tasks for "n" ports). A SQLite task manages all SQLite operations including user interfaces.

We did not address the integration of the USB file system or SQLite database files with the bare PC Webserver. However, the architecture of the mass storage system provides all the functionality needed for integration. The "Main task" is the main task that continually runs in the bare PC. When a network packet arrives, a "RCV task" runs to process the request. Similarly, when HTTP data has to be sent to a client, the "HTTP task" runs. Each task type has its own task pool created during the initialization process and kept in a stack. When a task is needed, it is popped from its appropriate task pool and placed in a circular list. The circular list tasks are processed on a first-come-first-serve basis. When a running task is complete, it will be pushed back on to its appropriate stack. When a task is waiting for an event, it is suspended and placed back in the circular list. This simple approach to managing tasks is used in the BMC paradigm. It is scalable as other types of task pools can be added in the same way.

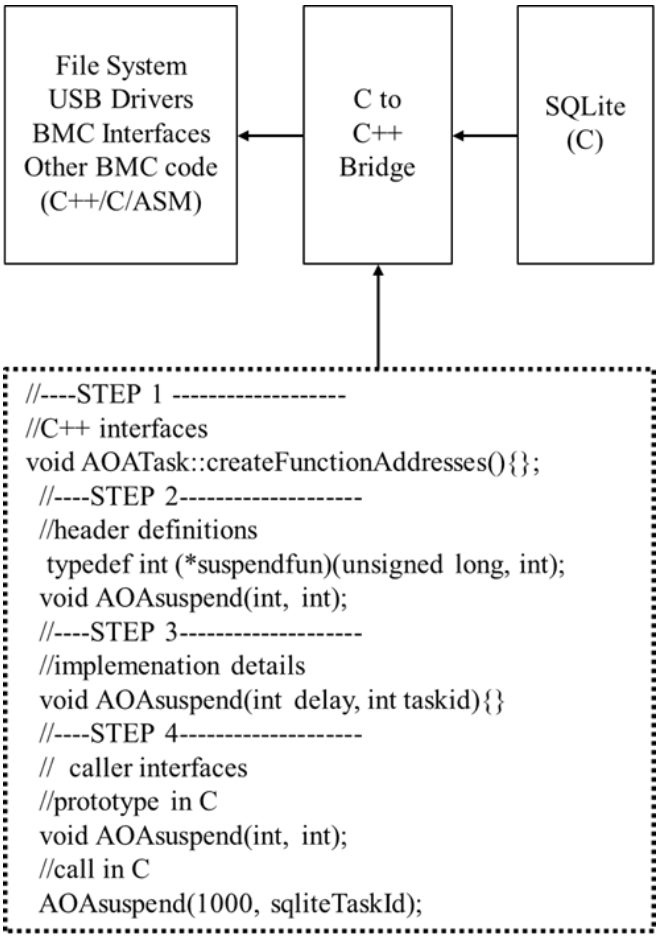

Figure 16. C to C++ bridge.

D. Class Flow Diagram

The mass storage system consists of three key class objects as shown in Fig. 20. The "fileobj" class provides a lean and efficient file API [1] [31] for bare PC applications. "USBFObj" consists of USB plug-play functions and interfaces to "fileobj". This object is managed by the USB file task. The file system API and all other interfaces can call "USBObj" interfaces for low-level USB commands such as test unit ready, read, write, sense, reset, clear feature, etc.

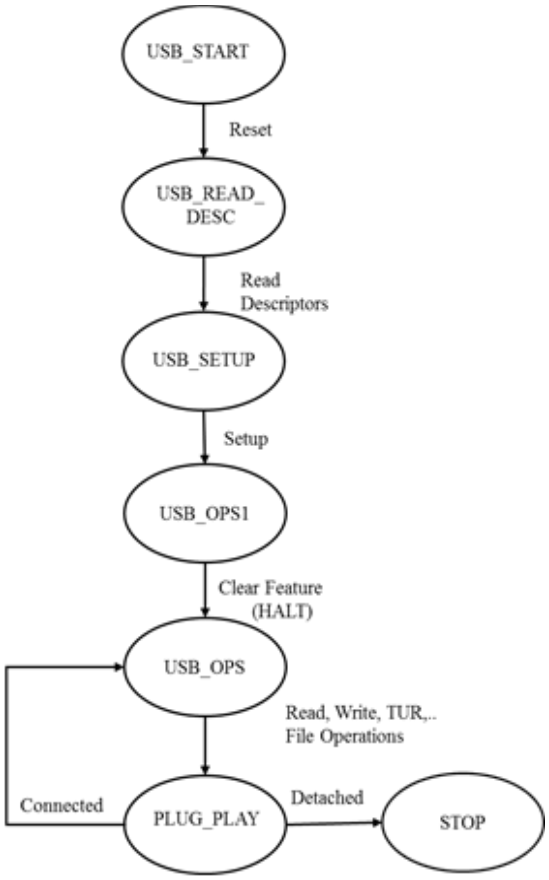

Figure 17. USB operation flow diagram.

The "USBObj", which is the USB device driver, communicates with USB controllers and devices. Each device has its own file system that is managed by the mass storage system. In the BMC paradigm, all the code needed for a given application suite is a single monolithic executable that runs by itself without the need for any external software or a kernel. Thus, a bare PC programmer has to manage all the intricacies of a given application suite. The application suite itself is independent of any external software and includes its own application and execution environment. A given bare PC application only carries the interfaces and code it needs (it avoids implementing unnecessary OS or kernel functionality).

E. Memory Map

In bare PC applications, the physical memory is managed by the application/system programmer. For a given physical memory, there is a need to organize a memory map at design time. Fig. 21 shows a typical memory map for the mass storage prototype. The first 1GB of memory is used for the Web server and other bare PC code including stack memory. The second 2GB of memory is used for USB file storage including SQLite database files.

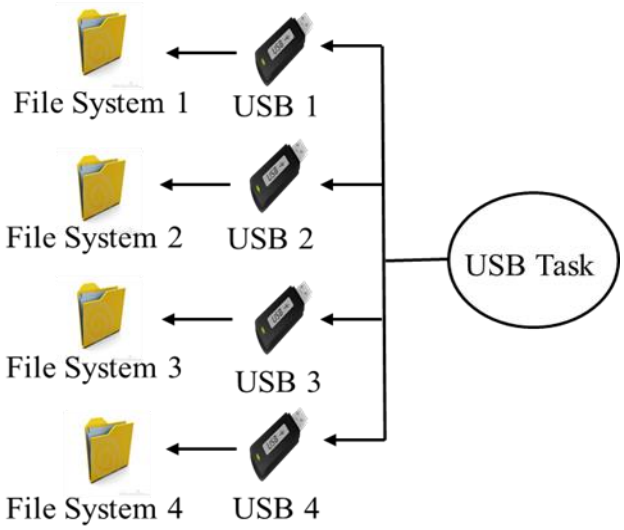

Figure 18. USB task diagram.

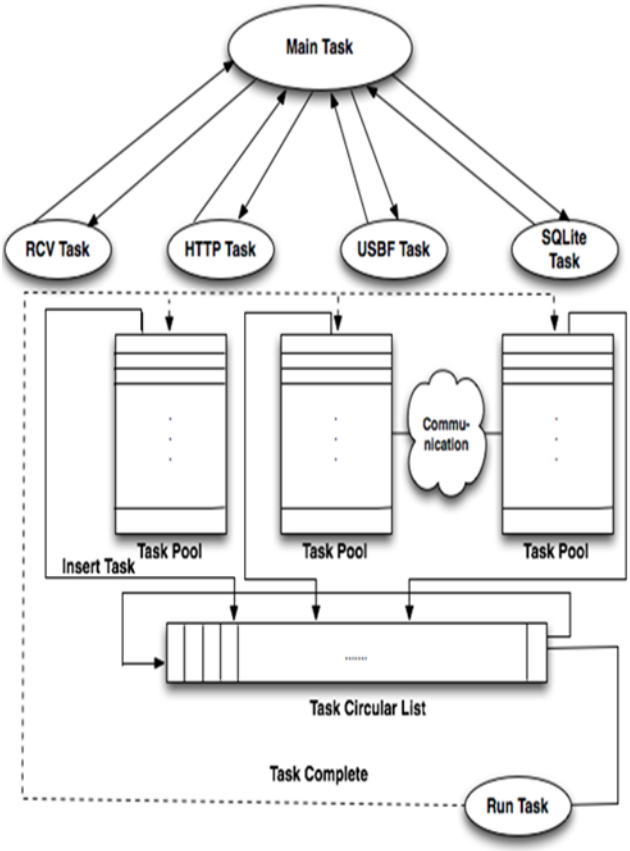

Figure 19. Task structure.

Two more GBs of memory can also be used for mass storage as needed. For four USBs, 256 MB storage is used for each USB (consisting of a total of 1 GB). 12 USBs can be mapped into a 4GB physical memory. When large memory is needed, the mass storage has to be swapped in and out of the USB devices.

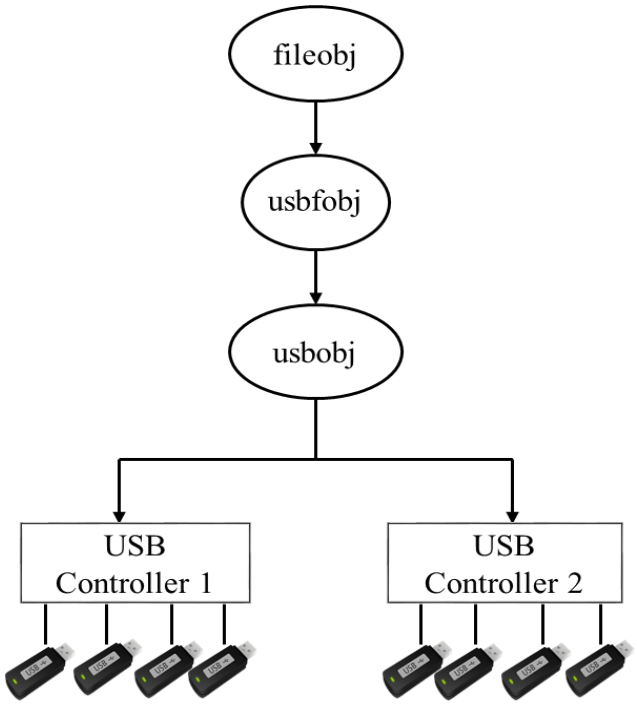

Figure 20. Class flow diagram.

F. Inter-process Communication

It is necessary to communicate between the SQLite and USB file tasks to invoke file operations such as flush, read and write. As shown in Fig. 19, the communication block is the inter-process communication element in the system. When SQLite is ready to flush, read, or write, it issues a command to the USB file task and waits synchronously until the command is complete. We use shared memory in real memory (< 1 MB) to communicate between these two processes. A single lock is used to implement this mechanism.

G. Implementation

The mass storage system was implemented in C/C++ with a small amount of assembly code for the direct hardware interfaces. The existing implementations of the bare FAT32 file system [1] [31], bare SQLite code transformation [26], and the bare Web server [2] [3] were used in building the mass storage system. The bare PC design is modular and extensible and allows new features and new applications to be added easily. The bare PC design methodology is described in [34].

VI. FUNCTIONAL OPERATION AND MEASUREMENTS

The mass storage system was tested on a Dell Optiplex 960 desktop with 2 GB Verbatim USBs. Four USBs were plugged in to the first controller and file operations were performed sequentially on the port numbers 3, 4, 5, and 6. File

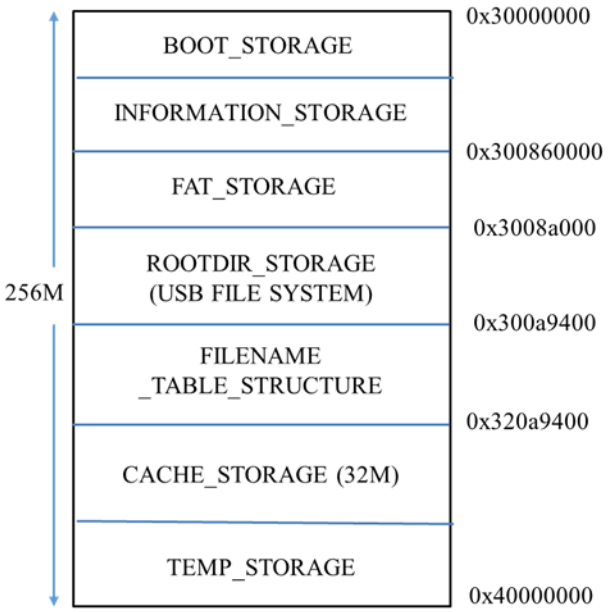

Figure 21. Memory map for each USB.

flush, read, write, and other file operations were tested to validate the mass storage system. SQLite database files were also stored on the above four USBs using four different database files. The read and write operations for regular files and the database files are same as they use the same file system. We varied USB file sizes from 1 MB to 30 MB to measure write and read timings. Fig. 22 shows write times using the file system and also using raw files.

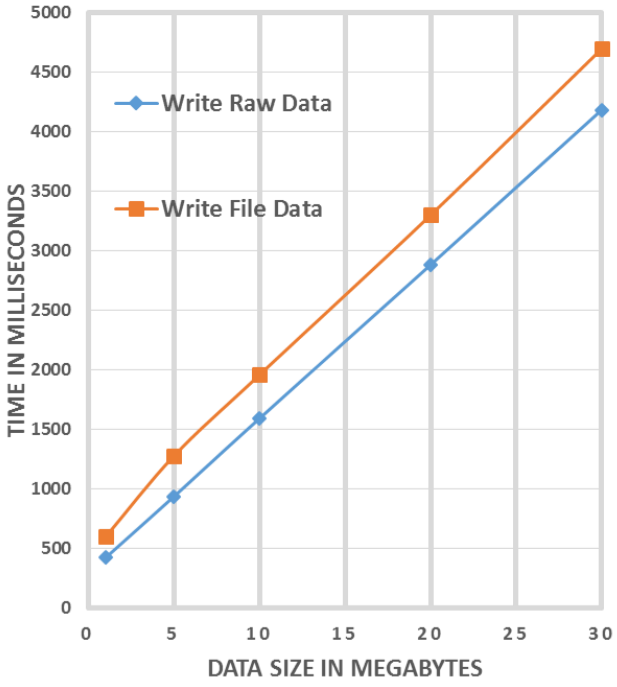

Figure 22. Write raw data/file.

A 30 MB file was written in 4.698 seconds. A 30 MB raw file (not using any file system) was written in 4.185 seconds. Thus, raw file writes can provide an approximately 12% performance improvement. This implies that bare PC applications should use raw files instead of a conventional file system to improve performance.

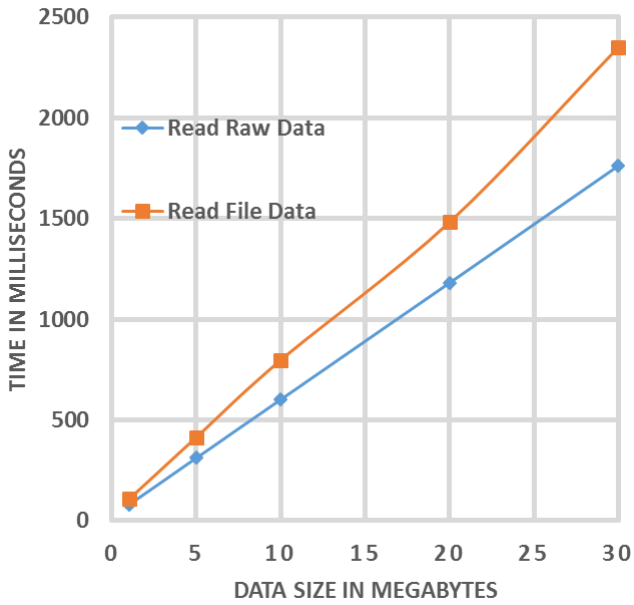

Figure 23. Read raw data/file.

In a Windows desktop, the same 30 MB file took **8.2** seconds to complete a write on the USB. This indicates that the bare PC file system has better performance than a Windows file system. In the bare PC, a 30 MB file was read in **2.351** seconds. The same file was read as a raw file in **1.762** seconds as shown in Fig. 23. This is a **33%** improvement in raw read versus a file system read. The bare PC systems are efficient and lean, and the footprint for executable files is small. The executable file size for this mass storage system is about 252 KB including the Web server and other bare PC code.

VII. CONCLUSION

We presented the architecture, design, and implementation of a mass storage system for bare PC applications. Large files and SQLite database files were used to demonstrate and test the feasibility of the system. Four USB flash drives were used to validate the design and measure basic performance. Timings for USB file write and read operations with large files were measured. Also, large raw file write and read timings were compared with the file operations. The results show that raw write and reads yield performance gains for bare PC systems. The mass storage system described in this paper is lean, simple, and scalable. It also has no OS-related vulnerabilities. As the code is simple and lean, it is easier to analyze for security flaws. The system is user-centric and runs on any x86-based architecture in bare mode.

We also presented a file API for bare PC applications. The bare PC 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 and mass storage system can be integrated with bare PC applications such as Web servers, Webmail/email servers, SIP servers, and VoIP clients. Future research could investigate the use of these systems for big data applications and cloud storage.

REFERENCES

[1] W. Thompson, R. K. Karne, S. Liang, A. L. Wijesinha, H. Alabsi, and H. Chang, “Implementing a USB File System for Bare PC Applications,” 12th Advanced International Conference on Telecommunication, 2016, pp. 58-63.

[2] 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.

[3] L. He, R. K. Karne, A. L. Wijesinha, and A. Emdadi, “A. A Study of Bare PC Web Server Performance for Workloads with Dynamic and Static Content,” 11th IEEE International Conference on High Performance Computing and Communications (HPCC), 2009, pp. 494-499.

[4] P. Appiah-Kubi, R. K. Karne, and A. L. Wijesinha, “The Design and Performance of a Bare PC Webmail Server,” 12th IEEE International Conference on High Performance Computing and Communications, (HPCC) 2010, pp. 521-526.

[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 Annual IEEE International Computer Software and Applications Conference (COMPSAC), 2009, pp. 480-485.

[6] 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.

[7] 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.

[8] R. Yasinovskyy, A. Alexander, A. L. Wijesinha, and R. K. Karne, “Bare PC SIP user agent implementation and performance for secure VoIP,” International Journal on Advances in Telecommunications, vol 5 no 3 & 4, 2012, pp. 111-119.

[9] 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.

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

[11] 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.

[12] "The OS Kit Project," School of Computing, University of Utah, SaltnLake ,UT, June 2002, http://www.cs.utah.edu/flux/oskit.

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

[14] 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.

[15] S. Soumya, R. Guerin and K. Hosanagar, "Functionality-rich vs. Minimalist Platforms: A Two-sided Market Analysis," ACM Computer Communication Review, vol. 41, no. 5, pp. 36-43, Sept. 2011.

[16] 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, pp. 50-55.

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

[18] R. Russon and Y. Fledel, "NTFS Documentation," http://dubeyko.com/development/FileSystems/NTFS/ntfsdoc.pdf. [retrieved: April 8, 2016]

[19] R. Shullich, "Reverse Engineering the Microsoft ExFAT File System,"https://www.sans.org/readingroom/whitepapers/forensics/reverse-engineering-microsoft-exfat-file-system-33274. [retrieved: April 8, 2016]

[20] 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.

[21] 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.

[22] 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.

[23] J. Larimer, "Beyond Autorun," Exploiting vulnerabilities with removable storage," 1–66, Jan. 2011. https://media.blackhat.com/bh-dc-11/Larimer/BlackHat_DC_2011_Larimer_Vulnerabiliters_w-removeable_storage-wp.pdf. [retrieved: April 8, 2016]

[24] 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.

[25] SQLite, http://www.sqlite.org/download.html. [retrieved: April 8, 2016]

[26] U. Okafor, R. K. Karne, A. L. Wijesinha and B. Rawal Transforming SQLITE to Run on a Bare PC," 7th International Conference on Software Paradigm Trends, 2012, pp. 311-314.

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

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

[29] 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]

[30] SCSI2.0 Specifications, http://ldkelley.com/SCSI2/index.html. [retrieved: April 8, 2016]

[31] 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.

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

[33] "How to mix C and C++," The C Programming Language, https://isocpp.org/wiki/faq/mixing-c-and-cpp. [retrieved: April 8, 2016]

[34] G. H. Khaksari, R. K. Karne and A. L. Wijesinha. "A Bare Machine Application Development Methodology," International Journal of Computers and Their Applications (IJCA), Vol. 19, No.1, March 2012, pp. 10-25.