IDE Driver and RTLFS file system for RTLinux

Alejandro Lucero, Vicente Esteve, Ismael Ripoll and Alfons Crespo
Universidad Politecnica de Valencia
Camino de Vera s/n, Valencia, Spain
{alucero, vesteve, iripoll, acrespo}@disca.upv.es

Abstract

RTLinux does not support direct access to any kind of permanent massive storage system, and in particular IDE hard disks. When a RTLinux thread has to store data on the hard disk, it has to use the Linux services. The usual way of doing the transfer is by means of RTFifos: a rtl-task sends the data to Linux through an RTfifo, and then, a Linux process writes this data on the hard disk.

This indirect method is slow, unpredictable and inneficient, since it is executed by Linux (background). Moreover, in some applications, like continuous media applications, it is required a certain degree of predictability. Therefore, for multimedia applications is more convenient that rtl-tasks have access to the hard drive, allowing higher determinism. To achieve this goal is necessary: 1) To port the IDE driver to RTLinux, 2) implement a real-time filesystem, and 3) implement a real-time disk scheduler.

The goal is to provide with a full "block layer" to RTLinux making the design scalable to allow future storage devices drivers implementations and offering a known interface to work with.

1 Introduction

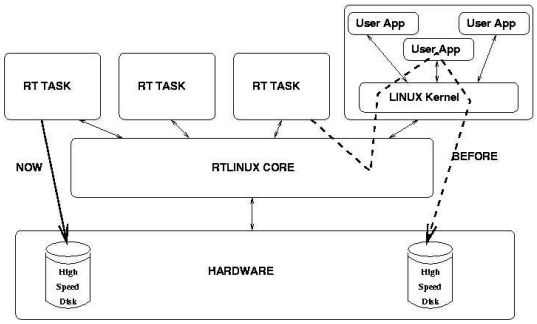

FIGURE 1: *RTLinux storage path*

RTLinux[Yodaiken] is a real time microkernel which implements the basic structure that allows RT tasks to execute with temporal requirements. RTLinux strength is its minimal size together with the Linux combination, since its philosophy is to leave to Linux every task without temporal requirements, as graphical interface, database accesses, remote comunications, data logging, etc. However, in some enviroments this architecture is not useful, for example when data logging must guarantee to store all the data produced by RT tasks. If we assume the storage system achieves the performance needed, data logging should be efficient, but the problem here is storage system is a resource shared between RTLinux and Linux. Although the theoretical disk performance would allow data logging to work, Linux interference avoids the determinsm needed.

In the last years a lot of research about real time in storage systems has been done. In Shriver[Shriver] are presented the most important advances, giving information about the current problems and how can be resolved following analytical methods to understand storage systems behaviour. However, this work has a more modest aim: to advance forward the determinsm when RT tasks use the IDE disk. Full determinsm is a goal hard(impossible?) to get, at least with systems using IDE controlllers into PCI buses together with ethernet cards, VGA and USB controllers, etc. Indeed, magnetic disks suffer some drawbacks that make it hard to know their behaviour, as thermal recalibrations or bad blocks ocurrences. Moreover, internal disk caches increase complexity. By other hand, permanent storage is a facility too powerful to forget, and there are no alternative technologies for replacing them in the next decade [Thompson]. We think this work can be a first step to obtain hard real time with magnetic disks, and this can be the base for future research.

In this first implementation of the RT IDE driver and the RTLFS file system, we need a disk for RT tasks distinct from the Linux ones, and placed in dis-

tinct IDE controller. Obviously, this is the ideal situation, but although this enviroment can be suitable in some systems as specific media servers, we should be able to work with a single disk shared by Linux and RTLinux. In these days arrive news about the possibility of introduce hard disks in mobile phones, since these devices are increasing their capabilities (and their memory requirements growing in parallel) and it does not seem a practical solution to design the phone with two hard disks. An interesting goal is to be closer to determinism when RT tasks access IDE disks, even if they share the same disk with Linux.

The first step forward higher determinsm is the implementation of a IDE driver for RT tasks, which is explained in section 2. Once we have virtualized the disk HW with the IDE driver, RT tasks can read and write data using it directly, but if we want to make things easier, we need a file system to avoid us to know where to read or to write data. The implementation of a file system with some requeriments to facilitate determinism with high performance is explained in section 3. The design and implementation of a block layer allowing to work with several partitions and disks concurrently is explained in section 4. And finally, in section 5 we present the conclusions and the future work.

2 RTL IDE Driver

The functionality of the RT IDE driver is not different from Linux ones, since the common goal is to hide the hardware complexity to the user. The first task in the porting is to evaluate the real necessity of using a distinct driver than Linux. This decision was taken based on the complexity of the Linux IDE driver update.

2.1 Implementation Issues

The main problem is Linux IDE driver is very related with the Linux Buffer Cache, a global structure that allows Linux to achieve a high throughput. As we will see in the next section, for our necessities we don't need this Buffer Cache, and indeed, if we would need it we could not share this global resource with Linux, since it would lead to priority inversion. As we can use the Linux IDE driver directly, it is necessary to make some modifications to the driver sources. However, we have considered these changes have more drawbacks that advantages, mainly due to the complexity to understand the driver functionality once the changes are introduced and the effort needed to maintain the changes between Linux versions. Moreover, an important drawback of using the Linux driver with updates is it slows down the development due to the instability of the system, since we are making changes in a critical component for Linux normal behaviour. These reasons convinced us to develop a distinct IDE driver for RT tasks.

Once we have taken the decision of a new RT IDE driver instead of the Linux one, other decisions about driver functionality are necessary. First of all is the question of supporting DMA functionality. The use of DMA(Direct Memory access) in real time systems could lead to unexpected behaviours where deadlines are missed, since capable Master DMA devices own the memory bus when they are working, blocking the cpu access to the memory during this period. However, DMA is a feature too rich to be ignored so, the use of DMA versus programmed I/O will be a user option and not an implementation decision. However, when the same disk is used by Linux and RT tasks, the DMA should not be available for Linux, to avoid non real time processes to damage real time ones. Another point is a feature inside DMA: the capacity to use scattered memory blocks in the same operation. This feature is very useful for file systems in general purpose operating systems(GPOS), and related again with the Linux Buffer Cache. Although this is a powerful functionality for GPOS, it has not so importance in our design, since memory is managed more restrictively (at initialization time) within the RTLinux approach. When developers ask for memory at inicialization time, they will use the get_free_pages Linux kernel function that returns a pointer to a contiguous memory block. Then, this advanced DMA capability is not needed and in consecuence, not implemented.

The way RT IDE driver is used has a direct impact in performance with DMA functionality. We have commented previously the Linux Buffer Cache goal, and explained why this structure is not necesary for real time tasks. However, using an intermediate buffer as Linux Buffer instead of directly the user buffer has the advantage that the data passed to disk is always aligned (a DMA requeriment is that memory pointer must be word aligned). For example: if one RT task with a 1MB data buffer decides to write 5 bytes of data, the memory pointer used by the RT IDE driver will point to the begin of the user data buffer. As the data buffer is created with the get_free_pages Linux function, the first byte of the buffer is word aligned. If later, the same RT task writes another 10 bytes, the user data pointer will point to the sixth byte of the user data buffer, which is not word aligned, leading to a bad DMA functionality. To avoid this to happen the RT IDE driver checks the alignement of the data pointer making a copy to a internal driver buffer when needed. Obvi-

ously this is a performance penalty and it should be avoided when possible.

One important point is how the RT IDE driver is initialized. At Linux boot time some structures are initialized as ide_driver_t and ide_hwif_t, which are related to IDE disks and IDE controllers. Our RT IDE driver uses these structures, together with other related with PCI interface and DMA tables. One posibility would be that RT IDE driver could initialize its own structures, avoiding the Linux dependency and then it would be ready to be used in the RTLinux Stand-Alone [Esteve].

2.2 Summary

We have implemented a IDE driver for RT tasks with the possibility to use DMA functionality, with some limitations related with the Linux IDE driver, but which don't affect RT necessities. Once we have this driver, RT tasks can use the disk directly instead of through the indirect path with fifos. The driver implements the POSIX functions: `open`, `read`, `write`, `close` and `llseek`.

3 RTLFS file system

Once RTLinux has its own IDE driver, the next step is to facilitate the management of free disk space and some way to know where data was written: these are tasks for a file system.

The design of a file system is not a simple thing, and it is driven by some specific goals together with the implicit ones commented before. For example, a file system used in a GPOS have a distinct goal than a parallel or a distributed one, used in scienfitic fields. Some important keys as internal and external fragmentation, which are common issues in operating systems, and arises in memory and disk space management. Some characteristics of file systems are related with disk space maximization, reliability, fast recovery, higher performance and others. But meanwhile file system goal in a GPOS is to achieve the best throughput, a real time OS must guarantee deadlines of real time tasks will not missed, so this implies that in the design of a file system to be used in a real time system, some of these characteristics, although important, are not the main goal (as disk space maximization and internal-external fragmentation).

Another important point in the file system design is what kind of access pattern will have the tasks supported, and then to make the design properly to adapt those kind of tasks. We have based our design in video streams support, wich have a well known pattern. Other usual RT tasks as data adquisition can gain benefit of this design.

3.1 Preliminary Study

As previously commented, the main goal of this component is to provide a file system to store media data. This point, along with the specific characteristics of a real time system leads the design. In this section we explain what are the main characteristics the file system should exhibit, and in a later section it is presented the specification of the design in detail. The discussion is not only about the file system, but about the full block layer which includes global system structures.

Next are outlined the main characteristics of an embedded operating system (RTLinux) and the applications requirements:

CONCURRENCY RTLinux is not a general purpose operating system as Linux, where is usual to have lot of tasks working at the same time. The expected RTLinux workload will be just a few threads, possibly one or two. Obviously, mechanisms to share the file system between several tasks is a must, but it is important to fix the number of concurrent tasks supported since it is necessary several structures per task and per open file.

SIMPLICITY The key in RTLinux is simplicity: it is not necessary to build a full real-time operating system with all kind of functionalities as a general purpose operating system. In this way, the RTLinux core is easier to maintain. If we don't want to break this approach, the file system design must be simple, avoiding complex implementations and features that are rarely used. We are not thinking in designing a file system for all kind of requirements, only to support media streams, which have a known access pattern.

SPACE ALLOCATION How the data is allocated in disk is one of the main functions of file systems. There are two points :

1. how data is allocated on the disk
2. how metadata is managed

Metadata is information about the file system: super block has global information of the file system; inodes are related with files and have information as size of the file or pointers to data blocks; free blocks list or bitmaps are used to manage free space, etc. File systems decisions about metadata (which data structures to use, and where to allocate them on the disk) are

important for file system performance. For example to try allocate the inodes of a file as close as possible of their data blocks. Other decision is if metadata must be written sync or asynchronously which has a direct impact on reliability. We need a file system that can be returned to a consistent state after a crash and to avoid metadata writes overhead can degrade performance of the file system.

The allocation policy is different depending on the feature that we want to optimise. For example, in general purpose operating systems, file systems are designed considering that most files are small, typically is a few kbytes, and that the lifetime of each file could vary from a few seconds to several months or years. The file size is important to avoid excessive fragmentation which leads to a poor usage of the disk, so general file systems use a minimal allocation unit of 1-4 Kbytes (1-8 sectors). The smaller the allocation unit is, the bigger is the metadata required to manage it, because there are more blocks (units) to handle. This implies more resources wasted and higher cost when searching through (or a complex structures to minimise the search cost).

In systems designed to collect data, the requirements are different since data will be stored for a long time (data will be processed afterwards) and will no be modified in a short space of time. Obviously, taking into account this characteristic, the approach to design the file system is different. As we focus to support the storage of media streams (large files) we can remove the complexity introduced by buffer caches and large blocks maps. Concepts as internal or external fragmentation are important for general purpose systems, but are not so critical in these kind of applications.

A critical point is how to search into the file system structures. This search must be optimized, avoiding complex data structures as AVL's used in current file systems as XFS [Trautman]. In some situations, as opening a file, the delay searching through the tables can be allowed (in our target), but the delay searching for free space is necessary to optimise.

We have discussed the design considering the disk access pattern, but it is critical to know how disks work to improve the performance. The minimal allocation unit used by general file system has sense in the global view, but this can leads to a excessive disk head movement since physical blocks can not be consecutive for a file. We need to minimize the disk head latency as much as possible since it is critical to achieve good performance.

BUFFER CACHE Disk latency is a bottleneck since CPU's speed began to grow as Moore's law predicts, and meanwhile disk were, and still are, restricted to a minor growth rate mainly due to the mechanical components inside. Operating systems use a tecnique to avoid this problem called Buffer Cache. This is a general memory buffer to allocate disk blocks temporally in main RAM memory that tries to avoid unnecessary disk requests.

Buffer Cache algorithms tries to exploit known disk access patterns as the short lifetime of some files (sometimes just seconds) and local and temporal references. These access patterns are valid for general purpose systems and applications. Some of the main characteristics of buffer caches are:

1. Read ahead, based in local references: when a disk block is requested for read, then the next contiguous blocks of that file are read and stored in the buffer cache too.
2. Flexibility for disk policy: as write operations are delayed, the final disk scheduler can rearrange them to minimise disk head movements.
3. Extra copy from user buffer to system buffer
4. Inconsistent state if a crash happens: data (and metadata) of the buffer cache still not written into the disk is lost when a crash happens, therefore there are more chances to lose more data.
5. Low size block to allow an easy management of the buffer: if these blocks are large a lot of resources are wasted when a few bytes are requested.

Points 3 and 4 are drawbacks and 5 is in conflict with the decision taken in the previous point about space allocation (large extents). Indeed, since other characteristics of general purpose operating systems as short life time of files or local and temporal references are not applicable for our purposes, it is not necessary in our design a buffer cache, therefore we can avoid the implementation (However, as we will see in our implementation we need a 512 bytes cache per file for performance).

RELIABILITY As reliability we mean to obtain a consistent state of the file systems after a crash. Usually, file systems changes are made in structures allocated in memory which are eventually written to disk. If a crash happens before these changes are written to disk the file system state is not consistent.

Reliability is very related with the design of the file system. Log (or journal) structured file systems [Ousterhout] were designed to provide a fast way to recover data when a crash happens, which is a drawback with ext2 Linux file system [Card], the first Linux file system implementation. But, with the log structured file system approach, reliability is achieved adding performance and resources penalty, along with a high complexity.

As one of the characteristics cited previously was simplicity, the reliability must not add excessive complexity to the design. And, of course, reliability should not achieved by loosing performance (only a minimal overhead is tolerable).

USABILITY Although the indirect path (thru Linux processes) followed until now by RT tasks to read or write to/from disk was very "tricky", it has as strong point the possibility to work later with the data using Linux tools. Then, the file system used in RTLinux should also be used in Linux to work comfortably with the broad possibilities offered.

USER BUFFER ALLOCATION RT tasks will use the file system with the standard read and write POSIX functions. These functions needs a buffer parameter, which is a pointer to a memory zone that will be used by the file system.

In the Linux approach, user buffer data is copied into the kernel buffers using buffers heads objects. This is a technique to improve performance, and works fine with general file systems due to the locality and temporal references concepts, storing data temporally in these buffer. Our expected workload will not exhibit temporal reference disk access, therefore buffers head are not necessary, avoiding to waste resources and the double data transfer, one from user buffer to kernel buffer and other from kernel buffer to disk. So, data buffer pointed by the read or write function is used directly by the device. This is possible because as we will see in the next point read and write functions blocks the caller task, so does not exist the danger of task reusing the buffer before the end of the request.

Taking into account that the RTLinux memory allocation (OCERA DYNMEM component [Ocera]) do not handle DMA address ranges properly [1], the buffer allocation for read and write functions by a task must be done at initialisation time.

READ AND WRITE FUNCTIONALITY Should a read or write call block? As we want to follow the POSIX standard for read and write, these functions must block the caller task. But, if we want to keep tasks inside the real time we need disk behaviour to be deterministic.

Otherwise, to make sure a task reading continuosly from a device does not lose data samples, the task division paradigm usual in UNIX must be followed: one task reads data and other writes data to disk, sharing a buffer. We assume disk can support the bandwith required.

TRUSTED ENVIRONMENT Operating systems use file attributes to check access rights, along with attributes of the process accessing the file. This is a necessity in untrusted environments where the operating system has to enforce the access policy.

Since the RTLfs file system will be used in trusted environments, where all the code is written and controlled by the end user, there is no point to implement any kind of access protection. Indeed, files atributes are based on the idea of users and groups, which is not valid in RTLinux.

Another important aspect from a security point of view is how new blocks are passed to files. Assuming a trusted environment file system assigns blocks to files without deleted previous data (disk data blocks are not zeroed).

Parameters checking are very restricted in untrusted environments to avoid the bad use of the functions by a malicious process. As a trusted enviroments is assumed the checking can be more relaxed, mostly related to catch programming bugs.

[1] current memory allocator manages a pool of memory which is not ensured to be in accessible address range of the DMA hardware. Next revisions of DYNMEM should consider this problem

3.2 Available Open Source Realtime FileSystems

As a previous step we have studied others file systems looking for how they match with the characteristics defined before, and evaluating the migration cost of that file systems to RTLinux.

We have found that the original file systems developed two decades ago fulfil the main points. It has sense as these first implementations were simple and executed in trusted environments, with a little disk capacity, and where disk latency was hidden by the lower performance of CPU's. This last point had motivated the main research of this field and the use of a general buffer cache. A simple file system implementation have the following characteristics:

- Sectors are grouped in clusters, which can be as long as disk capacity.
- Linked list using an index method is used to manage clusters (groups of sectors). In each entry of the index appear the next cluster (if exists) owned by the same file or a special character if unused. The metadata management full-fills the simplicity required.

The main drawback of this implementation is its inefficiency when it manages unused or spared clusters: the system must search through the index block table sequentially, what can lead to high variable search costs. This is a drawback to avoid in real time systems. On the other hand, traditional UNIX file system (in the UNIX first version) have a more complex block allocation structures: a linked list used for space management which resolves the problem.

The Linux file systems are very related with Linux structures, and their goals are different from ours. This along with other complexities usual in file systems designed for GPOS as file access attributes or file access times (if simplicity is a goal in the design we must avoid a lot of non critical features), influenced in the decision to implement a file system from scratch.

3.3 Real Time File System (RTLFS) Specification

Once analysed the characteristics that the file system should provide, this section presents the proposed design and the implementation that meets these requirements. Figure 2 shows the global structure of the file system:

We present the file system components with the main characteristics:

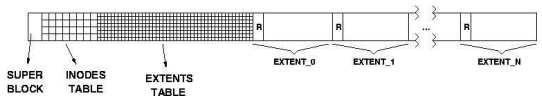

FIGURE 2: *RTLFS Global View*

SUPER BLOCK The super block contains information describing the layout of the file system. For example, the number of sectors for inodes and extents tables are stored here, along with the extent size (an extent is a large number of contiguous sectors). In our design, the super block have two important fields to manage free space: pointers to free lists of inodes and extents. This is where our design changes with regard to how other file systems search through the linked list.

INODES TABLE Inodes are structures used to manage metadata of files: size, mode, permisions, pointers to data blocks, etc. The decision to have a fixed number of i-nodes and extents length is to avoid complexity for data blocks allocation. In this way we can locate extents easily. The main drawback is the total amount of i-nodes the file system can have. In the current implementation the maximum number of sectors for inode table is 128, what is the upper limit a DMA operation supports. With this limit the maximum number of files allowed is 1638 (inode size = 40 bytes).

Obviously this is very low number for a general purpose operating system, but we think that it is enough for embedded real time systems. It's possible that some real time applications need more files but it does not seem the normal case.

Usually in UNIX implementations directories are files which data is a list of files owned by these directories along with a inode pointer per file. In our design the file name is inside the inode structure since we are not going to support directories: only a root dirrectory for the file system.

EXTENTS TABLE As i-node table, this is a fixed size structure created when the file system is formatted, along with the number of sectors per extent. In the current implementation the maximum number of sectors for the extent table is 128, as inode table. This sets the maximum number of extents to 16384 (an extent is a long).

Following a simple approach, this structure is a linked list using an index and is maintained

in memory to improve performance since the size is manageable thanks to the high number of sectors per extent. The next figure shows an example with a reduced extent table:

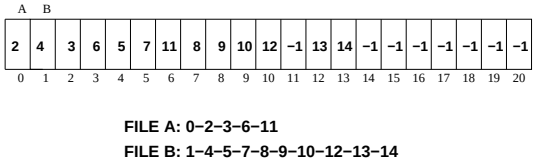

FIGURE 3: *Linked list used by Extents Table*

This approach has the advantage of the facility to found free extents when a file is deleted. As blocks owned by a file are linked, is easy to know what is the next free extent just following the links. This follows a simple algorithm to manage free extents, first found first served, althoug other algotithm could be used adding more complexity to the design. A Pointer to the head of the free list extents is stored in the super block.

EXTENTS RELIABILITY The problem with reliability is how to maintain the file system consistent when system crashes (power failure, critical application bug, etc.), and is very related with how metadata is written to disk. File systems designers had taken different approaches to solve it: BSD file system writes metadata synchronously to disk, meanwhile Linux file systems write metadata just in buffer memory, and later aynchrously to disk.

The metadata problem is due to the disk latency and the fact that metadata blocks may not be close to where current data is being written into disk, therefore it implies a large head movements to other disk zone. We have this problem with our design, as inodes and extents tables are fixed in the first sectors of the partition.

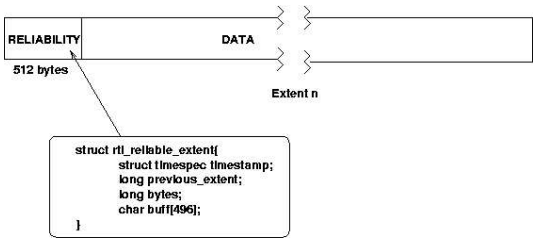

FIGURE 4: *RTLFS Reliability Approach*

We solve this using the first sector of the extents for reliability. The information written in the first sector is a *rtl_reliable_extent*:

The *timestamp* field is used to know the state of the extent related with the superblock. At recovery time, only extents with a timestamp newer than the superblock timestamp are processed for recovery. The *previous_extent* field helps to rebuild the extents list of a file. And in the *buf* field is stored the inode object of the file together with the super block of the file system at this moment. With this information is possible to get a consistent state of the file system.

PERFORMANCE DECISIONS We have explained that a general buffer cache is not necessary for our purposes, mainly due to the temporal reference characteristic is not a feature of the expected workload. However, the use of a minimal buffer cache has some advantages that could improve the performance of our file system.

The explanation is easy to understand with an example: a process writing 500 bytes of data every 200 ms. Since the disk sector size is 512 bytes, the request does not fill a sector; and when the next request arrives, data will be placed filling the last 12 bytes of that sector and 488 bytes of the next. This behaviour implies the first sector must be read from disk before the second write, because we don't have a general buffer cache. Only in this very common situation the temporal reference is true.

We solve this problem using a 512 bytes cache by file object that will be used when request are not sector aligned.

3.4 TOOLS

mkfs.rtlfs Used from Linux console, it builds a rtlfs file system on a device.

rtlfs.chk Used from Linux console, it checks and repairs a rtlfs file system.

Linux Module The main advantage of RTLinux is the possibility to use linux tools. As we want to maintain this, a file system module has been developed. Linux can mount a rtlfs file system when is not mounted by RTLinux, but can do it in READ mode only.

4 RTL Block Layer Architecture

The Block Layer is the structure needed to access distinct storage devices following a common interface. The user must be unaware of what device and what file system are being used.

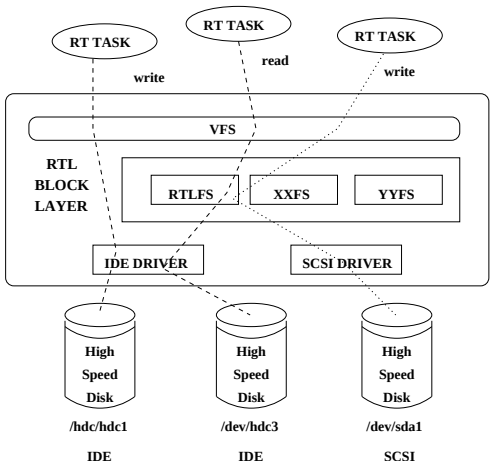

FIGURE 5: *RTLinux Block Layer Architecture*

In the figure we can see the idea behind: tasks use standar functions as POSIX `read` and `write` operations, and it is the block layer which links the operation with the right device, using the read or write operations of the appropiate file system.

In the implementation, we have followed the RTLinux philosophy about standards and we try to offer a popular (standar) mechanism to work with. This means the option "to mount" file systems following the Unix approach, and to use virtual file system(VFS) abstraction as found in modern operating systems.

The new architecture allows to use the device directly or using a file system (rtlfs file system or other future file systems). The object-oriented approach used by Linux in the VFS is used to achieve this point.

4.1 Implementation issues

The implementation of the Block Layer needs new structures and some updates to the existing ones. For example the *rtl_file* object, declared in the *rtl_posixio* RTLinux module, needs one aditional field to store the minor number of the device related with the file. Indeed, to support more than one disk and controller is necessary to add structures similar to the Linux *ide_hwif_t* and *ide_hwgroup_t*.

We have changed the RTLinux open function implementation, introducing a new layer to check mounted file systems. The full process is as follows: if the file name used in the *open* function does not follow the /dev/filename rule, then the *check_for_mounted_file_systems* function is called. This new function extracts the initial path of the file name (which must follows the unix path convention of names between slashes) and then, it searchs into a structure called *rtl_mount_list* where are stored the mounted file systems. Instead of */dev/filename*, the parameter will be */virtual_dir/filename* where *virtual_dir* is only a identifier (**no structure behind**). A mount point will consist in a virtual directory, a device (*kdev_t*) and a pointer to the specific file system operations. All of this information is stored in the *rtl_fs_list* and *rtl_mount_list* arrays.

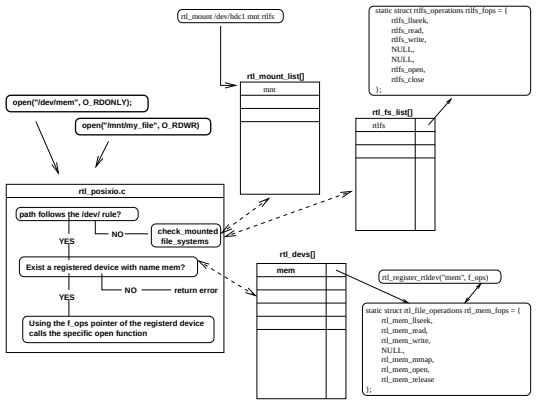

FIGURE 6: *RTLinux Open function update*

In the figure we can see the steps: the open function checks the path, and as it is not a path following the rule /dev/filename, it calls *check_for_mounted_file_system* function. This function searchs the *rtl_mount_list* array until a mount point is found matching /mnt/. When is found, the major and minor numbers of the rtl_file are set to the same values the mount point has (later these major and minor numbers will be used by the RTL IDE driver to use the right disk and partition), and the *file_operations* field in the *rtl_file* will point to the file system operations of the file system structure. The specific *open* function of the rtlfs filesystem will be called now.

4.2 Using the disk directly

If the rtlfs or other file systems are not necessary, RT tasks may access the disk directly using the old method, that is following the /dev/device rule. This is possible because during initialization the RT_IDE driver has registered the IDE disks detected during Linux initialization. Some precaution is necessary here, since disk partitions mounted by Linux should not be used.

One problem appears in how RTLinux registers devices. The *rtl_posixio* module implements the registration function using an indexed list by *major* number. By other hand, each IDE controller supports two IDE disks, which must be syncronized since they share the same hardware resources and the same irq line, and Linux uses the same major number for disks in the same controller. We need a the minor number of the device to differentitate the disk in the same controller, but we can register devices in RTLinux with the same major number. By now, this drawback when IDE disks are used directly, and only one disk by controller can be used in this way. This is not a restriction when rtlfs is used instead.

4.3 Tools

rtl_register_fs: Used to register RT file systems.

rtl_mount: Used to link file systems with devices partitions. This command needs the partition name, the virtual mount point and the file system type. Previously to use it, a rtlfs file system should be created in the partition, using the *mkfs.rtlfs* tool. By now, this tool is used from Linux console, using RT fifos to communicate with the RT Block Layer.

5 Conclusion and future work

The main goal was to advance forward higher determinism when IDE disks are accessed by RT tasks. We have done some steps needed to achieve this goal, as a RT IDE driver, a file system with a design aware of real time necesities, and a block layer, needed to work with distinct devices and file systems at the same time. A specific disk scheduler has not been implemented, although a primitive scheduler based in the tasks priorities is used by now. By other hand, a current limitation is Linux and RTLinux must work with distinct IDE disks and controllers.

The simplicity and performance were main keys in the rtlfs design. Usually, these keys are hard to obtain together, but we achieved them with the cost of suboptimized disk space. As we commented before, the goal of GPOS is to maximize the resources, meanwhile the goal of a real time operating system is to do the tasks before deadlines. RTLFS allocation policy could be improved if we would have a full knowledgement of disks behaviour. Moreover, to obtain hard real time when disks are used needs this knowledgement. This is one important goal for future work, and we think we may use this information to design and implement a real time disk scheduler. However, using this knowledgement to improve the file system design is a point to discuss, since better disk behaviour knowledgement can allow better allocation policies but with the cost of higher complexity, and it can limit the real time goal.

We have studied the disk sharing by Linux and RTLinux, which is a limitation in the current version. We have implemented a first solution trying to minimize the Linux interference to RT tasks, and we hope to present this work along with a more sophisticated disk scheduler soon.

We think this is a valuable work to be used for future research, and even to be used now in some real time systems with minor modifications, if the drawbacks exposed can be accepted.

References

[Card] Remy Card, Theodore Ts'o, Stephen Tweedie, *Design and Implementation of the Second Extended File Sytems*

[Esteve] Esteve, Ripoll, Crespo, 2003, *Stand-Alone RTLinux-GPL*, Real Time Workshop, 2003

[Ocera] www.ocera.org

[Ousterhout] Ousterhout, Rosemblum, 1992, *The Design and Implementation of a Log-Structured*, ACM Transactions on Computer Systems, Feb 1992

[Shriver] Shriver, 1997, *Performance modeling for realistic storage devices*, PhD thesis, Department of Computer Science, New York University, New York, NY.

[Thompson] Thompson, Best, 2000, *The Future of Magnetic data storage technology*, IBM J. RES. DEVELOP. VOL 44 NO.3 MAY 2000

[Trautman] Philip Trautman, *Scalability and Performance in Modern Filesystems*, www.sgi.com/software/xfs

[Yodaiken] Yodaiken, Barabanov, 1997, *A Real-Time Linux* , in Proceedings of the Linux Applications Development and Deployment Conference (USELINUX), Anaheim, CA. The USENIX Association.