

Table of Contents

1. RTLinux Disk scheduler and file system (<i>RTLfs</i>)	1
Summary	1
Description	1
Layer.....	1
API / Compatibility	2
Dependencies.....	2
Status.....	2
File System Description.....	2
Available open source realtime filesystems.....	6
Real Time File System (RTFS) Specification	6
Features.....	9
Validation Criteria	9
Tests	10
APENDIX A	10

Chapter 1. RTLinux Disk scheduler and file system (*RTLfs*)

Summary

Name	Disk scheduler and file system
Description	Disk scheduler and hard real-time filesystem design and utilities.
Author/s	Alejandro Lucero
Reviewer	Ismael Ripoll
Layer	High level RTLinux.
Version	0.1
Status	Testing
Dependencies	Relies on the RTLide component.
Release Date	M3

Description

The aim of this component, together with the IDE Device Driver component, is to provide a full block layer for disk storage devices to RTLinux. Nowadays there is a necessity for real time tasks to storage media streams without loosing any sample or frame.

This block layer implementation will consist of three distinct elements: 1) the file system, 2) the disk scheduler and 3) the disk device driver (See the IDE driver documentation). Although we implement these elements separately to support future developments of only one of them, they are not completely independent: for example in the file system design we need to know what mechanism supports the disk device (as DMA) to take advantage of it, or the disk scheduler need information of which are the response times of the disk device to take decisions to achieve the requirements.

This component defines and implements a new file system designed to provide hard real-time (specially designed to serve multimedia applications) and robust recovery. The current disk scheduler sorts the requests by the priority of the calling thread.

Most of the effort has focussed on the file system structure, and not on the user interface.

Layer

Current version of this component do not change the RTLinux core, it only adds new functionalities. Therefore, it is a High-level RTLinux component.

API / Compatibility

The interface between the component and the system is through POSIX standar for file systems. Every read/write request from a RTL thread will use the POSIX `read` and `write` calls, and these functions will sent the request to disk scheduler using a well-defined interface between the file system and the disk scheduler which is not in the user domain.

Accessing File system from RT tasks is through the POSIX-IO interface defined in `rtl_posixio.h` and implemented in `rtl_posixio.c`. Some changes have been done in these files: the `open` function has been modified to allow the use of the `rtlfs` file system. `Open` function searches the RTLinux devices structure, so a new device has been added: the `rtlfs` device. When a user needs to open a file must use the full path `open("/dev/rtlfs/filename")`.

The file system is Linux compatible, since we have developed a Linux file-system module for this possibility, that allows Linux to mount the file system in *READ Only* mode. There is a restriction to use this functionality: Linux can not mount the file system while it is mounted by RTLinux.

Dependencies

It is strongly related to the OCERA IDE device driver component, since only this driver has been implemented. Current implementation do not provide a clean layer between the IDE driver and the file-system. It will be provided in next release.

Status

There is a first final version of the component with a POSIX file system implemented. The disk scheduler sorts request by thread priority, being a future issue to develop a real time disk scheduler in next versions. This means that real time is not currently implemented in the block layer, since it is necessary to design and implement other kind of synchronisation mechanism to access the disk (now with mutex semaphores), and a more specific study of disks characteristics as geometry, caches or recalibration functions.

File System Description

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 the next section 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 is, 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 :

- i. how data is allocated on the disk
- ii. 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 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. 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 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 technique 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. Access patterns that 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 cache.
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 implementation¹.

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 were designed to provide a fast way to recover data when a crash happens, which is a drawback with ext2 Linux file system, 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).

PORTABILITY

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 can 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) do not handle DMA address ranges properly², 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 continuously 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 bandwidth required.

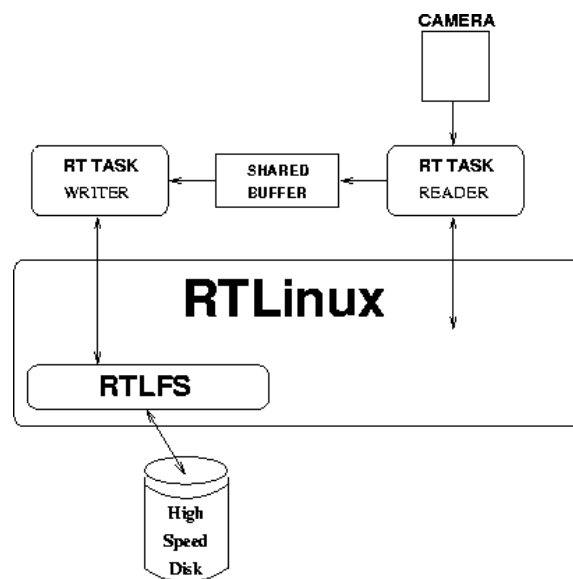


Figure 1-1. Reader and Writer division approach

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 attributes 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 environments is assumed the checking can be more relaxed, mostly related to catch programming bugs.

Available open source realtime filesystems

As a previous step we have studied others file systems looking for how match with the characteristics defined in the previous section, and to evaluate the cost of migration of that file system 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. FAT is an example of these file systems and some of the characteristics are:

- Sectors are grouped in clusters, which can be as long as disk capacity³.
- 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.

However, FAT file system are not very efficient when 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 system⁴ have a more complex block allocation structures. A linked list for space management using this functionality would achieve the requirements.

The FAT file system implementation of Linux is tightly dependant on the internal Linux structures as buffer cache. This along with other complexities usual in file systems designed for general purpose operating systems as file access attributes or file access times⁵, as well as legal issues (Microsoft™ is planing to request patent royalties) influenced in the decision of the implementation of a file system from scratch.

Real Time File System (RTFS) Specification

Once analysed the characteristics that the file system should provide, this section presents the proposed design and the implementation that meets these requirements.

A full definition of structures appears in APPENDIX A. The next figure shows the global structure of the file system:

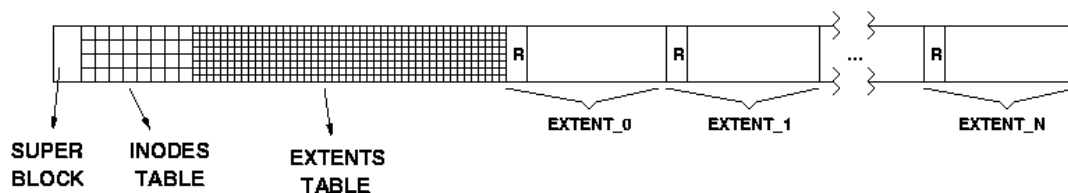


Figure 1-2. RTLFS structure

We present the file system components with the main characteristics:

- *SUPER BLOCK (SB)*

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 has 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, permissions, 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 per extent 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 a 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 an 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 directory 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:

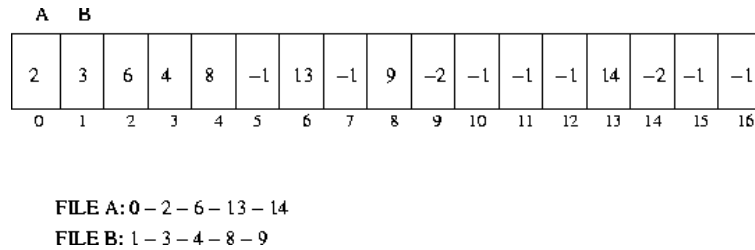


Figure 1-3. Extents table example

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, although other algorithm 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 asynchronously 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.

We solve this using the first sector of the extents for reliability. The information written in the first sector is a *rtl_reliable_extent* structure (see APENDIX A):

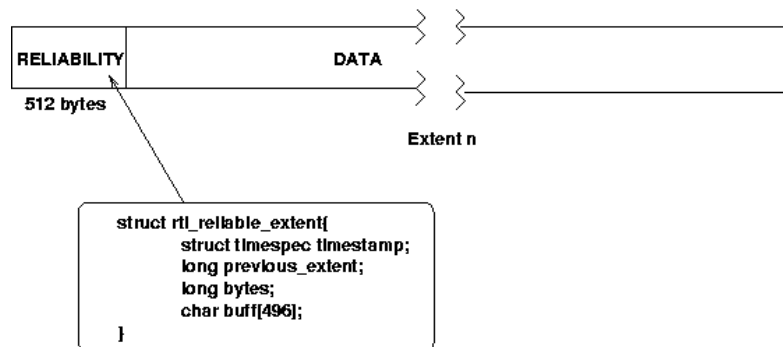


Figure 1-4. Extent header

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. 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.

Features

- *PORTABILITY*: 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 the file system when is not mounted by RTLinux, but can do it in READ mode only.
- *USABILITY*: Some aspects of the file system can be managed from Linux:
 - mkfs.rtlfs works as other mkfs tools, using a parameter given to format a partition.
 - rtlfs.chk searches the reliability data of a bad-unmounted file system building a consistent state. The file system will can not be used if it was not umounted (for example when a crash).
- *FILE SYSTEMS LIMITS*
 - The maximum size of a file is 2^{64} bytes.
 - The maximum number of extents per file is not limited.
 - Maximum number of files: 1638.
 - Maximum number of extents: 16384

Validation Criteria

Since this component has not utility without IDE device driver component, validation criteria will concern to both components, and it will be focused on the file system. The goal of the RTLinux block layer is to store media streams efficiently, then a criteria will be how many concurrent video streams can be written at the same time, and how it affects reading a video stream stored previously at the same time that

other video streams are written. As one of the aims was to share the same IDE device between Linux and RTLinux, another validation criteria would be what is the performance of Linux when some RT-tasks are working with the IDE device.

Tests

We will test what is the behaviour of the system when one or several video streams are being written to disk. For this, we will make simulations of the different MPEG modes, and this will help us to limit the system possibilities. It will also be tested which is the system behaviour when a task reads a video stream stored when, at the same time, other tasks are writing data. Finally, to see how Linux is affected with this implementation, tests will be done with Linux doing several input/output oriented tasks at several levels (desktop user, server machine).

APENDIX A

```
struct rtl_disk_super_block{
    unsigned char id;
    unsigned long s_magic;
    unsigned long start_sector;
    unsigned long total_sectors;
    unsigned long inodes_sectors;
    unsigned long extents_sectors;
    unsigned long extent_size;
    long total_inodes;
    long total_extents;
    long current_files;
    long current_extents;
    long last_inode;
    long last_extent;
    loff_t sequence;
    unsigned short mount_state; /* 1 if system crashes */
}
```

Figure 1-5. Super block

```
struct rtl_inode{
    char name[16];
    kdev_t dev;
    int pos_in_table;
    umode_t i_mode;
    unsigned char free;
    long extent_id;
    loff_t size;
}
```

Figure 1-6. I-node structure

Notes

1. However, as we will see in our implementation we need a 512 bytes cache per file for performance
2. 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. Usual FAT implementations have a upper limit of 128 sectors = 64Kbytes
4. In the UNIX first version
5. If simplicity is a goal in the design we must avoid a lot of non critical features

