Raiding the Noosphere:  
the open development of networked RAID  
support for the Linux kernel

Peter T. Breuer and Marisol García Valls

Universidad Carlos III de Madrid

SUMMARY

The Noosphere is a term borrowed by open-source advocate Eric Raymond to denote the virtual  
world of the Internet. Fitting a new driver into the Linux kernel requires a “noospheric” strategy as  
well as an engineering strategy, because the code is part of the open-source development process,  
not its end. This article recounts the technology and the development process followed for a “fast  
and intelligent” driver extension to the existing Linux software RAID subsystem. The development  
adapts the kernel RAID subsystem for use in the context of network-attached storage.

KEY WORDS:  Operating Systems, Open source, Linux, Storage, Networking, Software Engineering

Introduction

How does one write a device driver for the Linux kernel? It is not a simple process with ready-  
to-be-followed instructions. The ingredients are to be found in the published kernel source code,  
but “what and where” in those three and a half million lines of code are not precisely knowable  
before beginning a project, nor is the “how” for putting them together. Thus writing a piece of  
kernel code is in many ways a voyage of discovery into a strange new world – it will inevitably be  
accompanied by explorations aimed at developing, confirming or refuting theories about the new  
environment and it is difficult to chart a course in advance. But, putting aside for the moment  
the larger question of precisely how, approximately how long or how much effort should it take  
to write some Linux kernel code? That is a question that may be attacked via experience. Over  
December 2002, one of the present authors wrote an open-source driver targeted at the Linux  
operating system kernel version 2.4, so the first and most approximate answer to that question

*Correspondence to:  Peter T. Breuer, Universidad Carlos III de Madrid, Avda. de la Universidad, 30, Leganés,  
Madrid 28911. Spain. (ptb@it.uc3m.es)
is “approximately one man-month”, which is the standard atomic unit of costing in a software project. Despite the neatness of that statistic, however, the production of a working code is the first step in an open-source project, not the final step. The open-source development cycle is one of continuous review and improvement, and working code is only the price of entry, more like the deposit than the final payment. “RAID” stands for Redundant Array of Inexpensive Disks and refers to a class of disk storage technologies that provide redundancy and fail-over through the use of multiple physical disks. The intent of this article is to (1) document for the first time some of the code involved in the RAID subsystem of the Linux kernel, and describe changes made to it, and (2) describe the open-source development process experience – as well as propose a methodology for open-source development as it specifically relates to a Linux kernel project – gained in the fast RAID ("FR") project for the Linux kernel. The FR drivers are an effort to add intelligence and adaptability to the Linux kernel software RAID subsystem with the aim of supporting RAID over networked devices such as ENBD [1, 10, 2]. They are products aimed at the Noosphere.

“The Noosphere” is a term borrowed by open-source author Eric Raymond [8] to denote the virtual universe based on and around the Internet. Like any universe peopled by humans, it has its own economics and cultures, and developing software in the context of the Noosphere is a question of harnessing and synergising with its customs and forces. The title of this article ("Raiding the Noosphere") therefore refers not only to the kernel RAID subsystem targeted by the driver, but also to the getting and making use of the resources available in the Noosphere. Those are principally intellectual resources. The question of how one writes a driver for the Linux kernel has to be answered taking into account the reality that the driver project exists within the Noosphere. If that is disregarded, then the project will likely die for lack of market or lack of participation, lack of continued maintenance and consequent obsolescence.

This article will show that the life-cycle of an open-source Linux kernel software project does not obviously follow the classical forms of waterfall, spiral, V-cycle, rapid prototyping, and so on. That may be because there are no absolute requirements on it other than that it come to exist, survive and prosper, and it can certainly redefine itself and its goals as it goes along.

For example, the FR project’s initial coding effort deliberately served only to establish and facilitate contacts in the wide world of the Internet, and define an identity for itself there. Neither the man-month nor any internal measure of effort is meaningful in that context, because the result of a month’s work is not a deliverable for a fixed requirement, but rather serves as a beacon to attract intellectual capital and other assets together into an organisational entity that may then go on to develop in different directions. Although doubtless heavily touched by the founding aims, the project redefines itself as it goes in terms of the goals of the contributors it attracts. This is the “open-source” development model in action: a successful project generates effort not only internally, but from the bountiful resources contributed by others on the Internet, “passing through” on their lifetime journey in the Noosphere.

The layout of this article is as follows: first RAID will be introduced and the Linux kernel software RAID drivers sketched. Problems with RAID in the modern network setting will be discussed briefly and then the FR design objectives and a little of its technology will be introduced, before moving on to the section discussing high-level software engineering aspects of source code production in an open-source environment. Measures of project activity will be given and details of the life cycle experience in the Noosphere will be set out, together with a proposed methodology for project development in that context.
RAID

The acronym “RAID” stands for “Redundant Array of Inexpensive Disks” and has classically referred to a range of hardware devices which contain several individual physical hard disks but present them as a single large disk to the computer. Usually the device provides redundancy, permitting, for example, one disk in the set to fail without the complete unit suffering any loss of data overall. The failed disk can be removed, and a new one added, without interrupting uptime. RAID devices are an essential part of any kind of serious data and file service.

A fundamental characteristic of RAID is background re-synchronisation. When a component is replaced in an array, the array will automatically synchronise the new component with the existing ones, copying from them in the background and updating with new information at the same time. Throughout the re-synchronisation the array as a whole remains operational, and there is no perceptible impact on the user.

Disks have grown so large and networks have grown so pervasive that the standard RAID model, based on relatively small disks compacted physically close together, no longer has the general validity it once had. Deficiencies have shown up in the ever larger and more dispersed networked RAID configurations that are becoming common. Here is one specific problem that the fast RAID drivers aimed to solve: full re-synchronisation is intolerably slow for medium and large configurations in a network-attached storage setting. Here is another: keeping dozens or hundreds of remote disks connected properly over a network in a RAID configuration is like herding cats – difficult to start going and difficult to keep together thereafter.

In this section, software RAID – as seen within the context of the Linux kernel – will first be described, and then the problems with the current model in the networked setting will be examined in more detail.

Soft RAID in Linux

The “soft RAID” drivers within the Linux kernel perform the same function as physical hardware RAID controllers, but in software. They allow several different devices (which may be physical disks or other block devices or other RAID arrays) to be aggregated together and addressed as a single unit. The type of RAID used determines the kind and degree of redundancy offered.
Table I. The three principal operations on a running RAID subsystem.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>set faulty</td>
<td>This operation marks a disk as bad. It simulates the signal from a red failure light on a disk housing within a physical RAID array. Thereafter the RAID driver will not attempt to read from or write to the disk. A disk will automatically be marked faulty by the driver if an I/O request to the disk fails.</td>
</tr>
<tr>
<td>hot remove</td>
<td>This operation on a disk already marked faulty makes a space in the array, allowing a subsequent “hot add”. Hot remove is not allowed on a disk that has not been marked faulty.</td>
</tr>
<tr>
<td>hot add</td>
<td>This operation replaces a disk that has previously been removed. It triggers a rebuilding of the (redundant) data that should be on that disk from the data still present on the other disks in the array. The re-synchronisation process does not affect the accessibility or operation of the array.</td>
</tr>
</tbody>
</table>

The types of soft RAID standardly available in the kernel are linear, striped (type 0), mirrored (type 1), type 4 and type 5. These different types are illustrated in Figure 1. Linear and striped raid offer no redundancy, merely serving to aggregate several devices into a much larger device. Linear RAID aggregates the devices one after the other in linear order for block access, while striped RAID aggregates them in interleaved order, so that block 1 of the first disk is followed by block 1 of the second disk which is followed by block 2 of the first disk, and so on. Type 1 RAID is mirroring. That is, reads come from one of the component devices and writes go to all component devices. A mirror can survive without harm the loss of all but one of its components.

RAID types 4 and 5 are similar. In RAID type 4 a designated single device in the set contains the parity sums, bit by bit and block by block, for all the other devices. The other devices are usually aggregated in a striped mode, but details may vary. The parity device can be lost without any loss of information because the parity bits can be reconstructed as soon as the device is replaced. A data device can also be lost harmlessly, since its data can be reconstructed by summing the remaining devices, bit by bit and block by block, and comparing with the result recorded on the parity device. The difference is the data on the lost device. When the device is replaced, its data can be reconstructed by means of the above calculation, and in the meantime, the data can be reconstructed “on the fly” as it is needed. RAID type 5 is like RAID type 4, but the parity data is spread in a stripe that passes over all the disks in the array. There are several performance advantages under ordinary circumstances. But when one disk is lost, then it requires $n$ reads (one from each of the remaining $n$ disks) to recover each block of missing data. RAID type 4 and 5 can survive the loss of one of their $n + 1$ active components. Extra “hot spare” disks in the device, normally inactive, can further increase the degree of protection available via redundancy.

Standard (redundant) RAID devices permit three operations on each and any disk in a running array, without interruption of normal service: set-faulty, hot-remove, and hot-add. These are detailed in Table I. This set of operations is common across both classical RAID hardware devices and the Linux kernel’s software RAID subsystem.
Apart from those operations, an array can be *started* and *stopped*. It also has, at one point in its life, to be *initialised*. This creates on-disk records at the end of each of the array components. These 64KB *super-block* copies identify the array during subsequent starts and restarts.

**Problems with the RAID operating model**

The standard RAID operating model nowadays tends to suffer from both

- the size of the component disks and as a consequence, the arrays that they make up, and
- the speed of the connection to the disks.

In the Linux kernel, arrays of several terabytes in size are possible (the author has seen one of 1.7TB in real use) but 2TB is the current\(^\dagger\) limit on Intel 32 bit architectures. Communication speeds vary from 8MB/s over TCP on dedicated duplex 100BT Ethernet, to 60MB/s over a local bus.

In the standard model, it is expected that:

- a disk failure will be permanent, and
- repaired by replacing the disk with another.

The standard RAID action after hot-adding a replacement disk is a re-synchronisation of the whole disk. In a type 5 RAID array (the most common) this means reading data from each of the other disks, calculating, and writing to the target disk. The re-synchronisation will be throttled automatically by driver or hardware device so as not to impact adversely on I/O bandwidth, but even at top speed on a local bus (say 60MB/s - an impressive figure rarely achievable in practice) a 1TB array will take 16000s (4.5h) to reconstruct. During this time the array is without the redundancy that it normally has and another failure will be catastrophic.

\(^\dagger\)In the 2.4 kernel, block device sizes are counted in 512B sectors, using an unsigned 32 bit integer. In the 2.6 kernel these limits are being removed.
Table II. Goals of the FR1 driver.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Full re-syncs of newly replaced components in the array should be avoided in favour of intelligently directed partial re-syncs.</td>
</tr>
<tr>
<td>2.</td>
<td>Old components should be recognised automatically when replaced, and the appropriate action taken.</td>
</tr>
<tr>
<td>3.</td>
<td>In ordinary use, asynchronous modes should be available for writes.</td>
</tr>
<tr>
<td>4.</td>
<td>More failure modes should be dealt with automatically, particularly that where the node holding the array goes down (on reboot it is not always clear if there are minor imperfections that have gone undetected).</td>
</tr>
<tr>
<td>5.</td>
<td>When a component device is aggregated into an array, it should be informed of which array, and then if the device is errored off-line, it should call back when it becomes available again and the array should react appropriately.</td>
</tr>
</tbody>
</table>

Complete re-synchronisation is inevitable when the disk really is a new, replacement disk, but experience teaches that is not the most common failure mode. Temporary failures are common (bad cables, power supply problems, etc.), and whole-array failures are also common. Indeed, the latter likelihood is one reason why it is becoming ever more common to see RAID over disks that are located on remote nodes in a local or wide area network, connected by Ethernet.

The advantage to that configuration is

- protection against local disasters that would take out the whole array at once, and
- cheapness – Ethernet, TCP, etc. are cheap off-the-shelf technologies.

When the components of the array are distributed around the network, not only will the data survive, but it may well be possible to “fail over” the array to another configuration in which another network node supplants the failed array node and uses the same remote disks, with no appreciable down-time at all. The disadvantage in networked operation is increased latency in synchronous operations (such as reading) and increased susceptibility to new fault modes, such as network brown-outs. But specific technical solutions to these problems are available and will be discussed in this article.

In such networked configurations, it is very likely that:

- a disk failure will be temporary, and
- the failure will be repaired by restoring the same disk – a *hot repair*.

As a consequence, the remote disk data will be largely intact, but not quite up to date. In that situation, it is not necessary to synchronise the whole remote disk, but merely synchronise the blocks that have changed since it left the array. This hot repair procedure is illustrated in Figure 2.

Moreover, the costs of a complete synchronisation over a LAN or WAN are greater in terms of time, because transport velocities are often an order of magnitude slower than for local disks.
1MB/s is the maximum available over a full duplex 10BT Ethernet connection, and 10MB/s the absolute maximum over a full duplex 100BT connection, even in the absence of other traffic. Going any faster requires technologies such as Gigabit Ethernet, or fibre, with much greater costs for the physical infrastructure support, and/or reduced distances. Syncing a 1TB array over 100BT Ethernet may require three reads for every write in a three disk array, meaning that the maximum velocity is only 2.5MB/s, which means that it will take 40000s, or some 12h to re-sync the array. There are therefore many incentives working in favour of reducing the amount of re-synchronisation that needs to be done.

When the components of an array are networked disks instead of local disks, then failures from brown-outs will be more frequent. The administrator cannot be expected to deal with the resulting frequent calls and it is essential that the components of the array advise the array by themselves when they have recovered and how they should be brought up to date. Spontaneous recovery is not something that classically happens in physical RAID arrays.

These considerations drove the development of the fast RAID drivers for the Linux kernel.

The FR1 driver

The first fast RAID driver prototyped was for RAID type 1 (i.e., mirroring) and it became known as the “FR1” (pronounced “ferrari”) driver.

A list of its design objectives is given in Table II. The driver aims to achieve faster and more intelligent software RAID, sensitive to the requirements of network (LAN/WAN) attached storage.

The primary aim is to avoid total synchronisations of the array whenever possible, as well as to intelligently automate the administration of the array, in that it should initiate hot-add and hot-remove sequences when appropriate.

The key to the technology is the introduction of one bitmap for each RAID array. A bitmap is a linear array of 1 bit elements. The bitmap is marked (“dirtied”) at a position \( n \) in order to indicate that there is some array component with an imperfection in block \( n \). It is unmarked (“cleaned”) to indicate that all the array components are in sync at that position. See Box 1 for a summary of the way the array bitmap is used to record out-of-date blocks on a temporarily unavailable mirror component and subsequently

---

**Box 1. Bitmap use in FR1**

1. When a RAID disk is set faulty, the bitmap is started. A kernel event counter is stamped on the bitmap meta-data;
2. on subsequent writes to the RAID array (now without the faulted disk), the bitmap is marked dirty for the block being written;
3. when a disk is hot-added, persistent meta-data on the disk is compared. If
   a. it matches the array meta-data, and
   b. the disk major/minor numbers match what was there before, and
   c. the event count on the disk matches or exceeds the count on the bitmap,
   then the bitmap was started before the disk left the array and records its missed writes, so a partial re-sync of the disk is initiated;
4. during the re-sync, only those blocks marked dirty in the bitmap are written.
   The bitmap is cleared as each block is written. When the re-sync finishes, then the bitmap is stopped.
rectify those and only those during array resynchronisation. The mechanism also ensures that an old replacement disk is recognised in order that an intelligent re-sync may ensue.

The bitmap is also used to support an asynchronous write mode. In this (optional) mode a write to the mirror is acknowledged after one but before all mirror components have been updated. See Box 2 for an explanation of how it works.

This strategy lowers latency and is effective when a networked device is a component of the mirror – as it will be slower than the local components – but it increases the probability of data loss: if the local node crashes heavily after reacting to a successful mirror write but the local array component is unavailable at reboot, then the remote data will be used even though it may not be up to date. For that reason asynchronous write mode should be used with caution.

The array bitmap also contains features that enable it to work under low-memory conditions, that is, when calls to the kernel for more (temporary) memory via `kmalloc` fail, returning the null pointer instead of a pointer to an available block of memory.

It is to be expected in the case of an I/O system like RAID that it will experience memory pressure as data is streamed to the device. The kernel’s virtual memory system may fill with dirty buffers before data is sent from virtual memory down to the RAID device. The kernel is essentially out of memory at this point. If the driver asks for more memory, it may deadlock or livelock. This cannot be allowed to happen, so the driver code must run in a degraded mode when memory is short, running without requesting more memory.

To continue working, the bitmap drops in accuracy when memory is scarce, changing from a precision of 1KB to 2MB. In degraded mode, it marks a 2MB block as dirty if any 1KB block in that zone is dirty, and counts (balanced) mark and clear calls to do so. See Box 3 for how it works. The result is at worst extra catch-up work due to the imprecision in the event of a failure and resynchronisation, but no block that should be copied will not be copied.

What happens during a write storm? A networked component of the array is slower than the local components so dirty buffers waiting to be written to the network device can pile up in the virtual memory system, starving the kernel as a whole of memory, even to the point of choking.

---

**Box 2. Bitmap in asynchronous writes**

1. The bitmap is marked before every write;
2. the bitmap is cleared only if the write was completely successful – that is, if the write went to all the array components to which it should have gone;
3. If a write is errored, then the disk is faulted, the bitmap is marked active and it begins to account for all missed writes.

**Box 3. Bitmap memory for 1TB disk**

1. The bitmap gets new pages on demand and releases them when they are clean. A page is 4KB, representing 32K on-disk 1KB blocks. 32K pages map 1TB on disk.
   
   Total: \( \leq 128 \text{MB} \) (dynamic).
2. To notice when a bitmap page is clean, the bitmap has a 16 bit (2B) counter for each.
   
   Total: 64KB (preallocated).
3. When new pages are unavailable, the bitmap counts only the imbalance between the number of mark and clear attempts, using one 16 bit counter per 1/16 page.
   
   Total: 1M (preallocated).
4. The bitmap maintains a reserve of 8 bitmap pages in readiness.
   
   Total: \( \leq 32 \text{KB} \) (dynamic).
off TCP networking for lack of stack buffers. If the net is slow for whatever reason, however, the network device will time out requests. At the first error the RAID array will fault the network disk offline. Thereafter writes to the faulted component will not be attempted by the FR1 driver, and the array bitmap will be marked for the absent disk. At some point the kernel will come out from under memory pressure and contact over the net will be reestablished. The networked disk will tell the RAID system it is online again, and the FR1 driver will reinsert it in the array and begin (background) re-synchronisation of the missed writes in order to bring it up to date.

**The FR5 driver**

The changes made in the Linux kernel RAID subsystem in order to produce FR1 are listed in Appendix II. These changes are somewhat generic in nature and have been applied also to the production of a “Fast RAID 5” driver, or “FR5”. The appendix to this article categorises many of the patch components.

More details of the Linux RAID subsystem, enough at least for the FR1 (and hence FR5) patch to be described, are given in Appendix I.

---

**Software Engineering: Open-source Resource Estimation**

How much real effort goes into a Linux kernel project, and how much does one man month represent in such a context? The total effort was in this case about one hundred lines of C code per day for one month, in sum, counting both additions and changes in the daily total, according to the evidence from the version control files maintained from day to day.

That figure should be compared with that from the open-source GNOME project, where a comprehensive study [4, 5, 6] reports about one hundred and forty-six lines of C code a day per programmer which in turn is more than the numbers reported in Brooks’ famous essays [3] on the mythical man-month. The GNOME project seems to develop faster than the Linux kernel, in general, but it is a collection of relatively discrete libraries and applications that permits more independent development of its several parts.

The FR1 author wrote at maximum capacity, so that amount of one hundred lines per day for a month probably represents one or at most two “function points” per day, in the nomenclature of software metrics – the author finds it not psychologically sustainable to plan, attack and solve more than one problem in a work session, and there are at most two sessions available per day. Therefore a working driver probably constitutes between twenty-five and forty different problems to solve, i.e. that number of function points. Yet, in the experience of the author, a standard kernel

---

[The GNOME study quoted used exactly the same metric as applied here – i.e. total lines of C code added or changed. Linux kernel code follows an exactly defined “Kernighan & Ritchie” style with indentation of 8 characters per block and lines not exceeding 80 characters. While the GNOME project follows a slightly different C coding style to that, it is not very different. Each programmer contributed on average 21000 LOC to the project, plus 15000 LOC changed, for 36000 total. The average time contributed was 246 days, from first code submission to last. Therefore the average output was 146 LOC per day, modulo imprecisions from taking the ratio of averages.]
Table III. Components of the FR1 kernel driver, with the number of functions (not function points).

<table>
<thead>
<tr>
<th>component</th>
<th># functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(FR1)</td>
</tr>
<tr>
<td>initialisation function</td>
<td>1</td>
</tr>
<tr>
<td>cleanup function</td>
<td>1</td>
</tr>
<tr>
<td>ioctl interpreter</td>
<td>3–6</td>
</tr>
<tr>
<td>device open function</td>
<td>0</td>
</tr>
<tr>
<td>device close function</td>
<td>0</td>
</tr>
<tr>
<td>other block, char or file ops</td>
<td>0</td>
</tr>
<tr>
<td>internal kernel interfaces</td>
<td>4</td>
</tr>
<tr>
<td>auxiliary support</td>
<td>6</td>
</tr>
<tr>
<td>auxiliary interfaces</td>
<td>3</td>
</tr>
</tbody>
</table>

driver normally has the interfaces shown at right in Table III (giving number of functions in each interface), and the FR1 driver had, counting in the actual code, the number of interfaces shown at left in the table. The table shows significantly fewer functions (18-21) in the FR1 driver (14-34 is typical in kernel drivers) than the 25-40 function points estimated earlier in this paragraph.

Why are there fewer functions than function points? It is because existing code had to be altered, and each alteration is itself a function point from the point of view of the coder. Supporting that conclusion, the final kernel patch contains 42 separate 6-12 line “hunks” (groups of nearby lines changed in existing files) plus two new files, which is more in line with the estimate of 25 to 40 function points. Existing functions were not rewritten wholesale. Thus, as a first conclusion:

Effort depends on the number of individual system modifications required.

Effort does not depend on the total number of functions (or function points) in the code being transformed.

This observation sheds light also to the natural tendency of an author to rewrite existing code rather than build on it. It may be easier for an author to rewrite because that requires fewer changes, and the effort (in terms of function points) is proportional to the number of changes.

Other lessons garnered from the development of FR1 with respect to the evaluation of effort in the general context of kernel driver code can be summarised as follows:

- **study**: existing kernel code has to be studied, possibly 40% of development time may be devoted to that end;
- **testing**: testing of kernel code has to be continuous, going hand in hand with development, so effort has to be split 50/50 between both;
- **writing**: writing kernel code is hard: between 2× and 4× as hard as application code;
- **limits**: psychological exhaustion has to be taken into account! It limits the number of reboots and tests that can be done in a day even if the total effort involved appears small.
Testing must occur in parallel to the writing of new code in order to avoid postmortem debugging, which is both difficult and costly. Each change needs to be tested as it is made to avoid the build-up of uncorrected errors that would otherwise tend naturally to cause catastrophic kernel crashes at run-time. It is difficult to get information back while a kernel is crashing, and wearing to repeat the experiment.

As regards the effort involved in testing kernel code, in comparison to that involved in testing applications code:

- testing of kernel code is much harder than testing of application codes because a failed test is a final test for the system, and extra time is required for reboot and re-setup;
- designing tests for kernel development is much harder than for application codes, because very small scale incremental tests are required to go with each (correspondingly small) change;
- debugging is much harder because of the relative inaccessibility of the kernel, and it may require binary code search – such a search can take days and is very debilitating to the tester.

In summary, if only 60% of the total time is available for non-study activities (40% being devoted to code study), and half of that is devoted to critical testing activity that must proceed in parallel, then only 30% of the total time is available for developmental coding. So computations of effort based on time per function point need to estimate the overall time needed by multiplying by three.

If in addition development is two to four times as difficult as for application code (as the following paragraphs will suggest), and testing and debugging is similarly harder, then compensation and/or incentives need to be correspondingly higher too. Note that the increased and more frequent testing requirement brings into play psychological limits, raising stress levels.

An open-source project also has an increased communications requirement. There is not only a core team, but potentially also the whole of the Internet that has to be kept informed, for reasons discussed in the next Section. If the project is not to produce code that will be lost again as time passes, considerable effort has to be made to persuade others to adopt parts of the code, and incorporate hooks to it in their designs. This introduces a marketing component into the development methodology, in sharp contrast to other domains where the project has at least sold itself to its own authors by the time development begins. Thus:

Dissemination takes place with and should be considered part of development effort.

Certainly existing kernel code has to be assimilated and reverse engineered, compressing downwards the time available for coding. This factor is positive overall, however, because having the working example in the kernel to hand, even though it is not didactic in nature, enables full understanding to lag production of code without prejudicing initial milestones. Code quality will improve later as more understanding develops. Yet having to study the kernel code does impose an entry barrier on a potential author (the excellent book by Rubini [9] is a prerequisite first stop for kernel developers).

Interestingly, there is also an imperative towards communication on potential contributors to the code in an open-source project. When a new contributor chooses to contribute, he or she must first convince the project to adopt the offered code patches or other contributions. The established members of the project generally do not have to prove the adequacy or relevance of their work, but the newcomer does. A new arrival will have to explain and justify their proposal in detail in
order to establish their competence. This is in principle a high barrier to acceptance, because the newcomer does not know the existing code-base as well as the incumbents do. A newcomer should not be shy in asking for appropriate information, nor in providing it, as both demonstrate their competence.

Now, how hard is writing kernel code? From a personal perspective, writing kernel code certainly feels ten times as hard as writing application code, although statistics from projects by the same authors in which both kernel and user-space code is present suggest that kernel code is objectively only between twice (going by percentage rate of growth of the source base per author) and four times as hard to write (going by overall frequency of modifications made per author). It may be that the real extra difficulty is partially compensated by extra effort, which closes the gap as measured but increases the subjective perception of difficulty.

Or it may be that the effect stems from the awareness that the penalty for a mistake in kernel code is much higher than in user-space, which elicits a worry and nervousness that is ultimately exhausting. Whether the anxiety derives from the fear of egg on the face in public, or the very private fear of becoming snagged in accumulated undebugged errors, it is a pressure. Experience has taught the present authors that the extraordinary pain of kernel debugging mandates a line-by-line attention to perfection and testing during development which makes possible public criticism into a secondary consideration at that point, although of course its public nature sets the standard for the completed work.

Project Engineering and the Noosphere

In this part of the article, the methodology used to further develop an open source project will be set out. The methodology takes a project from the point at which a prototype working code is made available (“point zero”) until the “expansion point,” where it begins to grow under its own impetus. Figure 3 summarises the methodology.

It can be seen from the side-by-side comparison in Table IV that the methodology generally follows the downward sweep of the classical software “V-cycle” (see for example [7]). The titles of the subsections in the following text that deal with each of these topics appear at right in the table.

However, the open source development cycle contrasts with the V-cycle in that it is (1) inherently cyclic and continuous, and (2) includes a particular marketing component that is lacking in the V-cycle. The difference is visible firstly in the lack of a counterpart to the publicity generating phase of the open source development cycle, and secondly in the open source software cycle’s attention to communication at every phase.

The ENBD project contains 1 kernel C code file of about 6.5KLOC, and 53 user-space files of 18.5KLOC total. Over 600 days of recorded CVS updates, on average 8 lines were added and 14 lines changed every day in the kernel code, with changes logged every 4 days on average. Over the same time, the user-space code averaged 15 lines added and 31 lines changed every day, but on average each file had changes logged only every 42 days, although some file in the set was changed on average every 0.85 days. So the user-space code grows twice as rapidly and has changes logged more than four times as often as kernel code.
Write working prototype code.

Announce on public sites, make available via FTP, HTTP, etc.

Encourage discussion, make rapid updates, solicit comments, etc.

Implement ideas and solutions to problems identified.

Manage discussions, keep contributors in touch, solicit contributions, give reminders, etc.

Analyze voiced and unvoiced reasoning, pressures, etc.

Figure 3. Steps to set up a Noosphere-based project.

<table>
<thead>
<tr>
<th>V-Cycle</th>
<th>Open Source Development</th>
<th>Section Title in Text</th>
</tr>
</thead>
<tbody>
<tr>
<td>(no equivalent)</td>
<td>Public announcements</td>
<td>Setting up stall</td>
</tr>
<tr>
<td>Requirements capture</td>
<td>Public discussions</td>
<td>Pitching a sale</td>
</tr>
<tr>
<td>Requirements analysis</td>
<td>Analyse influences</td>
<td>Feeling the pulse</td>
</tr>
<tr>
<td>Architectural design</td>
<td>Design solutions</td>
<td>Taking stock</td>
</tr>
<tr>
<td>Component design</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Component coding</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testing phases ...</td>
<td>During deployment</td>
<td>(not treated here)</td>
</tr>
</tbody>
</table>

Table IV. Relationship of the classical V-Cycle to Open Source Development Cycle.

Classically, marketing consists of a "strategy" (brainstorming, pilot launches, market study, campaign design) phase followed by a "communications" (poster design, placement of adverts, etc.) phase, and these elements are constantly reflected in the open source cycle through the attention paid to getting feedback from the market – clients and authors and others – and to being driven by that market.

Due to the immediacy of the influences that bear on open source software development, it is difficult to separate out formal stages of architecture design and component design from coding. Usually architecture is designed on the back of an envelope, or in the head of the author, and is first expressed as code, then evolves over time.
Setting up stall

Webster’s Unabridged dictionary, 1913: Stall, n. 1. A stand; a station; a fixed spot. . . . 4. A bench or table on which small articles of merchandise are exposed for sale. To make progress in the open-source arena, a source code first has to be developed to the point at which other code authors and potentially interested parties can see where it might go, and thus become interested in helping with its future development. They may add new items to the wish-list or add new directions for the future, document existing code (writing a page for the user manual, for example, or starting a frequently answered questions list) or offer the benefit of their guidance, experience and wisdom. But, inevitably, real code has to be offered before anything will be taken seriously. This is the “zero point” code. It is too easy to hype vaporware.

Once it exists, the code also has to be announced and advertised. This might be interpreted as staking a claim to some intellectual property rights or rights to ownership or leadership in an intellectual domain (what Eric Raymond has called “home-steading” in the Noosphere), but if the ego of the author permits such an interpretation, it should rather be seen as an advertising and recruiting phase.

There exist places in the Noosphere where new-project announcements can be made and discussions can take place. The following are now classical forums in which to announce an open-source project:

1. The sourceforge.net or freshmeat.net project pages on the Web.

Sourceforge and freshmeat both provide generic project support facilities and are watched by a great many people, so announcing there guarantees attention. On freshmeat, a project is announced by filling out a form. The form describes the project, who its Noosphere owner is, classifies it, and details where to download source code and where to find the project’s home page on the Web, and its change list. Sourceforge focuses more on providing storage and computing resources, but does offer web-page hosting and discussion list hosting, and various other project support facilities.

Most important, a freshmeat announcement will appear in its “headlines” for that day, and will be briefly first in the headlined list, then fall down the headline page as the days pass until it drops out of the news altogether after a week. There are about twenty to fifty announcements a day, so keeping a project in the news can be a full-time occupation. It should also be borne in mind that making announce-able changes merely for the headline can adversely affect the perceived stability and reputation of the project. But, in general, the aphorism that any news is good news holds true.

2. One should look up the Linux core authors or maintainers concerned with the area of the Linux kernel concerned (in the case of a Linux kernel project), and send them an email.

The core kernel authors may not respond immediately, probably because they are already swamped with input from other Internet discussion groups:

3. There may be some mailing lists and/or Usenet newsgroups that are already devoted to the topic. These should be searched out and a copy of the announcement sent to them also.

The main Linux kernel mailing list, to which almost all Linux kernel authors are subscribed, generates traffic of about two to three hundred mails a day. It can be preferable to follow it via Usenet gateways.
Pitching a sale

Wordnet r1.7: pitch n. . . . 3: (British) a vendor’s position (especially on the sidewalk); 4: promotion by means of an argument and demonstration [syn: sales talk, sales pitch]. The code on offer has to have some very good points in its favour before potential contributors will devote their time to making it live and grow. It does not have to be totally correct code at the time of its first presentation, but it has to be functional enough to demonstrate an interesting idea. It also should exhibit at least one character that people want and which is not currently available, though uniqueness is not a necessity. That is because open source code competes in at least two markets:

1. the market for clients of the code, and
2. the market for potential collaborators.

These are different ends of the same rod; clients are not only consumers of the code, but also testers, if only in an informal sense, and they may well eventually contribute to the project in the form of bug reports, documentation, feature requests, or even code patches. When patches are submitted, then the client has definitively collaborated in the development of the code in the project, but other forms of collaboration may also arise spontaneously from time to time in the client-project relationship. Documentation and also guidance and advice are particularly welcome.

“Point zero” for the code is the point at which it can be paraded in the open source marketplace in order to allow its potential to be evaluated by possible clients and potential collaborators, mentors or advisors. The FR1 point zero code was published on the Linux-RAID mailing list, on the freshmeat web site, and the kernel RAID authors were notified separately. A summary of features was posted on the mailing list. However, there were initially very few responses – none at all from the mailing list and only about ten from freshmeat. This may to some extent be attributed to the fact of it being a startup - on the net, where this sale was pitched, pitching a sale equates to creating a presence in the forum of choice and a startup has no prior presence, and one must be created. But there are many competing voices on freshmeat and very few readers there are oriented towards this sort of specialist kernel software; they are more oriented towards end-user application software. The Linux-RAID mailing list had the right readers but was and is hardly active as a discussion group; its readers merely monitor it – and probably many other lists at the same time.

But any advertising at all is good advertising. List “lurkers” may not take part in discussions or coding but they do communicate to other people in their local communities via word-of-mouth, and spread the word to authors or other interested parties who may not have time to devote to scanning the news media. For these people, claims of excellence or novelty may be the more important part of a project, and even a little attention to it from people who they perceive as experts may sway them to evaluate the project highly and broadcast news of it. So it is worth making the effort to announce in peer-evaluated fora like the mailing lists even if responsiveness is low.

The mailing lists in particular are fora where one may expect to find those experts that are interested in and capable of furthering the aims of a project. However, experts are not necessarily the best collaborators, since their effort might already be largely committed elsewhere. Talented coders looking for a project in which to make their name are also a potential market. But this particular project was not large enough to be able to attract their attention as a coding vehicle.
Feeling the pulse

Webster’s Revised Unabridged Dictionary (1913): To feel one’s pulse. (a) To ascertain, by the sense of feeling, the condition of the arterial pulse. (b) Hence, to sound one’s opinion; to try to discover one’s mind. “Point zero”, in which one has a working code to offer, is more nearly the beginning than the end of the open-source development cycle. What happens afterwards? In the case of this particular project, the code was thrown away! Every line was discarded, and the whole rewritten from scratch.

Why? Because it was too much to be digested by potential collaborators, although it was of interest to clients. There was no feedback on the original offering received from the kernel core authors responsible for the area. The problem was that the code replaced sections of the existing kernel functionality with its own more tightly directed implementations. It could not be integrated with existing kernel code and, as it was, would always have to stand apart on its own. So authors of existing kernel code had no incentive for an investment of time and effort in the code. Given the demands on their time and their own involvements in other projects, they could not be expected to devote time to understanding the new code.

The reason why the new code replaced the existing functionality rather than integrated with it was because the existing kernel code was very complex — overly complex for the purposes of the proposed feature additions — and undocumented. The project author saved prototyping time by reverse engineering the existing code functionality via black-box testing of the kernel, and then implemented the observed behaviour, providing a parallel set of kernel support mechanisms. Interactions with other kernel mechanisms are always very difficult to control and it is preferable from the point of view of prototyping time and code safety to develop parallel mechanisms which cannot interfere with the existing ones. This works in order to achieve a prototype, but renders the result unattractive as far as potential collaborating authors are concerned.

It is important to take note of opinion in the Noosphere and lack of comment may be as significant as excess of comment.

Taking stock

WordNet (r) 1.7: take stock, v : to look at critically or searchingly, or in minute detail. During a further month of development, the FR1 driver was re-drafted as a patch to existing kernel code. This required a better understanding of the existing kernel code to be developed, but appeals to the mailing lists and the kernel authors for specific problems initially went unanswered. It can be a problem in an open-source community when the communication channels fail to work, and it is hard in those cases to know how to address it. The original authors have by default a monopoly position with regard to their own code, in terms of the understanding they possess, and it would take a long

The existing Linux kernel code for the subsystem in question consists of one base “class” file of four thousand lines of C code defining a support subsystem, and one file defining a client driver of some two thousand lines of C, plus the associated header files. The client is one of a set of five or six extant drivers all of about the same size which build on the same support base and which provide alternative and inter-operative functionalities. The design is flexible, but very difficult to abstract from the implementation code.
time for anyone other than them to develop the same understanding, even with tutelage. That acts as a barrier when it comes to developing code that has to operate with an existing open-source base that may be complex and not well documented. The FR1 code was re-drafted in order that it may pass through the barrier.

After one month of further work, the FR1 patch consisted of some eight hundred lines of additions to the existing kernel code, and some sixteen lines of changes to it. It evidently did not replace existing kernel functionality, but rather added to it, and as such was immediately much more acceptable to existing kernel authors. When it was published on the mailing lists it went accompanied by

1. an item by item account of what each section of the patch did, and
2. a high level description of the design, abstracted from the above analysis.

The latter was what succeeded in eliciting comments from the kernel authors. With the code listed, analysed, and presented at a very high level of design, then the intellectual effort required to understand it was low enough that contributions could be made by the authors, who are people familiar with the principles of their own code, but no longer the details of it. The demands on their comprehension time made by this presentation were low, so the kernel authors immediately offered comments, opinion and advice, despite the many other calls on their time in other projects. It turned out that they had experience and appreciation of the problem attacked by the new driver code, and had their own ideas on how to tackle it.

The development code account balance at that point showed not two thousand five hundred lines of C code for one month of effort, but eight hundred lines of C code for two months expended. That is only about a dozen lines of finalised code a day on average, but evidently the statistic hides more than it reveals. Considered cumulatively, about five thousand lines of code were written in total over two months, at a rate of about eighty-five lines of code a day, on average, for a final result of only some eight hundred lines of patch. These figures match the hard figures available from the version control application that manages another kernel project by the same author, for which twenty months of development also proceeded at a rate of about eighty-five lines of code changes a day, although only about half of this is represented by kernel code.

Summary of Contributions

The FR1 technology initially presented to the Linux-RAID mailing list used one bitmap for every RAID component device. After the presentation of the code as a patch, and the abstract summary

---

A second project by the author, ENBD, has hard long-term development figures available from the source code control system used (CVS). The code-base in question has a kernel-space part and a user-space part. Its kernel code now consists of some 6500 lines of C code in a single file plus header and the user-space part consists of 18000 lines of C in 53 files. Over a 20 month (600 day) period, the source control system records that the kernel code was revised 156 times, or once approximately every 4 days, and each time an average of 123 lines were added and 96 deleted. This is a net addition of 8 lines per day and 25 changed, or 33 per day total. Over the same period the user-space code suffered 711 changes, i.e. each file was changed approximately 14 times over the period, or once every 42 days, although overall there were 1.185 changes per day, somewhere. The average delta per change was 46 lines added and 31 deleted, for 15 lines added in total and 31 changed, a total of 54.5 per day. Together the total is in the range of 87 lines changed per day.
of the code features and technology on the list, the core kernel authors, Ingo Molnar and Neil Brown, made several design suggestions, one of them being to only have one bitmap per RAID array. The suggestion greatly simplified the code and was easy to implement.

Neil Brown also outlined the recognition mechanism now used to detect the re-insertion of an old device and which is used to trigger an intelligent resync. He pointed to a kernel “event counter” as a suitable indicator of when a device had separated from the array. When the array is in good health, from time to time the event counter value is stamped into the component devices on-disk meta-data zones. When the bitmap is activated the last such value is bound to the bitmap. When a device is reinserted into the array then its event counter value shows if it separated before the current bitmap was created, or after. If afterwards, then the bitmap can be used to direct the re-synchronisation of the device.

The event counter is stamped onto the disk components only at significant points in the lifetime of a RAID array: when the array is started; when it is stopped; when a disk is faulted (obviously the disk in question is not stamped); when a disk is removed; when a disk is added.

The idea for asynchronous writes was distilled from several different offerings by individuals. Russell Coker (a GNU/Linux Debian distribution maintainer and developer) saw the freshmeat announcements and proposed a journaling mechanism, in which the bitmap would be preserved on disk and used to aid recovery after an unexpected reboot, his main concern being with arrays that had to be fully re-synced or remade simply because they were in an imperfectly known state. At the same time, Paul Clements, lurking on the Linux-RAID mailing list, wrote to say that his company had considered some of the same changes as were currently being published in the FR1 device code, but had not been able to complete them, for various reasons. His main interest was in making the RAID device asynchronous, and he offered his collaboration.

As it turned out, once the bitmap has been reduced to one per array, it can at once be used as a journaling mechanism, and that in turn allows asynchronous writes to be implemented without any particular extra code being needed. The existing code had to be altered to allow I/O completion to be signalled by whichever array component received the first I/O of a set.

All in all, about a dozen posts to the Linux-RAID mailing list giving code and detailing progress or posing questions obtained about four direct responses from the kernel code maintainers on-list, and about a dozen correspondences with individuals and the maintainers off-list. Some of those individuals have involved themselves with writing code for the project. Their mails discussing their coding problems have been broadcast to the mailing-lists and other forums, and in turn have attracted more detailed advice from the core kernel authors.

Neil Brown in particular has pointed to defects under memory pressure in the detail of the schemes initially offered for asynchronous writes. His initial skepticism about the survivability of the technology under memory pressure prompted further explanations of why it was safe – performance had always been designed to degrade rather than fail. But Neil’s worries triggered checks of the code and also performance optimisations. Neil proposed, for example, a counter for pending writes per device “zone” when out of memory, so that when the counter dropped to zero again the memory allocation could be retried in safety. This required more careful balancing of the notifications for pending and completed writes. And Neil also proposed a small memory cache in the driver itself to make management faster and more proof against system shortages. These mechanisms have been implemented. Neil also requested that the degradation in device granularity from 1KB to 2MB zones that currently occurs under extreme memory pressure be reduced to be from 1KB to 256KB...
instead. This will be implemented in the future, as will requests to be able to vary the zoning from the 1KB-sized default that is used under normal circumstances.

Future Developments

Since this article was first written, Paul Clements, the Linux network block device driver maintainer, has produced a raid1 driver for Linux kernel 2.6 that is based on the FR1 driver, but which includes an on-disk bitmap, instead of an in-memory bitmap. Neil Brown, one of the kernel RAID maintainers, as of kernel 2.6.11 has begun to push the features of these two drivers into the kernel source. That is, notably:

- The bitmap has been made persistent, as a file on-disk on the node holding the array.

The feature has been labelled “intent logging”, but that properly refers only to the alteration of the make_request code in the raid1 driver to queue several write requests at a time before submitting them to the array. The bitmap is first marked dirty for the queued write requests, the dirty bitmap pages are flushed to disk, and the writes are then executed. As the writes are fully completed, the bitmap is unmarked, but changed bitmap pages are not necessarily flushed to disk – that happens opportunistically, so the bitmap state on disk is generally pessimistic.

The persistence of the bitmap allows for a mirror node to be rebooted and to pick up where it left off, without resyncing the whole array.

It also became apparent that the default Linux software RAID behaviour of expelling a disk from the array on a read error is excessively easy to trigger nowadays, both because disks do develop read defects (which cannot be filled in and corrected by the disks themselves on the fly, unlike write defects, which cause the disk to swap in a reserve sector for the faulty sector), and because the network itself is relatively more unreliable than cable. This occasioned a further “robust read” patch for FR1 which serves to:

- Retry failed reads on other mirror components but never expel the faulty disk.

Also:

- the technology and the kernel mechanisms now in place have been extended to cover the RAID type 5 driver (in the “FR5” driver), and should eventually be extended to cover other forms of software RAID too.
- A mechanism like the FR1 I-am-well/un-well notification mechanism needs to be made generic in the Linux kernel.

Whether the kernel’s existing mechanisms for notification of insertion and removal of removable media can serve here needs to be investigated.

The FR1 driver has been ported to Linux kernel 2.6, and patches derived from Paul Clements’ development of it are now being incorporated into the post 2.6.10 Linux kernel software RAID code base by Neil Brown. However, a generic bitmap architecture in the md driver that would serve all forms of software RAID has not yet been attempted by anyone.
Software


Conclusion

This article has described the Linux kernel RAID subsystem and the alterations to it made by the FR1 driver, which aims at adding intelligence to the subsystem in order that it may operate satisfactorily in the context of RAID over network attached storage.

The open development methodology used by the project has been described. It is the case that extra effort is required to make incremental changes to a system developed by others, as required by the open source development paradigm, rather than developing a system from scratch, and extra effort has to be apportioned to the publication and dissemination of intermediate results and news in order to coordinate efforts and integrate with other projects.

This article also may be considered part of that effort.

Acknowledgements

We are grateful to our colleagues Luis Sánchez, Andrés Marín, and Celeste Campo Vazquez for their helpful criticisms of the drafts of this article, to several unnameable Usenet-izens who kindly proof-read their way through early revisions, and to the anonymous referees who provided many deep and insightful comments.

REFERENCES

Figure I1. For writes, (left) the raid1 driver make_request method stores the incoming buffer head (bh) meta-data in a master raid1 buffer head (rh1_bh) and generates mirror I/O buffer heads (mbh) directed at the mirror components. During raid1 resync (right), the sync_request method generates read requests on one component, and passes the buffer to a write request on the other component.

APPENDIX I. The Linux Software RAID Subsystem

The following sections describe the Linux kernel RAID architecture in sufficient technical detail that the work in the project described in more global terms in the earlier text can be understood appropriately. It will also be useful to code authors aiming for a grasp of that architecture.

Support for the different kinds of software RAID in the Linux kernel is provided by the “multi disk” ("md") driver. This driver manages RAID arrays at high level. It implements the administrative aspects of the three major operations on RAID arrays set out originally in Table I, that is: set faulty; hot remove; hot add. But the semantics of many operations are delegated to specialised personality drivers which handle different types of RAID. The architecture of this area of the kernel is depicted in Figure I2.

A device may be included in several different RAID arrays as a result of stacking one RAID device on top of another, but it appears only once in the linked list of all device objects in the md driver. Pointers back and forth maintain the relationships between the device objects and the arrays which contain them. The md driver first verifies the validity of and then hands off operations on these arrays to the RAID personality driver for the array. These operations are usually derived from explicit set-faulty, hot-add and hot-remove ioctl commands issued by user-space RAID tools. In the md driver they give rise to corresponding internal diskop calls on the RAID personality drivers. Many more diskops exist than the basic three, however, and all have to be supported by the personality drivers.

In normal use, the md driver acts as a front-end which filters and then hands off block device read/write requests to the RAID personality driver associated with a RAID array. If the array is a type 1 (i.e. mirror) array, then the hand-off is to the raid1 driver. If components of the array are themselves other RAID arrays, then the raid1 driver will eventually pass requests back to the md driver, from where they will descend again to the components of the second array, and so on.
From the point of view of an author about to write a new RAID driver, it is sufficient to know only what the md driver expects to be implemented in a new personality driver. Exactly the functions detailed in Table II must be written, then assigned to the fields of the same name in a mdk_personality_t object, and then the object must be registered with the md driver via a register_md_personality() call at the time of the personality driver’s initialisation. The personality object must be unregistered via a unregister_md_personality() call at the driver’s termination.

The semantics of the methods of the mddev_t class (the objects which describe RAID arrays) listed in Table II are set out in Table III. They must also be implemented by the driver. The following is a fairly complete skeleton code for a personality driver, lacking only the implementations of the methods in the table:

```c
static mdk_personality_t my_personality = { /* method & field assignments */ }; #define MY_PERSONALITY 77 /* "77" is just for example */
static int __init my_init (void) {
    return register_md_personality (MY_PERSONALITY, &my_personality);
}
static void my_exit (void) {
    unregister_md_personality (MY_PERSONALITY);
}
module_init(my_init);
module_exit(my_exit);
```

In the remainder of this section, the action of each of these methods will be briefly sketched.

**Make request**

The make_request method constructs *mirrored requests* for the array components and submits them to the kernel’s block device subsystem. Both on read and on write, the raid1 driver makes a
Table III. The generic RAID personality driver methods (semantics).

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>make_request</td>
<td>Receives a buffer (as a buffer_head structure) and the command to write from or read to it (the details of the device offset and the number of bytes expected are in the buffer_head), and does so.</td>
</tr>
<tr>
<td>run</td>
<td>Assembles the RAID array and enables it, increments “in use” counter for the driver.</td>
</tr>
<tr>
<td>stop</td>
<td>Frees any resources obtained during the run() call for the RAID array and decrements the “in use” counter for the driver itself. It must return zero.</td>
</tr>
<tr>
<td>status</td>
<td>Prints a piece of status information into a buffer supplied in the call.</td>
</tr>
<tr>
<td>error_handler</td>
<td>Marks a component of the array inoperative. It returns zero if successful.</td>
</tr>
<tr>
<td>diskop</td>
<td>Handles array re-orderings and removals and additions. Its state argument designates the command type (for example, SPARE_ACTIVE) to be executed on a set mdp_disk_t of target disks, and manipulates the current array configuration meta-data.</td>
</tr>
<tr>
<td>stop_resync</td>
<td>Halts a resync thread – a kernel thread started by the md driver that brings a new mirror component up to date.</td>
</tr>
<tr>
<td>restart_resync</td>
<td>Restarts a resync thread that has previously been halted by stop_resync().</td>
</tr>
<tr>
<td>sync_request</td>
<td>Called by the resync thread in order to update a sequence of blocks. It decides the number for itself, and returns the number of sectors treated.</td>
</tr>
</tbody>
</table>

special master buffer head†† r1_bh. It holds the make_request call parameters. On read, the raid1 driver chooses one of the working array components, mirror_dev, sets the buffer head embedded in the master buffer head to point at it, sets the completion method for the buffer head to a special raid1_end_request function that will free the master buffer head, then submits a read request for the buffer head to the kernel’s block device layer. The request is constructed by the generic_make_request function from the now redirected buffer head, and the request structure submitted references it. Skeleton code for the function, both read and write parts, is shown in Figure I3.

On write (see Figure I1 and Figure I3 the raid1 driver manufactures a buffer head for each working component of the mirror. The manufactured buffer heads point back at the master buffer head in

---

††Buffer heads are specific kernel structures which contain a pointer to a memory area, a buffer. They represent a data buffer to the kernel, containing pointers to all the lists in which the buffer appears, and other accounting details.

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using speauth.cls
their private data field. The driver sets their completion methods to raid1_end_request and submits the buffer heads to the kernel’s block device layer. The raid1_end_request function decrements a counter in the master buffer head as each manufactured buffer head write I/O completes, and deallocates the master buffer head itself when the counter reaches zero, i.e. when the last of the mirrored buffer write I/O’s completes.

Run and stop

The run method performs startup tasks for the array. The raid1 driver copies the array metadata already assembled together by the md driver into a private meta-data area. This area is that pointed to by the field of the generic RAID array meta-data object named private. It is a type-opaque field and each personality driver usually puts a pointer to its own particular configuration meta-data structure there. Skeleton code for the run function is shown in Figure I4. Typically, certain initialisations are also performed in the run method. The driver’s own initialisation function
(the one marked with the "_init" modifier in the C source code) is generally used only to register the personality object with the md driver. The raid1 driver by way of initialisation in the run method, for example, preallocates a certain number of buffer head structures, and links them into a list of "free" buffer heads. It also calls the md driver to start a kernel thread that will later be used to perform certain resynchronisations, and increments the kernel's "in use" count for the driver itself.

Symmetrically, the stop method is responsible for shutdown tasks in the RAID array. The raid1 driver here frees all the buffer head structures it has preallocated, and unregisters and destroys the kernel thread object that it originally registered with the md driver during run(). Skeleton code is shown in Figure I4.

**Figure I4.** Skeleton RAID personality driver run, stop, status and error code.

---

Status, error and diskop

The status method is used to write array configuration data into a character buffer. The format is stylised to conform with that produced by other RAID personality drivers. It is incorporated in printout messages by the md driver. The skeleton code is shown in Figure I4. The information delivered by the method in the raid1 driver is the number of disks currently in the array and the number of them in a working state, plus a bitmap of the state per disk.

The error method is used to mark an array component as failed. It can decide to return unsuccessfully, for example when marking the component bad would destroy an array because there was no redundancy in reserve in that configuration. The raid1 driver in particular checks the
array and returns unsuccessfully if the named device has already been removed from the array. Skeleton code is shown in Figure I4.

The diskop method functions like an internal ioctl handler. It handles calls for hot-add, hot-remove, and set-faulty, and other internal manipulations of the array. These manipulations are very complex and cannot be detailed here.

Start and stop re-sync

The stop_resync and start_resync methods control a re-synchronisation process thread started by the md driver. In the raid1 driver, they call back to the md driver to stop and start it. Skeleton code for both functions is listed in Figure I5. The thread itself is created via a call to md_register_thread in the run method of the personality, and a pointer to it is kept in the array’s private configuration meta-data structure. The thread object is finally destroyed with a call to md_unregister_thread in the stop method of the personality, so there is precisely one resync thread per array.

Sync request

The sync_request method (see Figure I6) is called by the resync process thread from time to time in order to update a sequence of sectors. The call gives the first sector to update and then the method decides how many sectors to resync at that time and returns the number of sectors treated. The sync_request method is called repeatedly until synchronisation is complete. The raid1 driver does some special initialisations when this function is called for sector zero, which will be the first call in a resync sequence. Several progress counters are stored in the private raid1 configuration meta-data for the object representing the array, so they persist from call to call. The md driver itself throttles the sync_request calls carefully in order not to consume all the available I/O bandwidth. It takes into account information on all the extant I/O and resync threads.
In the raid1 driver, the sync_request function generates a read request for a chosen number of sectors from one of the mirror devices which is currently up to date. The request points to a specially generated raid1 “master” buffer head structure r1_bh with the current configuration metadata embedded in it. The read request is submitted to the kernel’s block device subsystem and completed asynchronously. When it completes, the end_sync_read method embedded in the request looks at the referenced meta-data and generates a write of the newly acquired buffer contents to all the mirror devices which are currently operational. When the write is completed to all operational mirrors, then accounting is updated.

APPENDIX II. The FR1 driver RAID modifications

The FR1 patch made a very concise set of changes to the standard Linux kernel md and raid1 drivers. These were easy to describe to the kernel RAID authors and maintainers. First, a set of changes to the md driver:

MD base patch

1. change the hot_add method so that it recognises a previous member of the array by the unique identifier (UUID) that forms part of its super-block area on disk (at the end of the device).

If the device UUID matches, then check the value of the events count in its super-block. If it is lower than it was at the activation of the bitmap, then act as in the original md and raid1 code – the device separated from the array before the present accounting trail came into existence.

If the events count matches or exceeds that at bitmap activation, then mark the array super-block in memory to signal that a “hot repair” is about to be done.
The intention is to trigger an intelligent resync, or *hot-repair*, when possible, instead of a complete resync, satisfying requirements 1., 2. and 4. of the list of design goals in Table II.

In order to allow testing or to force a hot-repair on a mirror without a persistent super-block on-disk, the md driver was also modified to:

2. Signal hot repair (by marking the array super-block in memory) when a hot-add of a component device immediately follows set-faulty without an intervening hot-remove.
   
The missing hot-remove is generated internally and the hot-add is then restarted from the beginning.

When the RAID device has no persistent super-block on disk, then a set-faulty followed immediately by hot-add triggers the added features (it was the only mechanism in the first version of the FR1 driver). The UUID and event count-based recognition mechanism was outlined by Neil Brown, one of the kernel RAID authors, in correspondence following the publication of the first version of the driver in patch format.

Item 5. in the list of requirements in Table II, automatic on- and off-lining of a component device in an array, also requires support in the md driver. The idea is that the RAID array should call a special ioctl on a component to tell it that it is now incorporated in a RAID array. The ioctl passes the address of a call-back function, and that function is used by the component whenever it changes status between working and not-working. The following patch was made to the md driver:

3. new ioctl calls (for BLKMDRGTR, BLKMDNTFY, and BLKMDUNTFY ioctls) were added to the bind_rdev_to_array and unbind_rdev_from_array functions which bind and unbind an array component to an array. These calls notify the component of its inclusion to or expulsion from a RAID array, and also register a call-back function with the component.
   
The *call-back function* was looks up the calling device by major and minor and runs set-faulty or hot-add for it.

The md driver will call with BLKMDRGTR at every inclusion and BLKMDUNTFY at every expulsion of the device from an array, and it will call with BLKMDRGTR at the very first inclusion of all. Then, on activation and disactivation of the component device through fault or otherwise, the component’s driver should use the call-back as follows:
if (md_count > 0 && md_notify_fn) {
    md_notify_fn (MKDEV(this_major,this_minor), active);
}

where the “active” parameter takes the value HOT_ADD_DISK and SET_DISK_FAULTY for good and bad respectively.

The above patch items are common to the FR5 (and other future such drivers) too. The remaining changes included in the FR1 patch are made to the raid1 driver alone:

**Raid1 base patch**

1. change the existing make_request method so that it marks its bitmap for a faulty component, instead of writing to the component, whenever the bitmap is active. If the bitmap is being dirtied for the first time, write the current value of the global kernel events counter in the array meta-data, so that later we can tell, when a disk comes back into the array, if it separated from the array before or after the bitmap started recording.

2. change the existing mark_disk_faulty function that is called by set-faulty so that it now activates the bitmap.

3. change the SPARE_ACTIVE case of the existing diskop method‡‡ to deactivate the bitmap if the array now has no non-operational devices.

4. change the existing resync_block function so that when handling the first sector it reads the RAID array super-block to check to see if the hot_add method in the md driver has indicated that we are doing a “hot repair”. If we are not, then the bitmap is deactivated and resync proceeds as it normally would. If it is a hot-repair, then the hot-repair flag in the array super-block is cleared, but during the remaining calls to resync_block the bitmap is consulted as to which blocks to really resync, and which to merely skip.

These changes are all in support of requirements 1. and 2. of the list of design goals in Table II. In addition, certain changes were made in order to potentiate asynchronous writes (requirement 3. in Table II). These are writes which are acknowledged to the kernel as having completed after only a single mirror component has been written to. The remaining components are updated asynchronously. The standard driver instead waits until all components have been updated before acknowledging:

**Asynchronous write patch**

1. An “async” boolean global was added to the driver, to control asynchronous writes. The existing make_request function was changed so that, when that global is set, a flag is set in each master buffer head generated by the driver in order to indicate that asynchronous completion is allowed for it.

‡‡SPARE_ACTIVE is the last internal diskop executed during the sequence of diskops corresponding to the action of hot-adding a component to the array. The case is called as a component is maneuvered into position into an array and after it has been successfully re-synced from the array data, so it indicates that the bitmap is no longer needed.

Copyright © 2003 John Wiley & Sons, Ltd.
Prepared using speauth.cls
2. The existing raid1_end_request function which completes I/O for the mirrored buffer heads on write was altered so as to acknowledge the original request back to the kernel on the first, instead of the last, of the mirrored buffer head I/O completions, when the flag appears in the master buffer head.

3. The existing raid1_make_request function was also changed to, when the bitmap is active and the async global is set, copy and save the buffer(s) referenced and mark the bitmap for the requested blocks, and flag the master buffer head generated to say that asynchronous I/O is now planned for that request. If the copy is unsuccessful (if, say, the kernel is short of memory), then the flag is not set and synchronous I/O is done instead.

4. The existing raid1_end_bh_io function, which is run from the completion function of the last mirrored buffer head to be treated, was changed to make it do an acknowledgement back to the kernel for the original request only conditionally, instead of unconditionally. The condition is that the corresponding flag in the master buffer head has not yet been set. The flag indicates that the original request has already been acknowledged.

5. The raid1_end_bh_io function was also changed to free the copied and saved buffer if the async flag is set. It already clears the bitmap as a result of the rest of the patch changes.

In addition, certain patches were required for correct reporting of the statistics during resynchronisation, which in turn control the point at which throttling is applied:

**Throttle correction patch**

This patch was added after several releases of the driver.

1. Add an array to the md driver containing a “discount” block count for each possible RAID device.

   Duplicate the calculation of the “current speed” in the md driver to give a “real speed” in which the discounted blocks are not counted in the calculation, and change the throttle trigger to use the real speed instead of the current speed.

2. Change the raid1 driver raid1_sync_request code to add the number of blocks skipped during a resynchronisation into the discount blocks count for that RAID device.
With this change the throttle will be applied only when the real I/O speed during resynchronisation exceeds the stipulated maximum. Without the patch the speed throttle will be applied inappropriately, when the resynchronisation is doing no real work, only skipping blocks marked clean in the bitmap.

**Read balance patch**

This patch was also added after many releases, in response to experience gained. It deals with the situation where one RAID mirror component is much slower than the others, usually because it is a network block device, accessing a remote disk over the net.

The existing raid1 driver switches the component device it reads from every 128 requests. The patch

1. adds code to the raid1_end_request functions to calculate the time taken for the buffer I/O to complete, and adds the time into a rolling average for the latency. The rolling average is kept in an extra field added to the raid1_private_data structure and the start time for each request is kept in an extra field of the raid1_bh structure used to control the read;
2. changes the code in the make_request function that selects the read device so that it chooses the device with the lowest recorded latency every time, and to every 1024 requests carry out a a testing regimen of 10 requests to each component device in order to make sure that the latency is computed for all devices.

**Robust read patch**

A further patch has recently been added that treats the problem that a disk is expelled from a Linux software RAID mirror array on every error. This begins to be troublesome for hard disks, since hardware read errors are relatively easy to provoke – on a write error, the error will be covered by the disk itself, which will invisibly swap in a new sector to substitute the old, but on read nothing can be done. On networked disks it is extremely troublesome, but hidden by the automatic reincorporation added by the FR1 patch.

The patch makes a read error on one component of the mirror be retried on another component, for as many components as there are in the array, without the disk ever being expelled. If all components error, an error is returned, but no disk is expelled even then.

An untried experimental section of the patch (not enabled by default) causes a rewrite of the data to all components to follow a successful retried read, thus mending the defect where it is possible.

That concludes the description of the FR1 patch, as submitted to the Linux RAID mailing lists.