Software−RAID−HOWTO

Table of Contents

Table of Contents

Table of Contents

The Software−RAID HOWTO

Jakob Østergaard `jakob@unthought.net` and Emilio Bueso `bueso@vives.org`

v1.1, 2004−06−03

This HOWTO describes how to use Software RAID under Linux. It addresses a specific version of the Software RAID layer, namely the 0.90 RAID layer made by Ingo Molnar and others. This is the RAID layer that is the standard in Linux−2.4, and it is the version that is also used by Linux−2.2 kernels shipped from some vendors. The 0.90 RAID support is available as patches to Linux−2.0 and Linux−2.2, and is by many considered far more stable that the older RAID support already in those kernels.

---

1. Introduction

2. Why RAID?

3. Devices

4. Hardware issues

5. RAID setup

6. Detecting, querying and testing

7. Tweaking, tuning and troubleshooting

8. Reconstruction

9. Performance

10. Related tools

11. Partitioning RAID / LVM on RAID

12. Credits

13. Changelog

- 13.1 Version 1.1

---

1. Introduction

This HOWTO describes the "new−style" RAID present in the 2.4 and 2.6 kernel series only. It does *not* describe the "old−style" RAID functionality present in 2.0 and 2.2 kernels.

The home site for this HOWTO is http://unthought.net/Software−RAID.HOWTO/, where updated versions appear first. The howto was originally written by Jakob Østergaard based on a large number of emails between the author and Ingo Molnar (mingo@chiara.csoma.elte.hu) −− one of the RAID developers −−, the linux−raid mailing list (linux−raid@vger.kernel.org) and various other people. Emilio Bueso (bueso@vives.org) co−wrote the 1.0 version.

If you want to use the new−style RAID with 2.0 or 2.2 kernels, you should get a patch for your kernel, from http://people.redhat.com/mingo/ The standard 2.2 kernels does not have direct support for the new−style RAID described in this HOWTO. Therefore these patches are needed. *The old−style RAID support in standard 2.0 and 2.2 kernels is buggy and lacks several important features present in the new−style RAID software.*

Some of the information in this HOWTO may seem trivial, if you know RAID all ready. Just skip those parts.

1.1 Disclaimer

The mandatory disclaimer:

All information herein is presented "as−is", with no warranties expressed nor implied. If you lose all your data, your job, get hit by a truck, whatever, it's not my fault, nor the developers'. Be aware, that you use the RAID software and this information at your own risk! There is no guarantee whatsoever, that any of the software, or this information, is in any way correct, nor suited for any use whatsoever. Back up all your data before experimenting with this. Better safe than sorry.

1.2 What is RAID?

In 1987, the University of California Berkeley, published an article entitled A Case for Redundant Arrays of Inexpensive Disks (RAID). This article described various types of disk arrays, referred to by the acronym RAID. The basic idea of RAID was to combine multiple small, independent disk drives into an array of disk drives which yields performance exceeding that of a Single Large Expensive Drive (SLED). Additionally, this array of drives appears to the computer as a single logical storage unit or drive.

The Mean Time Between Failure (MTBF) of the array will be equal to the MTBF of an individual drive, divided by the number of drives in the array. Because of this, the MTBF of an array of drives would be too low for many application requirements. However, disk arrays can be made fault−tolerant by redundantly storing information in various ways.

Five types of array architectures, RAID−1 through RAID−5, were defined by the Berkeley paper, each providing disk fault−tolerance and each offering different trade−offs in features and performance. In addition to these five redundant array architectures, it has become popular to refer to a non−redundant array of disk drives as a RAID−0 array.

Today some of the original RAID levels (namely level 2 and 3) are only used in very specialized systems (and in fact not even supported by the Linux Software RAID drivers). Another level, "linear" has emerged, and especially RAID level 0 is often combined with RAID level 1.

1.3 Terms

In this HOWTO the word "RAID" means "Linux Software RAID". This HOWTO does not treat any aspects of Hardware RAID. Furthermore, it does not treat any aspects of Software RAID in other operating system kernels.

When describing RAID setups, it is useful to refer to the number of disks and their sizes. At all times the letter **N** is used to denote the number of active disks in the array (not counting spare−disks). The letter **S** is the size of the smallest drive in the array, unless otherwise mentioned. The letter **P** is used as the performance of one disk in the array, in MB/s. When used, we assume that the disks are equally fast, which may not always be true in real−world scenarios.

Note that the words "device" and "disk" are supposed to mean about the same thing. Usually the devices that are used to build a RAID device are partitions on disks, not necessarily entire disks. But combining several partitions on one disk usually does not make sense, so the words devices and disks just mean "partitions on different disks".

1.4 The RAID levels

Here's a short description of what is supported in the Linux RAID drivers. Some of this information is absolutely basic RAID info, but I've added a few notices about what's special in the Linux implementation of the levels. You can safely skip this section if you know RAID already.

The current RAID drivers in Linux supports the following levels:

- **Linear mode**
    - Two or more disks are combined into one physical device. The disks are "appended" to each other, so writing linearly to the RAID device will fill up disk 0 first, then disk 1 and so on. The disks does not have to be of the same size. In fact, size doesn't matter at all here :)
    - There is no redundancy in this level. If one disk crashes you will most probably lose all your data. You can however be lucky to recover some data, since the filesystem will just be missing one large consecutive chunk of data.
    - The read and write performance will not increase for single reads/writes. But if several users use the device, you may be lucky that one user effectively is using the first disk, and the other user is accessing files which happen to reside on the second disk. If that happens, you will see a performance gain.
- **RAID−0**
    - Also called "stripe" mode. The devices should (but need not) have the same size. Operations on the array will be split on the devices; for example, a large write could be split up as 4 kB to disk 0, 4 kB to disk 1, 4 kB to disk 2, then 4 kB to disk 0 again, and so on. If one device is much larger than the other devices, that extra space is still utilized in the RAID device, but

you will be accessing this larger disk alone, during writes in the high end of your RAID device. This of course hurts performance.

  - Like linear, there is no redundancy in this level either. Unlike linear mode, you will not be able to rescue any data if a drive fails. If you remove a drive from a RAID−0 set, the RAID device will not just miss one consecutive block of data, it will be filled with small holes all over the device. e2fsck or other filesystem recovery tools will probably not be able to recover much from such a device.
  - The read and write performance will increase, because reads and writes are done in parallel on the devices. This is usually the main reason for running RAID−0. If the busses to the disks are fast enough, you can get very close to N*P MB/sec.
- **RAID−1**
  - This is the first mode which actually has redundancy. RAID−1 can be used on two or more disks with zero or more spare−disks. This mode maintains an exact mirror of the information on one disk on the other disk(s). Of Course, the disks must be of equal size. If one disk is larger than another, your RAID device will be the size of the smallest disk.
  - If up to N−1 disks are removed (or crashes), all data are still intact. If there are spare disks available, and if the system (eg. SCSI drivers or IDE chipset etc.) survived the crash, reconstruction of the mirror will immediately begin on one of the spare disks, after detection of the drive fault.
  - Write performance is often worse than on a single device, because identical copies of the data written must be sent to every disk in the array. With large RAID−1 arrays this can be a real problem, as you may saturate the PCI bus with these extra copies. This is in fact one of the very few places where Hardware RAID solutions can have an edge over Software solutions − if you use a hardware RAID card, the extra write copies of the data will not have to go over the PCI bus, since it is the RAID controller that will generate the extra copy. Read performance is good, especially if you have multiple readers or seek−intensive workloads. The RAID code employs a rather good read−balancing algorithm, that will simply let the disk whose heads are closest to the wanted disk position perform the read operation. Since seek operations are relatively expensive on modern disks (a seek time of 6 ms equals a read of 123 kB at 20 MB/sec), picking the disk that will have the shortest seek time does actually give a noticeable performance improvement.
- **RAID−4**
  - This RAID level is not used very often. It can be used on three or more disks. Instead of completely mirroring the information, it keeps parity information on one drive, and writes data to the other disks in a RAID−0 like way. Because one disk is reserved for parity information, the size of the array will be (N−1)*S, where S is the size of the smallest drive in the array. As in RAID−1, the disks should either be of equal size, or you will just have to accept that the S in the (N−1)*S formula above will be the size of the smallest drive in the array.
  - If one drive fails, the parity information can be used to reconstruct all data. If two drives fail, all data is lost.
  - The reason this level is not more frequently used, is because the parity information is kept on one drive. This information must be updated *every* time one of the other disks are written to. Thus, the parity disk will become a bottleneck, if it is not a lot faster than the other disks. However, if you just happen to have a lot of slow disks and a very fast one, this RAID level can be very useful.
- **RAID−5**
  - This is perhaps the most useful RAID mode when one wishes to combine a larger number of physical disks, and still maintain some redundancy. RAID−5 can be used on three or more disks, with zero or more spare−disks. The resulting RAID−5 device size will be (N−1)*S, just like RAID−4. The big difference between RAID−5 and −4 is, that the parity information is

distributed evenly among the participating drives, avoiding the bottleneck problem in RAID−4.

- If one of the disks fail, all data are still intact, thanks to the parity information. If spare disks are available, reconstruction will begin immediately after the device failure. If two disks fail simultaneously, all data are lost. RAID−5 can survive one disk failure, but not two or more.
- Both read and write performance usually increase, but can be hard to predict how much. Reads are similar to RAID−0 reads, writes can be either rather expensive (requiring read−in prior to write, in order to be able to calculate the correct parity information), or similar to RAID−1 writes. The write efficiency depends heavily on the amount of memory in the machine, and the usage pattern of the array. Heavily scattered writes are bound to be more expensive.

1.5 Requirements

This HOWTO assumes you are using Linux 2.4 or later. However, it is possible to use Software RAID in late 2.2.x or 2.0.x Linux kernels with a matching RAID patch and the 0.90 version of the raidtools. Both the patches and the tools can be found at http://people.redhat.com/mingo/. The RAID patch, the raidtools package, and the kernel should all match as close as possible. At times it can be necessary to use older kernels if raid patches are not available for the latest kernel.

If you use and recent GNU/Linux distribution based on the 2.4 kernel or later, your system most likely already has a matching version of the raidtools for your kernel.

---

2. Why RAID?

There can be many good reasons for using RAID. A few are; the ability to combine several physical disks into one larger "virtual" device, performance improvements, and redundancy.

It is, however, very important to understand that RAID is not a substitute for good backups. Some RAID levels will make your systems immune to data loss from single−disk failures, but RAID will not allow you to recover from an accidental `"rm -rf /"`. RAID will also not help you preserve your data if the server holding the RAID itself is lost in one way or the other (theft, flooding, earthquake, Martian invasion etc.)

RAID will generally allow you to keep systems up and running, in case of common hardware problems (single disk failure). It is **not** in itself a complete data safety solution. This is very important to realize.

2.1 Device and filesystem support

Linux RAID can work on most block devices. It doesn't matter whether you use IDE or SCSI devices, or a mixture. Some people have also used the Network Block Device (NBD) with more or less success.

Since a Linux Software RAID device is itself a block device, the above implies that you can actually *create a RAID of other RAID devices*. This in turn makes it possible to support RAID−10 (RAID−0 of multiple RAID−1 devices), simply by using the RAID−0 and RAID−1 functionality together. Other more exotic configurations, such a RAID−5 over RAID−5 "matrix" configurations are equally supported.

The RAID layer has absolutely nothing to do with the filesystem layer. You can put any filesystem on a RAID device, just like any other block device.

2.2 Performance

Often RAID is employed as a solution to performance problems. While RAID can indeed often be the solution you are looking for, it is not a silver bullet. There can be many reasons for performance problems, and RAID is only the solution to a few of them.

See Chapter one for a mention of the performance characteristics of each level.

2.3 Swapping on RAID

There's no reason to use RAID for swap performance reasons. The kernel itself can stripe swapping on several devices, if you just give them the same priority in the `/etc/fstab` file.

A nice `/etc/fstab` looks like:

```
/dev/sda2       swap           swap    defaults,pri=1   0 0
/dev/sdb2       swap           swap    defaults,pri=1   0 0
/dev/sdc2       swap           swap    defaults,pri=1   0 0
/dev/sdd2       swap           swap    defaults,pri=1   0 0
/dev/sde2       swap           swap    defaults,pri=1   0 0
/dev/sdf2       swap           swap    defaults,pri=1   0 0
/dev/sdg2       swap           swap    defaults,pri=1   0 0
```

This setup lets the machine swap in parallel on seven SCSI devices. No need for RAID, since this has been a kernel feature for a long time.

Another reason to use RAID for swap is high availability. If you set up a system to boot on eg. a RAID−1 device, the system should be able to survive a disk crash. But if the system has been swapping on the now faulty device, you will for sure be going down. Swapping on a RAID−1 device would solve this problem.

There has been a lot of discussion about whether swap was stable on RAID devices. This is a continuing debate, because it depends highly on other aspects of the kernel as well. As of this writing, it seems that swapping on RAID should be perfectly stable, you should however stress−test the system yourself until you are satisfied with the stability.

You can set up RAID in a swap file on a filesystem on your RAID device, or you can set up a RAID device as a swap partition, as you see fit. As usual, the RAID device is just a block device.

2.4 Why mdadm?

The classic raidtools are the standard software RAID management tool for Linux, so using mdadm is not a must.

However, if you find raidtools cumbersome or limited, mdadm (multiple devices admin) is an extremely useful tool for running RAID systems. It can be used as a replacement for the raidtools, or as a supplement.

The mdadm tool, written by Neil Brown, a software engineer at the University of New South Wales and a kernel developer, is now at version 1.4.0 and has proved to be quite stable. There is much positive response on the Linux−raid mailing list and mdadm is likely to become widespread in the future.

The main differences between mdadm and raidtools are:

* mdadm can diagnose, monitor and gather detailed information about your arrays
* mdadm is a single centralized program and not a collection of disperse programs, so there's a common syntax for every RAID management command
* mdadm can perform almost all of its functions without having a configuration file and does not use one by default
* Also, if a configuration file is needed, mdadm will help with management of it's contents

3. Devices

Software RAID devices are so−called "block" devices, like ordinary disks or disk partitions. A RAID device is "built" from a number of other block devices − for example, a RAID−1 could be built from two ordinary disks, or from two disk partitions (on separate disks − please see the description of RAID−1 for details on this).

There are no other special requirements to the devices from which you build your RAID devices − this gives you a lot of freedom in designing your RAID solution. For example, you can build a RAID from a mix of IDE and SCSI devices, and you can even build a RAID from other RAID devices (this is useful for RAID−0+1, where you simply construct two RAID−1 devices from ordinary disks, and finally construct a RAID−0 device from those two RAID−1 devices).

Therefore, in the following text, we will use the word "device" as meaning "disk", "partition", or even "RAID device". A "device" in the following text simply refers to a "Linux block device". It could be anything from a SCSI disk to a network block device. We will commonly refer to these "devices" simply as "disks", because that is what they will be in the common case.

However, there are several roles that devices can play in your arrays. A device could be a "spare disk", it could have failed and thus be a "faulty disk", or it could be a normally working and fully functional device actively used by the array.

In the following we describe two special types of devices; namely the "spare disks" and the "faulty disks".

3.1 Spare disks

Spare disks are disks that do not take part in the RAID set until one of the active disks fail. When a device failure is detected, that device is marked as "bad" and reconstruction is immediately started on the first spare−disk available.

Thus, spare disks add a nice extra safety to especially RAID−5 systems that perhaps are hard to get to (physically). One can allow the system to run for some time, with a faulty device, since all redundancy is preserved by means of the spare disk.

You cannot be sure that your system will keep running after a disk crash though. The RAID layer should handle device failures just fine, but SCSI drivers could be broken on error handling, or the IDE chipset could lock up, or a lot of other things could happen.

Also, once reconstruction to a hot−spare begins, the RAID layer will start reading from all the other disks to re−create the redundant information. If multiple disks have built up bad blocks over time, the reconstruction itself can actually trigger a failure on one of the "good" disks. This will lead to a complete RAID failure. If you do frequent backups of the entire filesystem on the RAID array, then it is highly unlikely that you would ever get in this situation − this is another very good reason for taking frequent backups. Remember, RAID is not a substitute for backups.

3.2 Faulty disks

When the RAID layer handles device failures just fine, crashed disks are marked as faulty, and reconstruction is immediately started on the first spare−disk available.

Faulty disks still appear and behave as members of the array. The RAID layer just treats crashed devices as inactive parts of the filesystem.

---

4. Hardware issues

This section will mention some of the hardware concerns involved when running software RAID.

If you are going after high performance, you should make sure that the bus(ses) to the drives are fast enough. You should not have 14 UW−SCSI drives on one UW bus, if each drive can give 20 MB/s and the bus can only sustain 160 MB/s. Also, you should only have one device per IDE bus. Running disks as master/slave is horrible for performance. IDE is really bad at accessing more that one drive per bus. Of Course, all newer motherboards have two IDE busses, so you can set up two disks in RAID without buying more controllers. Extra IDE controllers are rather cheap these days, so setting up 6−8 disk systems with IDE is easy and affordable.

4.1 IDE Configuration

It is indeed possible to run RAID over IDE disks. And excellent performance can be achieved too. In fact, today's price on IDE drives and controllers does make IDE something to be considered, when setting up new RAID systems.

- **Physical stability:** IDE drives has traditionally been of lower mechanical quality than SCSI drives. Even today, the warranty on IDE drives is typically one year, whereas it is often three to five years on SCSI drives. Although it is not fair to say, that IDE drives are per definition poorly made, one should be aware that IDE drives of *some* brand *may* fail more often that similar SCSI drives. However, other brands use the exact same mechanical setup for both SCSI and IDE drives. It all boils down to: All disks fail, sooner or later, and one should be prepared for that.
- **Data integrity:** Earlier, IDE had no way of assuring that the data sent onto the IDE bus would be the same as the data actually written to the disk. This was due to total lack of parity, checksums, etc. With the Ultra−DMA standard, IDE drives now do a checksum on the data they receive, and thus it becomes highly unlikely that data get corrupted. The PCI bus however, does not have parity or checksum, and that bus is used for both IDE and SCSI systems.
- **Performance:** I am not going to write thoroughly about IDE performance here. The really short story is:
    - IDE drives are fast, although they are not (as of this writing) found in 10.000 or 15.000 rpm versions as their SCSI counterparts
    - IDE has more CPU overhead than SCSI (but who cares?)
    - Only use **one** IDE drive per IDE bus, slave disks spoil performance

* **Fault survival:** The IDE driver usually survives a failing IDE device. The RAID layer will mark the disk as failed, and if you are running RAID levels 1 or above, the machine should work just fine until you can take it down for maintenance.

It is **very** important, that you only use **one** IDE disk per IDE bus. Not only would two disks ruin the performance, but the failure of a disk often guarantees the failure of the bus, and therefore the failure of all disks on that bus. In a fault−tolerant RAID setup (RAID levels 1,4,5), the failure of one disk can be handled, but the failure of two disks (the two disks on the bus that fails due to the failure of the one disk) will render the array unusable. Also, when the master drive on a bus fails, the slave or the IDE controller may get awfully confused. One bus, one drive, that's the rule.

There are cheap PCI IDE controllers out there. You often get two or four busses for around $80. Considering the much lower price of IDE disks versus SCSI disks, an IDE disk array can often be a really nice solution if one can live with the relatively low number (around 8 probably) of disks one can attach to a typical system.

IDE has major cabling problems when it comes to large arrays. Even if you had enough PCI slots, it's unlikely that you could fit much more than 8 disks in a system and still get it running without data corruption caused by too long IDE cables.

Furthermore, some of the newer IDE drives come with a restriction that they are only to be used a given number of hours per day. These drives are meant for desktop usage, and it **can** lead to severe problems if these are used in a 24/7 server RAID environment.

4.2 Hot Swap

Although hot swapping of drives is supported to some extent, it is still not something one can do easily.

Hot−swapping IDE drives

Don't ! IDE doesn't handle hot swapping at all. Sure, it may work for you, if your IDE driver is compiled as a module (only possible in the 2.2 series of the kernel), and you re−load it after you've replaced the drive. But you may just as well end up with a fried IDE controller, and you'll be looking at a lot more down−time than just the time it would have taken to replace the drive on a downed system.

The main problem, except for the electrical issues that can destroy your hardware, is that the IDE bus must be re−scanned after disks are swapped. While newer Linux kernels do support re−scan of an IDE bus (with the help of the hdparm utility), re−detecting partitions is still something that is lacking. If the new disk is 100% identical to the old one (wrt. geometry etc.), it *may* work, but really, you are walking the bleeding edge here.

Hot−swapping SCSI drives

Normal SCSI hardware is not hot−swappable either. It **may** however work. If your SCSI driver supports re−scanning the bus, and removing and appending devices, you may be able to hot−swap devices. However, on a normal SCSI bus you probably shouldn't unplug devices while your system is still powered up. But then again, it may just work (and you may end up with fried hardware).

The SCSI layer **should** survive if a disk dies, but not all SCSI drivers handle this yet. If your SCSI driver dies when a disk goes down, your system will go with it, and hot−plug isn't really interesting then.

Hot−swapping with SCA

With SCA, it is possible to hot−plug devices. Unfortunately, this is not as simple as it should be, but it is both possible and safe.

Replace the RAID device, disk device, and host/channel/id/lun numbers with the appropriate values in the example below:

- Dump the partition table from the drive, if it is still readable:

  ```
  sfdisk −d /dev/sdb > partitions.sdb
  ```
- Remove the drive to replace from the array:

  ```
  raidhotremove /dev/md0 /dev/sdb1
  ```
- Look up the Host, Channel, ID and Lun of the drive to replace, by looking in

  ```
  /proc/scsi/scsi
  ```
- Remove the drive from the bus:

  ```
  echo "scsi remove−single−device 0 0 2 0" > /proc/scsi/scsi
  ```
- Verify that the drive has been correctly removed, by looking in

  ```
  /proc/scsi/scsi
  ```
- Unplug the drive from your SCA bay, and insert a new drive
- Add the new drive to the bus:

  ```
  echo "scsi add−single−device 0 0 2 0" > /proc/scsi/scsi
  ```

  (this should spin up the drive as well)
- Re−partition the drive using the previously dumped partition table:

  ```
  sfdisk /dev/sdb < partitions.sdb
  ```
- Add the drive to your array:

  ```
  raidhotadd /dev/md0 /dev/sdb2
  ```

The arguments to the "scsi remove−single−device" commands are: Host, Channel, Id and Lun. These numbers are found in the "/proc/scsi/scsi" file.

The above steps have been tried and tested on a system with IBM SCA disks and an Adaptec SCSI controller. If you encounter problems or find easier ways to do this, please discuss this on the linux−raid mailing list.

---

5. RAID setup

5.1 General setup

This is what you need for any of the RAID levels:

- A kernel. Preferably a kernel from the 2.4 series. Alternatively a 2.0 or 2.2 kernel with the RAID patches applied.
- The RAID tools.

- Patience, Pizza, and your favorite caffeinated beverage.

All of this is included as standard in most GNU/Linux distributions today.

If your system has RAID support, you should have a file called `/proc/mdstat`. Remember it, that file is your friend. If you do not have that file, maybe your kernel does not have RAID support. See what the contains, by doing a `cat /proc/mdstat`. It should tell you that you have the right RAID personality (eg. RAID mode) registered, and that no RAID devices are currently active.

Create the partitions you want to include in your RAID set.

5.2 Downloading and installing the RAID tools

The RAID tools are included in almost every major Linux distribution.

IMPORTANT: If using Debian Woody (3.0) or later, you can install the package by running

```
apt−get install raidtools2
```

This `raidtools2` is a modern version of the old `raidtools` package, which doesn't support the persistent−superblock and parity−algorithm settings.

5.3 Downloading and installing mdadm

You can download the most recent mdadm tarball at http://www.cse.unsw.edu.au/~neilb/source/mdadm/. Issue a nice `make install` to compile and then install mdadm and its documentation, manual pages and example files.

```
tar xvf ./mdadm−1.4.0.tgz
cd mdadm−1.4.0.tgz
make install
```

If using an RPM−based distribution, you can download and install the package file found at http://www.cse.unsw.edu.au/~neilb/source/mdadm/RPM.

```
rpm −ihv mdadm−1.4.0−1.i386.rpm
```

If using Debian Woody (3.0) or later, you can install the package by running

```
apt−get install mdadm
```

Gentoo has this package available in the portage tree. There you can run

```
emerge mdadm
```

Other distributions may also have this package available. Now, let's go mode−specific.

5.4 Linear mode

Ok, so you have two or more partitions which are not necessarily the same size (but of course can be), which you want to append to each other.

Set up the `/etc/raidtab` file to describe your setup. I set up a raidtab for two disks in linear mode, and the file looked like this:

```
raiddev /dev/md0
        raid-level      linear
        nr-raid-disks   2
        chunk-size      32
        persistent-superblock 1
        device          /dev/sdb6
        raid-disk       0
        device          /dev/sdc5
        raid-disk       1
```

Spare−disks are not supported here. If a disk dies, the array dies with it. There's no information to put on a spare disk.

You're probably wondering why we specify a `chunk-size` here when linear mode just appends the disks into one large array with no parallelism. Well, you're completely right, it's odd. Just put in some chunk size and don't worry about this any more.

Ok, let's create the array. Run the command

```
  mkraid /dev/md0
```

This will initialize your array, write the persistent superblocks, and start the array.

If you are using mdadm, a single command like

```
   mdadm --create --verbose /dev/md0 --level=linear --raid-devices=2 /dev/sdb6 /dev/sdc5
```

should create the array. The parameters talk for themselves. The output might look like this

```
  mdadm: chunk size defaults to 64K
  mdadm: array /dev/md0 started.
```

Have a look in `/proc/mdstat`. You should see that the array is running.

Now, you can create a filesystem, just like you would on any other device, mount it, include it in your `/etc/fstab` and so on.

5.5 RAID−0

You have two or more devices, of approximately the same size, and you want to combine their storage capacity and also combine their performance by accessing them in parallel.

Set up the `/etc/raidtab` file to describe your configuration. An example raidtab looks like:

```
raiddev /dev/md0
        raid-level      0
        nr-raid-disks   2
        persistent-superblock 1
        chunk-size     4
```

```
        device          /dev/sdb6
        raid-disk       0
        device          /dev/sdc5
        raid-disk       1
```

Like in Linear mode, spare disks are not supported here either. RAID−0 has no redundancy, so when a disk dies, the array goes with it.

Again, you just run

```
  mkraid /dev/md0
```

to initialize the array. This should initialize the superblocks and start the raid device. Have a look in `/proc/mdstat` to see what's going on. You should see that your device is now running.

/dev/md0 is now ready to be formatted, mounted, used and abused.

5.6 RAID−1

You have two devices of approximately same size, and you want the two to be mirrors of each other. Eventually you have more devices, which you want to keep as stand−by spare−disks, that will automatically become a part of the mirror if one of the active devices break.

Set up the `/etc/raidtab` file like this:

```
raiddev /dev/md0
        raid-level      1
        nr-raid-disks   2
        nr-spare-disks  0
        persistent-superblock 1
        device          /dev/sdb6
        raid-disk       0
        device          /dev/sdc5
        raid-disk       1
```

If you have spare disks, you can add them to the end of the device specification like

```
        device          /dev/sdd5
        spare-disk      0
```

Remember to set the `nr-spare-disks` entry correspondingly.

Ok, now we're all set to start initializing the RAID. The mirror must be constructed, eg. the contents (however unimportant now, since the device is still not formatted) of the two devices must be synchronized.

Issue the

```
  mkraid /dev/md0
```

command to begin the mirror initialization.

Check out the `/proc/mdstat` file. It should tell you that the /dev/md0 device has been started, that the mirror is being reconstructed, and an ETA of the completion of the reconstruction.

Reconstruction is done using idle I/O bandwidth. So, your system should still be fairly responsive, although your disk LEDs should be glowing nicely.

The reconstruction process is transparent, so you can actually use the device even though the mirror is currently under reconstruction.

Try formatting the device, while the reconstruction is running. It will work. Also you can mount it and use it while reconstruction is running. Of Course, if the wrong disk breaks while the reconstruction is running, you're out of luck.

5.7 RAID−4

Note! I haven't tested this setup myself. The setup below is my best guess, not something I have actually had up running. If you use RAID−4, please write to the author and share your experiences.

You have three or more devices of roughly the same size, one device is significantly faster than the other devices, and you want to combine them all into one larger device, still maintaining some redundancy information. Eventually you have a number of devices you wish to use as spare−disks.

Set up the `/etc/raidtab` file like this:

```
raiddev /dev/md0
        raid-level      4
        nr-raid-disks   4
        nr-spare-disks  0
        persistent-superblock 1
        chunk-size      32
        device          /dev/sdb1
        raid-disk       0
        device          /dev/sdc1
        raid-disk       1
        device          /dev/sdd1
        raid-disk       2
        device          /dev/sde1
        raid-disk       3
```

If we had any spare disks, they would be inserted in a similar way, following the raid−disk specifications;

```
        device         /dev/sdf1
        spare-disk     0
```

as usual.

Your array can be initialized with the

```
   mkraid /dev/md0
```

command as usual.

You should see the section on special options for mke2fs before formatting the device.

5.8 RAID−5

You have three or more devices of roughly the same size, you want to combine them into a larger device, but still to maintain a degree of redundancy for data safety. Eventually you have a number of devices to use as spare−disks, that will not take part in the array before another device fails.

If you use N devices where the smallest has size S, the size of the entire array will be (N−1)*S. This "missing" space is used for parity (redundancy) information. Thus, if any disk fails, all data stay intact. But if two disks fail, all data is lost.

Set up the `/etc/raidtab` file like this:

```
raiddev /dev/md0
        raid-level      5
        nr-raid-disks   7
        nr-spare-disks  0
        persistent-superblock 1
        parity-algorithm        left-symmetric
        chunk-size      32
        device          /dev/sda3
        raid-disk       0
        device          /dev/sdb1
        raid-disk       1
        device          /dev/sdc1
        raid-disk       2
        device          /dev/sdd1
        raid-disk       3
        device          /dev/sde1
        raid-disk       4
        device          /dev/sdf1
        raid-disk       5
        device          /dev/sdg1
        raid-disk       6
```

If we had any spare disks, they would be inserted in a similar way, following the raid−disk specifications;

```
        device         /dev/sdh1
        spare-disk     0
```

And so on.

A chunk size of 32 kB is a good default for many general purpose filesystems of this size. The array on which the above raidtab is used, is a 7 times 6 GB = 36 GB (remember the (n−1)*s = (7−1)*6 = 36) device. It holds an ext2 filesystem with a 4 kB block size. You could go higher with both array chunk−size and filesystem block−size if your filesystem is either much larger, or just holds very large files.

Ok, enough talking. You set up the `/etc/raidtab`, so let's see if it works. Run the

```
  mkraid /dev/md0
```

command, and see what happens. Hopefully your disks start working like mad, as they begin the reconstruction of your array. Have a look in `/proc/mdstat` to see what's going on.

If the device was successfully created, the reconstruction process has now begun. Your array is not consistent until this reconstruction phase has completed. However, the array is fully functional (except for the handling of device failures of course), and you can format it and use it even while it is reconstructing.

See the section on special options for mke2fs before formatting the array.

Ok, now when you have your RAID device running, you can always stop it or re−start it using the

```
raidstop /dev/md0
```

or

```
raidstart /dev/md0
```

commands.

With mdadm you can stop the device using

```
mdadm −S /dev/md0
```

and re−start it with

```
mdadm −R /dev/md0
```

Instead of putting these into init−files and rebooting a zillion times to make that work, read on, and get autodetection running.

5.9 The Persistent Superblock

Back in "The Good Old Days" (TM), the raidtools would read your `/etc/raidtab` file, and then initialize the array. However, this would require that the filesystem on which `/etc/raidtab` resided was mounted. This is unfortunate if you want to boot on a RAID.

Also, the old approach led to complications when mounting filesystems on RAID devices. They could not be put in the `/etc/fstab` file as usual, but would have to be mounted from the init−scripts.

The persistent superblocks solve these problems. When an array is initialized with the `persistent−superblock` option in the `/etc/raidtab` file, a special superblock is written in the beginning of all disks participating in the array. This allows the kernel to read the configuration of RAID devices directly from the disks involved, instead of reading from some configuration file that may not be available at all times.

You should however still maintain a consistent `/etc/raidtab` file, since you may need this file for later reconstruction of the array.

The persistent superblock is mandatory if you want auto−detection of your RAID devices upon system boot. This is described in the **Autodetection** section.

5.10 Chunk sizes

The chunk−size deserves an explanation. You can never write completely parallel to a set of disks. If you had two disks and wanted to write a byte, you would have to write four bits on each disk, actually, every second bit would go to disk 0 and the others to disk 1. Hardware just doesn't support that. Instead, we choose some chunk−size, which we define as the smallest "atomic" mass of data that can be written to the devices. A write of 16 kB with a chunk size of 4 kB, will cause the first and the third 4 kB chunks to be written to the first disk, and the second and fourth chunks to be written to the second disk, in the RAID−0 case with two disks. Thus, for large writes, you may see lower overhead by having fairly large chunks, whereas arrays that are primarily holding small files may benefit more from a smaller chunk size.

Chunk sizes must be specified for all RAID levels, including linear mode. However, the chunk−size does not make any difference for linear mode.

For optimal performance, you should experiment with the value, as well as with the block−size of the filesystem you put on the array.

The argument to the chunk−size option in `/etc/raidtab` specifies the chunk−size in kilobytes. So "4" means "4 kB".

RAID−0

Data is written "almost" in parallel to the disks in the array. Actually, `chunk-size` bytes are written to each disk, serially.

If you specify a 4 kB chunk size, and write 16 kB to an array of three disks, the RAID system will write 4 kB to disks 0, 1 and 2, in parallel, then the remaining 4 kB to disk 0.

A 32 kB chunk−size is a reasonable starting point for most arrays. But the optimal value depends very much on the number of drives involved, the content of the file system you put on it, and many other factors. Experiment with it, to get the best performance.

RAID−0 with ext2

The following tip was contributed by michael@freenet−ag.de:

There is more disk activity at the beginning of ext2fs block groups. On a single disk, that does not matter, but it can hurt RAID0, if all block groups happen to begin on the same disk. Example:

With 4k stripe size and 4k block size, each block occupies one stripe. With two disks, the stripe−#disk−product is 2*4k=8k. The default block group size is 32768 blocks, so all block groups start on disk 0, which can easily become a hot spot, thus reducing overall performance. Unfortunately, the block group size can only be set in steps of 8 blocks (32k when using 4k blocks), so you can not avoid the problem by adjusting the block group size with the −g option of mkfs(8).

If you add a disk, the stripe−#disk−product is 12, so the first block group starts on disk 0, the second block group starts on disk 2 and the third on disk 1. The load caused by disk activity at the block group beginnings spreads over all disks.

In case you can not add a disk, try a stripe size of 32k. The stripe−#disk−product is 64k. Since you can change the block group size in steps of 8 blocks (32k), using a block group size of 32760 solves the problem.

Additionally, the block group boundaries should fall on stripe boundaries. That is no problem in the examples above, but it could easily happen with larger stripe sizes.

RAID−1

For writes, the chunk−size doesn't affect the array, since all data must be written to all disks no matter what. For reads however, the chunk−size specifies how much data to read serially from the participating disks. Since all active disks in the array contain the same information, the RAID layer has complete freedom in choosing from which disk information is read − this is used by the RAID code to improve average seek times by picking the disk best suited for any given read operation.

RAID−4

When a write is done on a RAID−4 array, the parity information must be updated on the parity disk as well.

The chunk−size affects read performance in the same way as in RAID−0, since reads from RAID−4 are done in the same way.

RAID−5

On RAID−5, the chunk size has the same meaning for reads as for RAID−0. Writing on RAID−5 is a little more complicated: When a chunk is written on a RAID−5 array, the corresponding parity chunk must be updated as well. Updating a parity chunk requires either

- The original chunk, the new chunk, and the old parity block
- Or, all chunks (except for the parity chunk) in the stripe

The RAID code will pick the easiest way to update each parity chunk as the write progresses. Naturally, if your server has lots of memory and/or if the writes are nice and linear, updating the parity chunks will only impose the overhead of one extra write going over the bus (just like RAID−1). The parity calculation itself is extremely efficient, so while it does of course load the main CPU of the system, this impact is negligible. If the writes are small and scattered all over the array, the RAID layer will almost always need to read in all the untouched chunks from each stripe that is written to, in order to calculate the parity chunk. This will impose extra bus−overhead and latency due to extra reads.

A reasonable chunk−size for RAID−5 is 128 kB, but as always, you may want to experiment with this.

Also see the section on special options for mke2fs. This affects RAID−5 performance.

5.11 Options for mke2fs

There is a special option available when formatting RAID−4 or −5 devices with mke2fs. The `-R stride=nn` option will allow mke2fs to better place different ext2 specific data−structures in an intelligent way on the RAID device.

If the chunk−size is 32 kB, it means, that 32 kB of consecutive data will reside on one disk. If we want to build an ext2 filesystem with 4 kB block−size, we realize that there will be eight filesystem blocks in one array chunk. We can pass this information on the mke2fs utility, when creating the filesystem:

```
mke2fs -b 4096 -R stride=8 /dev/md0
```

RAID−{4,5} performance is severely influenced by this option. I am unsure how the stride option will affect other RAID levels. If anyone has information on this, please send it in my direction.

The ext2fs blocksize *severely* influences the performance of the filesystem. You should always use 4kB block size on any filesystem larger than a few hundred megabytes, unless you store a very large number of very small files on it.

6. Detecting, querying and testing

This section is about life with a software RAID system, that's communicating with the arrays and tinkertoying them.

Note that when it comes to md devices manipulation, you should always remember that you are working with entire filesystems. So, although there could be some redundancy to keep your files alive, you must proceed with caution.

6.1 Detecting a drive failure

No mistery here. It's enough with a quick look to the standard log and stat files to notice a drive failure.

It's always a must for `/var/log/messages` to fill screens with tons of error messages, no matter what happened. But, when it's about a disk crash, huge lots of kernel errors are reported. Some nasty examples, for the masochists,

```
kernel: scsi0 channel 0 : resetting for second half of retries.
kernel: SCSI bus is being reset for host 0 channel 0.
kernel: scsi0: Sending Bus Device Reset CCB #2666 to Target 0
kernel: scsi0: Bus Device Reset CCB #2666 to Target 0 Completed
kernel: scsi : aborting command due to timeout : pid 2649, scsi0, channel 0, id 0, lun 0 Write (6) 18 33 11 24 00
kernel: scsi0: Aborting CCB #2669 to Target 0
kernel: SCSI host 0 channel 0 reset (pid 2644) timed out - trying harder
kernel: SCSI bus is being reset for host 0 channel 0.
kernel: scsi0: CCB #2669 to Target 0 Aborted
kernel: scsi0: Resetting BusLogic BT-958 due to Target 0
kernel: scsi0: *** BusLogic BT-958 Initialized Successfully ***
```

Most often, disk failures look like these,

```
kernel: sidisk I/O error: dev 08:01, sector 1590410
kernel: SCSI disk error : host 0 channel 0 id 0 lun 0 return code = 28000002
```

or these

```
kernel: hde: read_intr: error=0x10 { SectorIdNotFound }, CHS=31563/14/35, sector=0
kernel: hde: read_intr: status=0x59 { DriveReady SeekComplete DataRequest Error }
```

And, as expected, the classic `/proc/mdstat` look will also reveal problems,

```
Personalities : [linear] [raid0] [raid1] [translucent]
read_ahead not set
md7 : active raid1 sdc9[0] sdd5[8] 32000 blocks [2/1] [U_]
```

Later on this section we will learn how to monitor RAID with mdadm so we can receive alert reports about disk failures. Now it's time to learn more about `/proc/mdstat` interpretation.

6.2 Querying the arrays status

You can always take a look at `/proc/mdstat`. It won't hurt. Let's learn how to read the file. For example,

```
Personalities : [raid1]
read_ahead 1024 sectors
md5 : active raid1 sdb5[1] sda5[0]
      4200896 blocks [2/2] [UU]

md6 : active raid1 sdb6[1] sda6[0]
      2104384 blocks [2/2] [UU]

md7 : active raid1 sdb7[1] sda7[0]
      2104384 blocks [2/2] [UU]

md2 : active raid1 sdc7[1] sdd8[2] sde5[0]
      1052160 blocks [2/2] [UU]

unused devices: none
```

To identify the spare devices, first look for the [#/#] value on a line. The first number is the number of a complete raid device as defined. Lets say it is "n". The raid role numbers [#] following each device indicate its role, or function, within the raid set. Any device with "n" or higher are spare disks. 0,1,..,n−1 are for the working array.

Also, if you have a failure, the failed device will be marked with (F) after the [#]. The spare that replaces this device will be the device with the lowest role number n or higher that is not marked (F). Once the resync operation is complete, the device's role numbers are swapped.

The order in which the devices appear in the `/proc/mdstat` output means nothing.

Finally, remember that you can always use raidtools or mdadm to check the arrays out.

```
mdadm --detail /dev/mdx
lsraid -a /dev/mdx
```

These commands will show spare and failed disks loud and clear.

6.3 Simulating a drive failure

If you plan to use RAID to get fault−tolerance, you may also want to test your setup, to see if it really works. Now, how does one simulate a disk failure?

The short story is, that you can't, except perhaps for putting a fire axe thru the drive you want to "simulate" the fault on. You can never know what will happen if a drive dies. It may electrically take the bus it is attached to with it, rendering all drives on that bus inaccessible. I have never heard of that happening though, but it is entirely possible. The drive may also just report a read/write fault to the SCSI/IDE layer, which in turn makes the RAID layer handle this situation gracefully. This is fortunately the way things often go.

Remember, that you must be running RAID−{1,4,5} for your array to be able to survive a disk failure. Linear− or RAID−0 will fail completely when a device is missing.

Force−fail by hardware

If you want to simulate a drive failure, you can just plug out the drive. You should do this with the power off. If you are interested in testing whether your data can survive with a disk less than the usual number, there is no point in being a hot−plug cowboy here. Take the system down, unplug the disk, and boot it up again.

Look in the syslog, and look at `/proc/mdstat` to see how the RAID is doing. Did it work?

Faulty disks should appear marked with an `(F)` if you look at `/proc/mdstat`. Also, users of mdadm should see the device state as `faulty`.

When you've re−connected the disk again (with the power off, of course, remember), you can add the "new" device to the RAID again, with the raidhotadd command.

Force−fail by software

Newer versions of raidtools come with a `raidsetfaulty` command. By using `raidsetfaulty` you can just simulate a drive failure without unplugging things off.

Just running the command

```
raidsetfaulty /dev/md1 /dev/sdc2
```

should be enough to fail the disk /dev/sdc2 of the array /dev/md1. If you are using mdadm, just type

```
mdadm --manage --set-faulty /dev/md1 /dev/sdc2
```

Now things move up and fun appears. First, you should see something like the first line of this on your system's log. Something like the second line will appear if you have spare disks configured.

```
kernel: raid1: Disk failure on sdc2, disabling device.
kernel: md1: resyncing spare disk sdb7 to replace failed disk
```

Checking `/proc/mdstat` out will show the degraded array. If there was a spare disk available, reconstruction should have started.

Another fresh utility in newest raidtools is `lsraid`. Try with

```
lsraid -a /dev/md1
```

users of mdadm can run the command

```
mdadm --detail /dev/md1
```

and enjoy the view.

Now you've seen how it goes when a device fails. Let's fix things up.

First, we will remove the failed disk from the array. Run the command

```
raidhotremove /dev/md1 /dev/sdc2
```

users of mdadm can run the command

```
mdadm /dev/md1 -r /dev/sdc2
```

Note that `raidhotremove` cannot pull a disk out of a running array. For obvious reasons, only crashed disks are to be hotremoved from an array (running raidstop and unmounting the device won't help).

Now we have a /dev/md1 which has just lost a device. This could be a degraded RAID or perhaps a system in the middle of a reconstruction process. We wait until recovery ends before setting things back to normal.

So the trip ends when we send /dev/sdc2 back home.

```
raidhotadd /dev/md1 /dev/sdc2
```

As usual, you can use mdadm instead of raidtools. This should be the command

```
mdadm /dev/md1 -a /dev/sdc2
```

As the prodigal son returns to the array, we'll see it becoming an active member of /dev/md1 if necessary. If not, it will be marked as an spare disk. That's management made easy.

6.4 Simulating data corruption

RAID (be it hardware− or software−), assumes that if a write to a disk doesn't return an error, then the write was successful. Therefore, if your disk corrupts data without returning an error, your data *will* become corrupted. This is of course very unlikely to happen, but it is possible, and it would result in a corrupt filesystem.

RAID cannot and is not supposed to guard against data corruption on the media. Therefore, it doesn't make any sense either, to purposely corrupt data (using `dd` for example) on a disk to see how the RAID system will handle that. It is most likely (unless you corrupt the RAID superblock) that the RAID layer will never find out about the corruption, but your filesystem on the RAID device will be corrupted.

This is the way things are supposed to work. RAID is not a guarantee for data integrity, it just allows you to keep your data if a disk dies (that is, with RAID levels above or equal one, of course).

6.5 Monitoring RAID arrays

You can run mdadm as a daemon by using the follow−monitor mode. If needed, that will make mdadm send email alerts to the system administrator when arrays encounter errors or fail. Also, follow mode can be used to trigger contingency commands if a disk fails, like giving a second chance to a failed disk by removing and reinserting it, so a non−fatal failure could be automatically solved.

Let's see a basic example. Running

```
mdadm --monitor --mail=root@localhost --delay=1800 /dev/md2
```

should release a mdadm daemon to monitor /dev/md2. The delay parameter means that polling will be done in intervals of 1800 seconds. Finally, critical events and fatal errors should be e−mailed to the system manager. That's RAID monitoring made easy.

Finally, the `--program` or `--alert` parameters specify the program to be run whenever an event is detected.

Note that the mdadm daemon will never exit once it decides that there are arrays to monitor, so it should normally be run in the background. Remember that your are running a daemon, not a shell command.

Using mdadm to monitor a RAID array is simple and effective. However, there are fundamental problems with that kind of monitoring − what happens, for example, if the mdadm daemon stops? In order to overcome this problem, one should look towards "real" monitoring solutions. There is a number of free software, open source, and commercial solutions available which can be used for Software RAID monitoring on Linux. A search on FreshMeat should return a good number of matches.

7. Tweaking, tuning and troubleshooting

7.1 `raid-level` and `raidtab`

Some GNU/Linux distributions, like RedHat 8.0 and possibly others, have a bug in their init−scripts, so that they will fail to start up RAID arrays on boot, if your `/etc/raidtab` has spaces or tabs before the `raid-level` keywords.

The simple workaround for this problem is to make sure that the `raid-level` keyword appears in the very beginning of the lines, without any leading spaces of any kind.

7.2 Autodetection

Autodetection allows the RAID devices to be automatically recognized by the kernel at boot−time, right after the ordinary partition detection is done.

This requires several things:

1. You need autodetection support in the kernel. Check this
2. You must have created the RAID devices using persistent−superblock
3. The partition−types of the devices used in the RAID must be set to **0xFD** (use fdisk and set the type to "fd")

NOTE: Be sure that your RAID is NOT RUNNING before changing the partition types. Use `raidstop /dev/md0` to stop the device.

If you set up 1, 2 and 3 from above, autodetection should be set up. Try rebooting. When the system comes up, cat'ing `/proc/mdstat` should tell you that your RAID is running.

During boot, you could see messages similar to these:

```
Oct 22 00:51:59 malthe kernel: SCSI device sdg: hdwr sector= 512
 bytes. Sectors= 12657717 [6180 MB] [6.2 GB]
Oct 22 00:51:59 malthe kernel: Partition check:
Oct 22 00:51:59 malthe kernel:  sda: sda1 sda2 sda3 sda4
Oct 22 00:51:59 malthe kernel:  sdb: sdb1 sdb2
Oct 22 00:51:59 malthe kernel:  sdc: sdc1 sdc2
Oct 22 00:51:59 malthe kernel:  sdd: sdd1 sdd2
Oct 22 00:51:59 malthe kernel:  sde: sde1 sde2
```

```
 Oct 22 00:51:59 malthe kernel:  sdf: sdf1 sdf2
 Oct 22 00:51:59 malthe kernel:  sdg: sdg1 sdg2
 Oct 22 00:51:59 malthe kernel: autodetecting RAID arrays
 Oct 22 00:51:59 malthe kernel: (read) sdb1's sb offset: 6199872
 Oct 22 00:51:59 malthe kernel: bind<sdb1,1>
 Oct 22 00:51:59 malthe kernel: (read) sdc1's sb offset: 6199872
 Oct 22 00:51:59 malthe kernel: bind<sdc1,2>
 Oct 22 00:51:59 malthe kernel: (read) sdd1's sb offset: 6199872
 Oct 22 00:51:59 malthe kernel: bind<sdd1,3>
 Oct 22 00:51:59 malthe kernel: (read) sde1's sb offset: 6199872
 Oct 22 00:51:59 malthe kernel: bind<sde1,4>
 Oct 22 00:51:59 malthe kernel: (read) sdf1's sb offset: 6205376
 Oct 22 00:51:59 malthe kernel: bind<sdf1,5>
 Oct 22 00:51:59 malthe kernel: (read) sdg1's sb offset: 6205376
 Oct 22 00:51:59 malthe kernel: bind<sdg1,6>
 Oct 22 00:51:59 malthe kernel: autorunning md0
 Oct 22 00:51:59 malthe kernel: running: <sdg1><sdf1><sde1><sdd1><sdc1><sdb1>
 Oct 22 00:51:59 malthe kernel: now!
 Oct 22 00:51:59 malthe kernel: md: md0: raid array is not clean --
  starting background reconstruction
```

This is output from the autodetection of a RAID−5 array that was not cleanly shut down (eg. the machine crashed). Reconstruction is automatically initiated. Mounting this device is perfectly safe, since reconstruction is transparent and all data are consistent (it's only the parity information that is inconsistent − but that isn't needed until a device fails).

Autostarted devices are also automatically stopped at shutdown. Don't worry about init scripts. Just use the /dev/md devices as any other /dev/sd or /dev/hd devices.

Yes, it really is that easy.

You may want to look in your init−scripts for any raidstart/raidstop commands. These are often found in the standard RedHat init scripts. They are used for old−style RAID, and has no use in new−style RAID with autodetection. Just remove the lines, and everything will be just fine.

7.3 Booting on RAID

There are several ways to set up a system that mounts it's root filesystem on a RAID device. Some distributions allow for RAID setup in the installation process, and this is by far the easiest way to get a nicely set up RAID system.

Newer LILO distributions can handle RAID−1 devices, and thus the kernel can be loaded at boot−time from a RAID device. LILO will correctly write boot−records on all disks in the array, to allow booting even if the primary disk fails.

If you are using grub instead of LILO, then just start grub and configure it to use the second (or third, or fourth...) disk in the RAID−1 array you want to boot off as its root device and run setup. And that's all.

For example, on an array consisting of /dev/hda1 and /dev/hdc1 where both partitions should be bootable you should just do this:

```
grub
grub>device (hd0) /dev/hdc
grub>root (hd0,0)
grub>setup (hd0)
```

Some users have experienced problems with this, reporting that although booting with one drive connected worked, booting with *both* two drives failed. Nevertheless, running the described procedure with both disks fixed the problem, allowing the system to boot from either single drive or from the RAID−1

Another way of ensuring that your system can always boot is, to create a boot floppy when all the setup is done. If the disk on which the `/boot` filesystem resides dies, you can always boot from the floppy. On RedHat and RedHat derived systems, this can be accomplished with the `mkbootdisk` command.

7.4 Root filesystem on RAID

In order to have a system booting on RAID, the root filesystem (/) must be mounted on a RAID device. Two methods for achieving this is supplied bellow. The methods below assume that you install on a normal partition, and then − when the installation is complete − move the contents of your non−RAID root filesystem onto a new RAID device. Please not that this is no longer needed in general, as most newer GNU/Linux distributions support installation on RAID devices (and creation of the RAID devices during the installation process). However, you may still want to use the methods below, if you are migrating an existing system to RAID.

Method 1

This method assumes you have a spare disk you can install the system on, which is not part of the RAID you will be configuring.

- First, install a normal system on your extra disk.
- Get the kernel you plan on running, get the raid−patches and the tools, and make your system boot with this new RAID−aware kernel. Make sure that RAID−support is **in** the kernel, and is not loaded as modules.
- Ok, now you should configure and create the RAID you plan to use for the root filesystem. This is standard procedure, as described elsewhere in this document.
- Just to make sure everything's fine, try rebooting the system to see if the new RAID comes up on boot. It should.
- Put a filesystem on the new array (using `mke2fs`), and mount it under /mnt/newroot
- Now, copy the contents of your current root−filesystem (the spare disk) to the new root−filesystem (the array). There are lots of ways to do this, one of them is

  ```
  cd /
  find . -xdev | cpio -pm /mnt/newroot
  ```

  another way to copy everything from / to /mnt/newroot could be

  ```
   cp -ax / /mnt/newroot
  ```
- You should modify the `/mnt/newroot/etc/fstab` file to use the correct device (the `/dev/md?` root device) for the root filesystem.
- Now, unmount the current `/boot` filesystem, and mount the boot device on `/mnt/newroot/boot` instead. This is required for LILO to run successfully in the next step.
- Update `/mnt/newroot/etc/lilo.conf` to point to the right devices. The boot device must still be a regular disk (non−RAID device), but the root device should point to your new RAID. When done, run

  ```
  lilo -r /mnt/newroot
  ```

This LILO run should complete with no errors.
- Reboot the system, and watch everything come up as expected :)

If you're doing this with IDE disks, be sure to tell your BIOS that all disks are "auto−detect" types, so that the BIOS will allow your machine to boot even when a disk is missing.

Method 2

This method requires that your kernel and raidtools understand the `failed−disk` directive in the `/etc/raidtab` file − if you are working on a really old system this may not be the case, and you will need to upgrade your tools and/or kernel first.

You can **only** use this method on RAID levels 1 and above, as the method uses an array in "degraded mode" which in turn is only possible if the RAID level has redundancy. The idea is to install a system on a disk which is purposely marked as failed in the RAID, then copy the system to the RAID which will be running in degraded mode, and finally making the RAID use the no−longer needed "install−disk", zapping the old installation but making the RAID run in non−degraded mode.

- First, install a normal system on one disk (that will later become part of your RAID). It is important that this disk (or partition) is not the smallest one. If it is, it will not be possible to add it to the RAID later on!
- Then, get the kernel, the patches, the tools etc. etc. You know the drill. Make your system boot with a new kernel that has the RAID support you need, compiled into the kernel.
- Now, set up the RAID with your current root−device as the `failed−disk` in the `/etc/raidtab` file. Don't put the `failed−disk` as the first disk in the `raidtab`, that will give you problems with starting the RAID. Create the RAID, and put a filesystem on it. If using mdadm, you can create a degraded array just by running something like `mdadm −C /dev/md0 −−level raid1 −−raid−disks 2 missing /dev/hdc1`, note the `missing` parameter.
- Try rebooting and see if the RAID comes up as it should
- Copy the system files, and reconfigure the system to use the RAID as root−device, as described in the previous section.
- When your system successfully boots from the RAID, you can modify the `/etc/raidtab` file to include the previously `failed−disk` as a normal `raid−disk`. Now, `raidhotadd` the disk to your RAID.
- You should now have a system that can boot from a non−degraded RAID.

7.5 Making the system boot on RAID

For the kernel to be able to mount the root filesystem, all support for the device on which the root filesystem resides, must be present in the kernel. Therefore, in order to mount the root filesystem on a RAID device, the kernel *must* have RAID support.

The normal way of ensuring that the kernel can see the RAID device is to simply compile a kernel with all necessary RAID support compiled in. Make sure that you compile the RAID support *into* the kernel, and *not* as loadable modules. The kernel cannot load a module (from the root filesystem) before the root filesystem is mounted.

However, since RedHat−6.0 ships with a kernel that has new−style RAID support as modules, I here describe how one can use the standard RedHat−6.0 kernel and still have the system boot on RAID.

Booting with RAID as module

You will have to instruct LILO to use a RAM−disk in order to achieve this. Use the `mkinitrd` command to create a ramdisk containing all kernel modules needed to mount the root partition. This can be done as:

```
mkinitrd --with=<module> <ramdisk name> <kernel>
```

For example:

```
mkinitrd --preload raid5 --with=raid5 raid-ramdisk 2.2.5-22
```

This will ensure that the specified RAID module is present at boot−time, for the kernel to use when mounting the root device.

Modular RAID on Debian GNU/Linux after move to RAID

Debian users may encounter problems using an initrd to mount their root filesystem from RAID, if they have migrated a standard non−RAID Debian install to root on RAID.

If your system fails to mount the root filesystem on boot (you will see this in a "kernel panic" message), then the problem may be that the initrd filesystem does not have the necessary support to mount the root filesystem from RAID.

Debian seems to produce its `initrd.img` files on the assumption that the root filesystem to be mounted is the current one. This will usually result in a kernel panic if the root filesystem is moved to the raid device and you attempt to boot from that device using the same initrd image. The solution is to use the `mkinitrd` command but specifying the proposed new root filesystem. For example, the following commands should create and set up the new initrd on a Debian system:

```
% mkinitrd -r /dev/md0 -o /boot/initrd.img-2.4.22raid
% mv /initrd.img /initrd.img-nonraid
% ln -s /boot/initrd.img-raid /initrd.img"
```

7.6 Converting a non−RAID RedHat System to run on Software RAID

This section was written and contributed by Mark Price, IBM. The text has undergone minor changes since his original work.

Notice: the following information is provided "AS IS" with no representation or warranty of any kind either express or implied. You may use it freely at your own risk, and no one else will be liable for any damages arising out of such usage.

Introduction

The technote details how to convert a linux system with non RAID devices to run with a Software RAID configuration.

Scope

This scenario was tested with Redhat 7.1, but should be applicable to any release which supports Software RAID (md) devices.

Pre−conversion example system

The test system contains two SCSI disks, sda and sdb both of of which are the same physical size. As part of the test setup, I configured both disks to have the same partition layout, using fdisk to ensure the number of blocks for each partition was identical.

```
DEVICE      MOUNTPOINT  SIZE        DEVICE      MOUNTPOINT  SIZE
/dev/sda1   /           2048MB      /dev/sdb1               2048MB
/dev/sda2   /boot       80MB        /dev/sdb2               80MB
/dev/sda3   /var/       100MB       /dev/sdb3               100MB
/dev/sda4   SWAP        1024MB      /dev/sdb4   SWAP        1024MB
```

In our basic example, we are going to set up a simple RAID−1 Mirror, which requires only two physical disks.

Step−1 − boot rescue cd/floppy

The redhat installation CD provides a rescue mode which boots into linux from the CD and mounts any filesystems it can find on your disks.

At the lilo prompt type

```
    lilo: linux rescue
```

With the setup described above, the installer may ask you which disk your root filesystem in on, either sda or sdb. Select sda.

The installer will mount your filesytems in the following way.

```
DEVICE      MOUNTPOINT  TEMPORARY MOUNT POINT
/dev/sda1   /           /mnt/sysimage
/dev/sda2   /boot       /mnt/sysimage/boot
/dev/sda3   /var        /mnt/sysimage/var
/dev/sda6   /home       /mnt/sysimage/home
```

Note: − Please bear in mind other distributions may mount your filesystems on different mount points, or may require you to mount them by hand.

Step−2 − create a `/etc/raidtab` file

Create the file /mnt/sysimage/etc/raidtab (or wherever your real /etc file system has been mounted.

For our test system, the raidtab file would like like this.

```
raiddev /dev/md0
    raid-level              1
    nr-raid-disks           2
```

```
    nr-spare-disks          0
    chunk-size              4
    persistent-superblock   1
    device                  /dev/sda1
    raid-disk               0
    device                  /dev/sdb1
    raid-disk               1

raiddev /dev/md1
    raid-level              1
    nr-raid-disks           2
    nr-spare-disks          0
    chunk-size              4
    persistent-superblock   1
    device                  /dev/sda2
    raid-disk               0
    device                  /dev/sdb2
    raid-disk               1

raiddev /dev/md2
    raid-level              1
    nr-raid-disks           2
    nr-spare-disks          0
    chunk-size              4
    persistent-superblock   1
    device                  /dev/sda3
    raid-disk               0
    device                  /dev/sdb3
    raid-disk               1
```

Note: − It is important that the devices are in the correct order. ie. that `/dev/sda1` is `raid-disk 0` and not `raid-disk 1`. This instructs the md driver to sync from `/dev/sda1`, if it were the other way around it would sync from `/dev/sdb1` which would destroy your filesystem.

Now copy the raidtab file from your real root filesystem to the current root filesystem.

```
(rescue)# cp /mnt/sysimage/etc/raidtab /etc/raidtab
```

Step−3 − create the md devices

There are two ways to do this, copy the device files from /mnt/sysimage/dev or use mknod to create them. The md device, is a (b)lock device with major number 9.

```
(rescue)# mknod /dev/md0 b 9 0
(rescue)# mknod /dev/md1 b 9 1
(rescue)# mknod /dev/md2 b 9 2
```

Step−4 − unmount filesystems

In order to start the raid devices, and sync the drives, it is necessary to unmount all the temporary filesystems.

```
(rescue)# umount /mnt/sysimage/var
(rescue)# umount /mnt/sysimage/boot
(rescue)# umount /mnt/sysimage/proc
(rescue)# umount /mnt/sysimage
```

Please note, you may not be able to umount `/mnt/sysimage`. This problem can be caused by the rescue system − if you choose to manually mount your filesystems instead of letting the rescue system do this automatically, this problem should go away.

Step−5 − start raid devices

Because there are filesystems on /dev/sda1, /dev/sda2 and /dev/sda3 it is necessary to force the start of the raid device.

```
(rescue)# mkraid --really-force /dev/md2
```

You can check the completion progress by cat'ing the `/proc/mdstat` file. It shows you status of the raid device and percentage left to sync.

Continue with /boot and /

```
(rescue)# mkraid --really-force /dev/md1
(rescue)# mkraid --really-force /dev/md0
```

The md driver syncs one device at a time.

Step−6 − remount filesystems

Mount the newly synced filesystems back into the /mnt/sysimage mount points.

```
(rescue)# mount /dev/md0 /mnt/sysimage
(rescue)# mount /dev/md1 /mnt/sysimage/boot
(rescue)# mount /dev/md2 /mnt/sysimage/var
```

Step−7 − change root

You now need to change your current root directory to your real root file system.

```
(rescue)# chroot /mnt/sysimage
```

Step−8 − edit config files

You need to configure lilo and `/etc/fstab` appropriately to boot from and mount the md devices.

Note: − The boot device MUST be a non−raided device. The root device is your new md0 device. eg.

```
boot=/dev/sda
map=/boot/map
install=/boot/boot.b
prompt
timeout=50
message=/boot/message
linear
default=linux

image=/boot/vmlinuz
    label=linux
    read-only
```

```
    root=/dev/md0
```

Alter `/etc/fstab`

```
/dev/md0               /                        ext3    defaults        1 1
/dev/md1               /boot                    ext3    defaults        1 2
/dev/md2               /var                     ext3    defaults        1 2
/dev/sda4              swap                     swap    defaults        0 0
```

Step−9 − run LILO

With the `/etc/lilo.conf` edited to reflect the new `root=/dev/md0` and with `/dev/md1` mounted as `/boot`, we can now run `/sbin/lilo −v` on the chrooted filesystem.

Step−10 − change partition types

The partition types of the all the partitions on ALL Drives which are used by the md driver must be changed to type 0xFD.

Use fdisk to change the partition type, using option 't'.

```
(rescue)# fdisk /dev/sda
(rescue)# fdisk /dev/sdb
```

Use the 'w' option after changing all the required partitions to save the partion table to disk.

Step−11 − resize filesystem

When we created the raid device, the physical partion became slightly smaller because a second superblock is stored at the end of the partition. If you reboot the system now, the reboot will fail with an error indicating the superblock is corrupt.

Resize them prior to the reboot, ensure that the all md based filesystems are unmounted except root, and remount root read−only.

```
(rescue)# mount / -o remount,ro
```

You will be required to fsck each of the md devices. This is the reason for remounting root read−only. The −f flag is required to force fsck to check a clean filesystem.

```
(rescue)# e2fsck -f /dev/md0
```

This will generate the same error about inconsistent sizes and possibly corrupted superblock.Say N to 'Abort?'.

```
(rescue)# resize2fs /dev/md0
```

Repeat for all `/dev/md` devices.

Step−12 − checklist

The next step is to reboot the system, prior to doing this run through the checklist below and ensure all tasks have been completed.

- All devices have finished syncing. Check `/proc/mdstat`
- `/etc/fstab` has been edited to reflect the changes to the device names.
- `/etc/lilo.conf` has beeb edited to reflect root device change.
- `/sbin/lilo` has been run to update the boot loader.
- The kernel has both SCSI and RAID(MD) drivers built into the kernel.
- The partition types of all partitions on disks that are part of an md device have been changed to 0xfd.
- The filesystems have been fsck'd and resize2fs'd.

Step−13 − reboot

You can now safely reboot the system, when the system comes up it will auto discover the md devices (based on the partition types).

Your root filesystem will now be mirrored.

7.7 Sharing spare disks between different arrays

When running mdadm in the follow/monitor mode you can make different arrays share spare disks. That will surely make you save storage space without losing the comfort of fallback disks.

In the world of software RAID, this is a brand new never−seen−before feature: for securing things to the point of spare disk areas, you just have to provide one single idle disk for a bunch of arrays.

With mdadm is running as a daemon, you have an agent polling arrays at regular intervals. Then, as a disk fails on an array without a spare disk, mdadm removes an available spare disk from another array and inserts it into the array with the failed disk. The reconstruction process begins now in the degraded array as usual.

To declare shared spare disks, just use the `spare−group` parameter when invoking mdadm as a daemon.

7.8 Pitfalls

Never NEVER **never** re−partition disks that are part of a running RAID. If you must alter the partition table on a disk which is a part of a RAID, stop the array first, then repartition.

It is easy to put too many disks on a bus. A normal Fast−Wide SCSI bus can sustain 10 MB/s which is less than many disks can do alone today. Putting six such disks on the bus will of course not give you the expected performance boost. It is becoming equally easy to saturate the PCI bus − remember, a normal 32−bit 33 MHz PCI bus has a theoretical maximum bandwidth of around 133 MB/sec, considering command overhead etc. you will see a somewhat lower real−world transfer rate. Some disks today has a throughput in excess of 30 MB/sec, so just four of those disks will actually max out your PCI bus! When designing high−performance RAID systems, be sure to take the whole I/O path into consideration − there are boards with more PCI busses, with 64−bit and 66 MHz busses, and with PCI−X.

More SCSI controllers will only give you extra performance, if the SCSI busses are nearly maxed out by the disks on them. You will not see a performance improvement from using two 2940s with two old SCSI disks, instead of just running the two disks on one controller.

If you forget the persistent−superblock option, your array may not start up willingly after it has been stopped. Just re−create the array with the option set correctly in the raidtab. Please note that this will destroy the information on the array!

If a RAID−5 fails to reconstruct after a disk was removed and re−inserted, this may be because of the ordering of the devices in the raidtab. Try moving the first "device ..." and "raid−disk ..." pair to the bottom of the array description in the raidtab file.

8. Reconstruction

If you have read the rest of this HOWTO, you should already have a pretty good idea about what reconstruction of a degraded RAID involves. Let us summarize:

- Power down the system
- Replace the failed disk
- Power up the system once again.
- Use `raidhotadd /dev/mdX /dev/sdX` to re−insert the disk in the array
- Have coffee while you watch the automatic reconstruction running

And that's it.

Well, it usually is, unless you're unlucky and your RAID has been rendered unusable because more disks than the ones redundant failed. This can actually happen if a number of disks reside on the same bus, and one disk takes the bus with it as it crashes. The other disks, however fine, will be unreachable to the RAID layer, because the bus is down, and they will be marked as faulty. On a RAID−5 where you can spare one disk only, loosing two or more disks can be fatal.

The following section is the explanation that Martin Bene gave to me, and describes a possible recovery from the scary scenario outlined above. It involves using the `failed−disk` directive in your `/etc/raidtab` (so for people running patched 2.2 kernels, this will only work on kernels 2.2.10 and later).

8.1 Recovery from a multiple disk failure

The scenario is:

- A controller dies and takes two disks offline at the same time,
- All disks on one scsi bus can no longer be reached if a disk dies,
- A cable comes loose...

In short: quite often you get a *temporary* failure of several disks at once; afterwards the RAID superblocks are out of sync and you can no longer init your RAID array.

If using mdadm, you could first try to run:

```
mdadm --assemble --force
```

If not, there's one thing left: rewrite the RAID superblocks by `mkraid --force`

To get this to work, you'll need to have an up to date `/etc/raidtab` − if it doesn't **EXACTLY** match devices and ordering of the original disks this will not work as expected, but **will most likely completely**

obliterate whatever data you used to have on your disks.

Look at the sylog produced by trying to start the array, you'll see the event count for each superblock; usually it's best to leave out the disk with the lowest event count, i.e the oldest one.

If you `mkraid` without `failed−disk`, the recovery thread will kick in immediately and start rebuilding the parity blocks − not necessarily what you want at that moment.

With `failed−disk` you can specify exactly which disks you want to be active and perhaps try different combinations for best results. BTW, only mount the filesystem read−only while trying this out... This has been successfully used by at least two guys I've been in contact with.

9. Performance

This section contains a number of benchmarks from a real−world system using software RAID. There is some general information about benchmarking software too.

Benchmark samples were done with the `bonnie` program, and at all times on files twice− or more the size of the physical RAM in the machine.

The benchmarks here *only* measures input and output bandwidth on one large single file. This is a nice thing to know, if it's maximum I/O throughput for large reads/writes one is interested in. However, such numbers tell us little about what the performance would be if the array was used for a news spool, a web−server, etc. etc. Always keep in mind, that benchmarks numbers are the result of running a "synthetic" program. Few real−world programs do what `bonnie` does, and although these I/O numbers are nice to look at, they are not ultimate real−world−appliance performance indicators. Not even close.

For now, I only have results from my own machine. The setup is:

- Dual Pentium Pro 150 MHz
- 256 MB RAM (60 MHz EDO)
- Three IBM UltraStar 9ES 4.5 GB, SCSI U2W
- Adaptec 2940U2W
- One IBM UltraStar 9ES 4.5 GB, SCSI UW
- Adaptec 2940 UW
- Kernel 2.2.7 with RAID patches

The three U2W disks hang off the U2W controller, and the UW disk off the UW controller.

It seems to be impossible to push much more than 30 MB/s thru the SCSI busses on this system, using RAID or not. My guess is, that because the system is fairly old, the memory bandwidth sucks, and thus limits what can be sent thru the SCSI controllers.

9.1 RAID−0

Read is **Sequential block input**, and **Write** is **Sequential block output**. File size was 1GB in all tests. The tests where done in single−user mode. The SCSI driver was configured not to use tagged command queuing.

	Block size	Read kB/s	Write kB/s

Chunk size			
4k	1k	19712	18035
4k	4k	34048	27061
8k	1k	19301	18091
8k	4k	33920	27118
16k	1k	19330	18179
16k	2k	28161	23682
16k	4k	33990	27229
32k	1k	19251	18194
32k	4k	34071	26976

>From this it seems that the RAID chunk−size doesn't make that much of a difference. However, the ext2fs block−size should be as large as possible, which is 4kB (eg. the page size) on IA−32.

9.2 RAID−0 with TCQ

This time, the SCSI driver was configured to use tagged command queuing, with a queue depth of 8. Otherwise, everything's the same as before.

Chunk size	Block size	Read kB/s	Write kB/s
32k	4k	33617	27215

No more tests where done. TCQ seemed to slightly increase write performance, but there really wasn't much of a difference at all.

9.3 RAID−5

The array was configured to run in RAID−5 mode, and similar tests where done.

Chunk size	Block size	Read kB/s	Write kB/s
8k	1k	11090	6874
8k	4k	13474	12229
32k	1k	11442	8291
32k	2k	16089	10926
32k	4k	18724	12627

Now, both the chunk−size and the block−size seems to actually make a difference.

9.4 RAID−10

RAID−10 is "mirrored stripes", or, a RAID−1 array of two RAID−0 arrays. The chunk−size is the chunk sizes of both the RAID−1 array and the two RAID−0 arrays. I did not do test where those chunk−sizes differ, although that should be a perfectly valid setup.

Chunk size	Block size	Read kB/s	Write kB/s
32k	1k	13753	11580
32k	4k	23432	22249

No more tests where done. The file size was 900MB, because the four partitions involved where 500 MB each, which doesn't give room for a 1G file in this setup (RAID−1 on two 1000MB arrays).

9.5 Fresh benchmarking tools

To check out speed and performance of your RAID systems, do NOT use hdparm. It won't do real benchmarking of the arrays.

Instead of hdparm, take a look at the tools described here: IOzone and Bonnie++.

IOzone is a small, versatile and modern tool to use. It benchmarks file I/O performance for `read, write, re−read, re−write, read backwards, read strided, fread, fwrite, random read, pread, mmap, aio_read` and `aio_write` operations. Don't worry, it can run on any of the ext2, ext3, reiserfs, JFS, or XFS filesystems in OSDL STP.

You can also use IOzone to show throughput performance as a function of number of processes and number of disks used in a filesystem, something interesting when it's about RAID striping.

Although documentation for IOzone is available in Acrobat/PDF, PostScript, nroff, and MS Word formats, we are going to cover here a nice example of IOzone in action:

```
iozone −s 4096
```

This would run a test using a 4096KB file size.

And this is an example of the output quality IOzone gives

```
        File size set to 4096 KB
        Output is in Kbytes/sec
        Time Resolution = 0.000001 seconds.
        Processor cache size set to 1024 Kbytes.
        Processor cache line size set to 32 bytes.
        File stride size set to 17 * record size.
                                                            random  random    bkwd  record  stride
              KB  reclen   write rewrite    read    reread    read   write    read rewrite    read   fwrite frewrite   fread  freread
            4096       4   99028  194722   285873   298063  265560  170737  398600  436346  380952    91651   127212  288309   292633
```

Now you just need to know about the feature that makes IOzone useful for RAID benchmarking: the file operations involving RAID are the `read strided`. The example above shows a 380.952Kb/sec. for the `read strided`, so you can go figure.

Bonnie++ seems to be more targeted at benchmarking single drives that at RAID, but it can test more than 2Gb of storage on 32−bit machines, and tests for file `creat, stat, unlink` operations.

10. Related tools

While not described in this HOWTO, some useful tools for Software−RAID systems have been developed.

10.1 RAID resizing and conversion

It is not easy to add another disk to an existing array. A tool to allow for just this operation has been developed, and is available from http://unthought.net/raidreconf. The tool will allow for conversion between RAID levels, for example converting a two−disk RAID−1 array into a four−disk RAID−5 array. It will also allow for chunk−size conversion, and simple disk adding.

Please note that this tool is not really "production ready". It seems to have worked well so far, but it is a rather time−consuming process that, if it fails, will absolutely guarantee that your data will be irrecoverably scattered over your disks. **You absolutely *must* keep good backups prior to experimenting with this tool**.

10.2 Backup

Remember, RAID is no substitute for good backups. No amount of redundancy in your RAID configuration is going to let you recover week or month old data, nor will a RAID survive fires, earthquakes, or other disasters.

It is imperative that you protect your data, not just with RAID, but with *regular* good backups. One excellent system for such backups, is the Amanda backup system.

---

11. Partitioning RAID / LVM on RAID

RAID devices cannot be partitioned, like ordinary disks can. This can be a real annoyance on systems where one wants to run, for example, two disks in a RAID−1, but divide the system onto multiple different filesystems. A horror example could look like:

```
# df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/md2              3.8G  640M  3.0G  18% /
/dev/md1               97M   11M   81M  12% /boot
/dev/md5              3.8G  1.1G  2.5G  30% /usr
/dev/md6              9.6G  8.5G  722M  93% /var/www
/dev/md7              3.8G  951M  2.7G  26% /var/lib
/dev/md8              3.8G   38M  3.6G   1% /var/spool
/dev/md9              1.9G  231M  1.5G  13% /tmp
/dev/md10             8.7G  329M  7.9G   4% /var/www/html
```

11.1 Partitioning RAID devices

If a RAID device could be partitioned, the administrator could simply have created one single `/dev/md0 device` device, partitioned it as he usually would, and put the filesystems there. Instead, with today's Software RAID, he must create a RAID−1 device for every single filesystem, even though there are only two disks in the system.

There have been various patches to the kernel which would allow partitioning of RAID devices, but none of them have (as of this writing) made it into the kernel. In short; it is not currently possible to partition a RAID device − but luckily there *is* another solution to this problem.

11.2 LVM on RAID

The solution to the partitioning problem is LVM, Logical Volume Management. LVM has been in the stable Linux kernel series for a long time now − LVM2 in the 2.6 kernel series is a further improvement over the older LVM support from the 2.4 kernel series. While LVM has traditionally scared some people away because of its complexity, it really is something that an administrator could and should consider if he wishes to use more than a few filesystems on a server.

We will not attempt to describe LVM setup in this HOWTO, as there already is a fine HOWTO for exactly this purpose. A small example of a RAID + LVM setup will be presented though. Consider the `df` output below, of such a system:

```
# df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/md0              942M  419M  475M  47% /
/dev/vg0/backup        40G  1.3M   39G   1% /backup
/dev/vg0/amdata       496M  237M  233M  51% /var/lib/amanda
/dev/vg0/mirror        62G   56G  2.9G  96% /mnt/mirror
/dev/vg0/webroot       97M  6.5M   85M   8% /var/www
/dev/vg0/local        2.0G  458M  1.4G  24% /usr/local
/dev/vg0/netswap      3.0G  2.1G 1019M  67% /mnt/netswap
```

"What's the difference" you might ask... Well, this system has only two RAID−1 devices − one for the root filesystem, and one that cannot be seen on the `df` output − this is because `/dev/md1` is used as a "physical volume" for LVM. What this means is, that `/dev/md1` acts as "backing store" for all "volumes" in the "volume group" named `vg0`.

All this "volume" terminology is explained in the LVM HOWTO − if you do not completely understand the above, there is no need to worry − the details are not particularly important right now (you will need to read the LVM HOWTO anyway if you want to set up LVM). What matters is the benefits that this setup has over the many−md−devices setup:

- No need to reboot just to add a new filesystem (this would otherwise be required, as the kernel cannot re−read the partition table from the disk that holds the root filesystem, and re−partitioning would be required in order to create the new RAID device to hold the new filesystem)
- Resizing of filesystems: LVM supports hot−resizing of volumes (with RAID devices resizing is difficult and time consuming − but if you run LVM on top of RAID, all you need in order to resize a filesystem is to resize the volume, not the underlying RAID device). With a filesystem such as XFS, you can even resize the filesystem without un−mounting it first (!) Ext3 does not (as of this writing) support hot−resizing, you can, however, resize the filesystem without rebooting, you just need to un−mount it first.
- Adding new disks: Need more storage? Easy! Simply insert two new disks in your system, create a RAID−1 on top of them, make your new `/dev/md2` device a physical volume and add it to your volume group. That's it! You now have more free space in your volume group for either growing your existing logical volumes, or for adding new ones.

All in all − for servers with many filesystems, LVM (and LVM2) is definitely a *fairly simple* solution which should be considered for use on top of Software RAID. Read on in the LVM HOWTO if you want to learn more about LVM.

12. Credits

The following people contributed to the creation of this documentation:

- Mark Price and IBM
- Steve Boley of Dell
- Damon Hoggett
- Ingo Molnar
- Jim Warren
- Louis Mandelstam
- Allan Noah
- Yasunori Taniike
- Martin Bene
- Bennett Todd
- Kevin Rolfes
- Darryl Barlow
- Brandon Knitter
- Hans van Zijst
- Matthew Mcglynn
- Jimmy Hedman
- Tony den Haan
- The Linux−RAID mailing list people
- The ones I forgot, sorry :)

Please submit corrections, suggestions etc. to the author. It's the only way this HOWTO can improve.

13. Changelog

13.1 Version 1.1

- New sub−section: "Downloading and installing the RAID tools"
- Grub support at section "Booting on RAID"
- Mention LVM on top of RAID
- Other minor corrections.