Terabytes on a Diet

Peter Chubb

July 15, 2002

Abstract

You can buy a multiTerabyte raid array off the shelf nowadays. But it's not much use if you can't plug it into your trusty Linux box.

Although the block layer is in flux, there's still a lot of careless coding that means:

- Even 64 bit platforms are limited to 1 or 2 Tb filesystems (use of 32-bit signed type to hold sector number; sector size hard-coded to 512 bytes)
- Even where the partitioning scheme allows partitioning of larger discs (e.g., EFI's GPT), other limitations prevent them from being used to their full capacity
- Even though the page-cache limit is 16Tb with 4k pages (and indeed if you can create a file this big you can read and write it!) you can't have a filesystem that big.

So...

I set out to remove these limitations on both 64 and 32 bit platforms.

But how do you test support for huge (>2TB) filesystems under Linux when the biggest disc you have is 100G? Simple, write a simulator, and use a sparse file for the disc contents. But...it's not that simple.

1 The problem

Discs are getting bigger and bigger. Figure 1 (from the SCSI industry trade association, `http://www.scsita.org/statech/01s005r1.pdf`) shows ever bigger discs in the short-term future. Even though this graph was created in 1992, the year 2000 disc sizes and speeds are pretty close to spot-on. If Moore's law continues to hold, we'll have Terabyte[1] discs in our high-end desk-top machines within 5 years.

The only problem is that Linux at present doesn't support large discs. The limitations lie in several places (see figure 2).

[1] Throughout this document, Terabytes, Exabytes etc., are *binary* Terabytes. The correspondence is as follows:

Prefix		value
Mega-	2^{20}	1 048 576
Giga-	2^{30}	1 073 741 824
Tera-	2^{40}	1 099 511 627 776
Peta-	2^{50}	1 125 899 906 842 624
Exa-	2^{60}	1 152 921 504 606 846 976

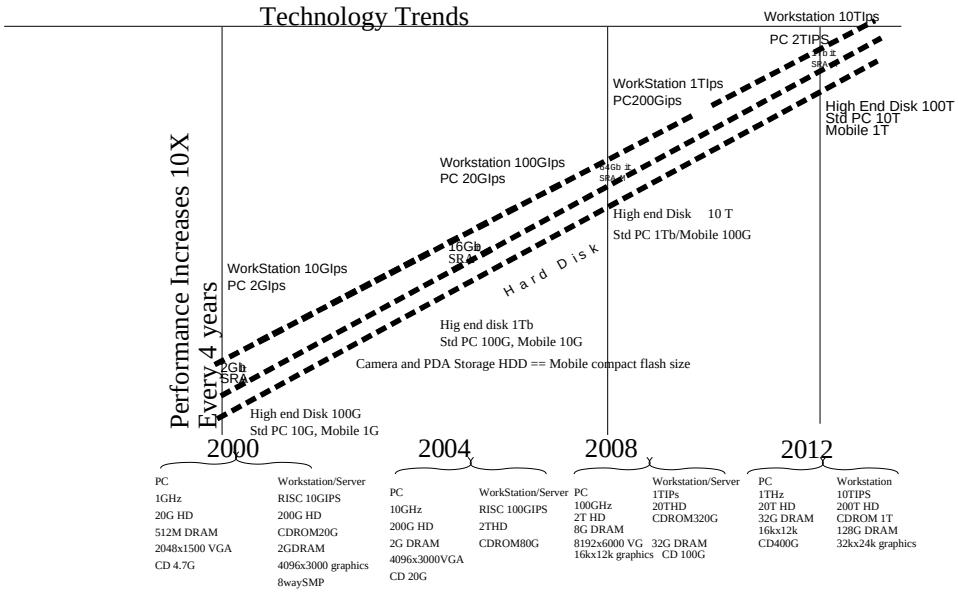

Figure 1: How discs are getting bigger

- The size in kilobytes of a block device is held in an `int`, which means the maximum size that can be held is $2^{31} \times 1024$ bytes — or 2TB.
- The size of a partition is kept as `unsigned long`, which means that on 64-bit platforms, a partition can be larger than the device that can hold it!
- The SCSI and ATA drivers use the ten-byte command set, which means that the maximum sized disc that can be accessed is $2^{32} - 1 \times$ hardware_blocksize. Many discs use 512 or 1024 byte hard sectors, so the limit is 2TB or 4TB for those discs.
- The standard ext2 filesystem layout restricts the maximum size of a block device to circa 2TB if 1k blocks are used, or circa 16TB (16384GB) if 4kB blocks are used.
- All file system operations go through the page cache, which maps a file and an index to a chunk of memory. The index is an unsigned long, and the chunk is currently one page (4k on many platforms), which with 32-bit longs means the absolute largest addressable offset within a file or block device is 16TB.
- Because of these limitations, the user-mode utilities (`mkfs`, `fsck`, etc) have never been tested with large files, and so there's no guarantee that they'll work — and in fact they do not.

2 Some Solutions

In February 2002, Jens Axboe, one of the block-layer maintainers, introduced a new type `sector_t` into the 2.5 kernel. The intent was that this

type should be used wherever sectors or blocks were counted or indexed. On 64-bit platforms, it was 64-bits; on 32-bit platforms, 32 bits now, and possibly 64 bits later on. However, it has not been used consistently.

The places sizes of discs and partitions are stored are:

1. In the `struct gendisk` there's an array of `int`, indexed by device minor number. This array holds the device size in kilobytes.

2. Also in `struct gendisk` there's an array of `struct hd_struct` indexed by minor number; `struct hd_struct` contains the size and offset of each partition in sectors, stored as `unsigned long`

3. Some of the drivers (e.g., the ataraid driver, the 'old' hd driver, etc.) declare a static array of `int` that is eventually pointed to by a `struct gendisk`. Others allocate the array with `kmalloc()`.

4. There is an array `int *blk_size[]` indexed by major and minor device number that contains the size of each disc. this is, as far as I'm aware, used directly only by the `blkdev_size_in_bytes()` function.

All these had to be changed to use `sector_t` instead of `int`, as did the access functions that are used to extract the starting sector and size for each partition.

The structure used to request a block from the block layer `struct request` already used a `sector_t` at the time I started work.

The partition recognition code was also changed to return partition offsets and sizes in `sector_t`. The only partitioning scheme that that currently helps is the EFI GPT scheme, which uses 64-bit integers on-disc to mark out the partitions.

Other partitioning schemes use `unsigned long` if you're lucky, `int` if you're not, and in any case use 32-bit on-disc numbers.

The final changes were to the SCSI get-capacity command. Code like

```
sdkp->capacity = 1 + ((buffer[0] << 24) |
                      (buffer[1] << 16) |
                      (buffer[2] << 8) |
                      buffer[3]);
```

where `buffer` is an array of `unsigned char` had to be changed to cast `buffer[0]` to `unsigned` explicitly, so that the compiler didn't convert it to an `int` then sign extend.

3 Preliminary Testing

The first thing I did was to redefine `sector_t` to be a struct, and change the obvious places where disc sizes were stored from `int` to `sector_t`, so that the compiler would show me all the places it was used, so they could all be found and fixed. Then `sector_t` could be made back into an integral type.

Having made the changes, the first thing was to see that the result worked on a 32-bit system without enabling the large block device code (i.e., with `sector_t` a `unsigned long`).

After fixing a few typos, the result worked!

OK, create a large (15Tb) sparse file, then mount it via the loop device. I found at this point, that even though writes to the device appeared to succeed, all reads failed. It turns out that if you want a block device to work, its size in sectors must fit into a `sector_t`. So I fixed error handling on the loop device.

It was now possible to create up to a 2TB (sparse) file, and create a file system on it and mount it via the loop device. Attempts to use the loop device on larger files now returned sensible errors. (I still had access to only 100G of disc).

Then the next step was to enable `CONFIG_LBD` on i386 (which turned a `sector_t` into a 64-bit unsigned type), rebuild and reboot, then see what happened.

Now, writes succeed to the loop device using a sparse file up to 16 TB minus one byte, and one can read back what one has written (and it's the same!) Hurrah.

Next was to test `mkfs` for different file systems. All the filesystems tested had essentially the same bug: they did not obtain the actual size of the device. This is because they all used virtually the same code:

```
Call ioctl(fd, BLKGETSIZE, &sz);
    If it fails,
        call ioctl(fd, FDGETPRM, &x)
        if it fails,
            do binary search to find end of partition
```

and if these all failed, (because, for instance, the size was held in a 32-bit integer that wasn't big enough) didn't notice, and tried then to create a filesystem on a device of negative size, or that was very small (depending on whether the 32-bit integer was signed or unsigned).

There were two problems here:

1. For a start, `BLKGETSIZE` returned an overflowed 32-bit number if the number-of-blocks wouldn't fit. After fixing this in the kernel,

2. the binary-search overflowed its offset

I fixed the latter by using the `BLKGETSIZE64` ioctl, which is supposed to return the size in bytes of a block device. As it turns out, it'd probably be better to just seek to the end of the device, and return the resulting offset (perhaps trying to read from the last block to make sure it's really there), as `BLKGETSIZE64` has different ioctl numbers different versions of Linux. I'm not sure why the binary search is used in all these `mkfs` (and `mkswap`, for that matter).

3.1 Ext[23]

After fixing `mkfs`, I created a large sparse file (16TB), and ran `mkfs` on it. After half an hour or so, it was still trying to write out the same inode group.... and the console was going crazy.

`mkfs` doesn't expect to fail writes because the disc has filled up. (On a *real* disc, it can't happen) (remember I had only 100G of disc).

The ext2/ext3 filesystem layout uses quite a lot of metadata — inodes and bitmaps are written to the disc at `mkfs` time. On a disc of any size, one inode per 4Megabytes of disc. Fortunately this can be controlled by a flag, to allow a sparse file on limited disc storage to be an ext2 file system.

3.2 JFS

On IA32, JFS worked like a charm (once the initial problems in mkfs were fixed). The JFS maintainers are quite responsive, and even though I found three bugs (two in mkfs, and one related to page size not equal to block size) — they're all fixed now.

3.3 The Loop Device

The loop device is a way of making a file or a block device appear as a different block device (`/dev/loop`n). As data is transferred through the loop device it can be massaged in various ways, e.g., to add transparent encryption/decryption. In addition, because a file can be made to appear as a block device, a file containing a file system image can be mounted into the file system hierarchy.

The loop device required some degree of surgery. Unfortunately, there are modules not distributed with the kernel, that do encryption etc., that are going to break with the interface changes necessary to do large block devices.

At present, I've just changed the interfaces, but probably ought to add new ones in parallel with the existing interfaces.

3.4 Fake SCSI

One way to check that the SCSI layer is working properly, is to put a simulation of a SCSI host adapter on top of a loop-like device structure. Using real SCSI adapters at present is not an option, because none of them have (yet) been audited for 64-bit cleanness (in fact, the ones I'm using to test 2.4TB file systems will work only up to 2^{32} 512byte sectors.)

There are two SCSI simulators in the Linux kernel. There is one that uses a small (8M) memory region as a disc (`CONFIG_SCSI_DEBUG`) and one that is designed to run on top of the Itanium simulator (`CONFIG_HPSIM` and `CONFIG_SIMSCSI`). I grabbed the latter, moved it into the `drivers/scsi` directory, and hacked at it with a large axe until it used Linux kernel services rather than the simulator services. Because the driver was meant only as a debugging aid, it uses extremely dubious code (I wrote `kernel_write()` to go with `kernel_read()`, modified both to throw away some error checking, and added a custom `loopscsi_open()` routine as well.

This allowed me to check that the partitioning code worked, and that large partitions could be created and recognised by the kernel.

4 Current Status of the patch

With my patch I see this:

```
SCSI device sda: 18446744073395863552 512-byte hdwr sectors (3783947180439 MB)
```

Without the patch on the same hardware I see this:

```
SCSI device sda: -313688064 512-byte hdwr sectors (-160607 MB)
```

On IA32, (Linux 2.5.24) I've tested ext3, reiserfs and JFS on linear software RAID up to 3.5TB on real hardware, and up to 15TB on the loopback device. All three filesystems work.

On IA64, (Linux 2.5.18 plus IA64 patches), I've tested ext2, reiserfs and JFS on a 2.4TB linear software RAID, but only ext2 works. I didn't test ext3 because of the well-known data-corruption problems in 2.5.18 with ext3.

JFS has issues with page sizes greater than 4k, that I believe are fixed in the current (1.0.20) release. (Even without the LBD patch, JFS from the 2.5.18 kernel would not work on this platform).

Reiserfs wouldn't work either. On mounting, one gets the error: `reiserfs_fill_super: unable to rea` because the filesystem tries to allocate a chunk of memory 173656 bytes long using kmalloc. Kmalloc thinks this is too large (the largest chunk it'll allocate is 131072 bytes).

Thus Reiserfs is limited at present to 1073709055 blocks (just under 4TB) on a 32-bit system and 536838143 blocks on a 64-bit system (just under 2TB).

The filesystem developers are responsive, and I expect all these problems to be fixed by the time this paper is presented.

You can fetch the patch from `http://www.gelato.unsw.edu.au/patches`

Whether or not the patch makes it into the mainline kernel is in one way irrelevant — what it has done is to raise awareness amongst all the developers of what the issues are, and how to work around them. Recent patches that *have* made it in have used `sector_t` correctly, and don't get in the way of future large-block-device work.

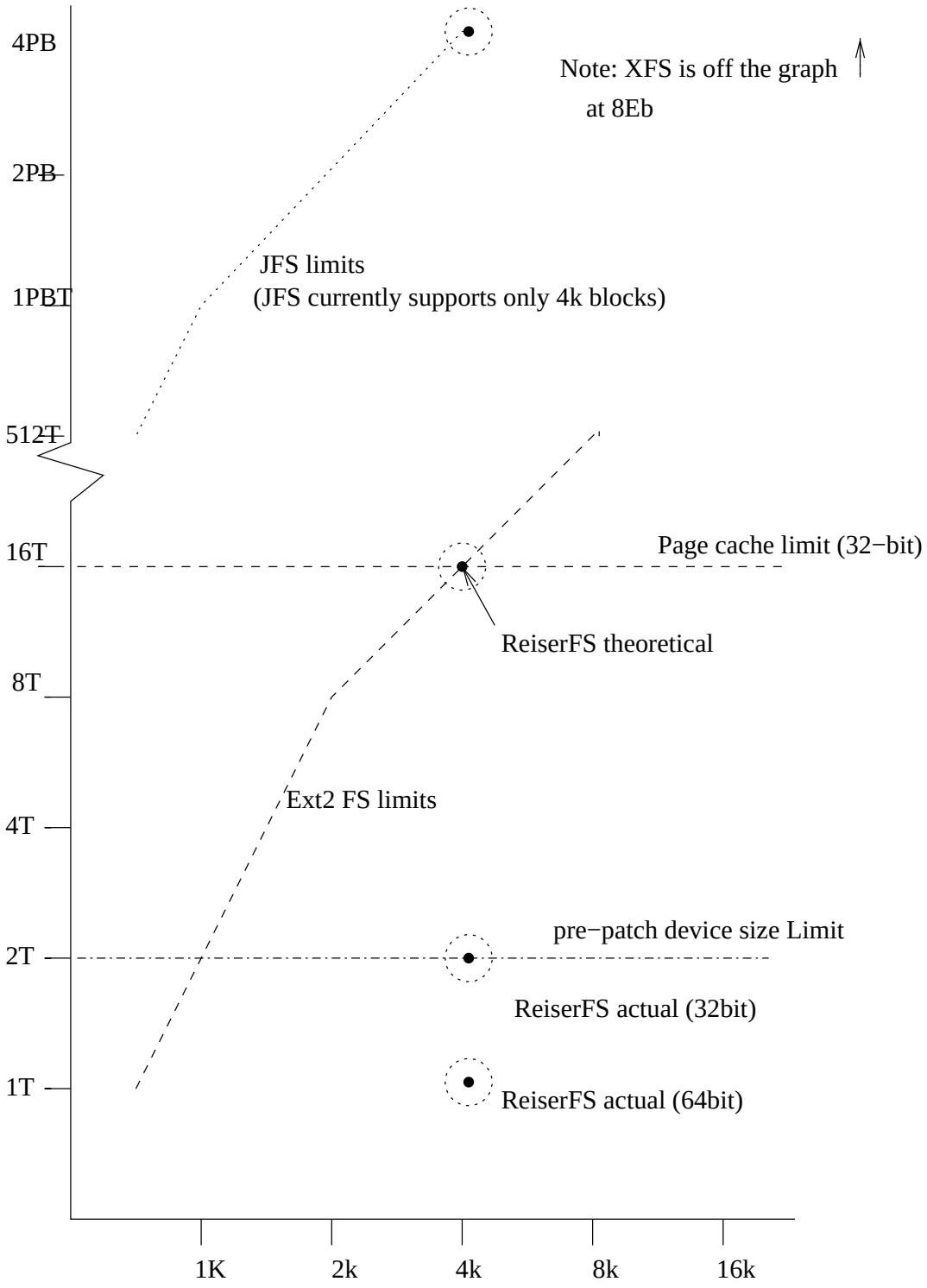

Figure 2: Block device size limitations in Linux 2.4