Testkit: A data storage system design and testing kit

Revision: 1.6

Stewart Smith
sesmith@csse.monash.edu.au

Date: 2003/06/05 00:27:11

1 Aims

The testkit package is designed to aid the design and debugging of data storage systems built on top of block devices. It aims to closely simulate how the Linux Kernel[1] does things, namely its buffer cache[2] interface.

1.1 Scope

In order of priority, testkit will simulate the Linux:

- buffer cache - for reading from and writing to block devices
- virtual file system - for reading from and writing to objects on block devices

The VFS emulation will be limited, the buffer cache should be more complete. Issues relating to SMP or preempt will be ignored for the sake of simulation.

1.2 Relation to Walnut

The buffer cache API can be directly ported across, and I would argue that it's a relatively sane programming interface that Walnut could benefit from implementing. I am aiming to keep testkit only implementing very portable concepts - i.e. that of block devices and trying to stay away from reliance on non-portable concepts such as VFS (Walnut has something much different).

2 Simulation

The simulation is implemented in the C language using the glib library[3]. Many of the data structures have been directly copied out of the Linux Kernel with irrelevent parts commented out.

2.1 Block Device

Block devices are simulated files stored on disk. It is also possible to specify a real block device as the file to emulate. A block device has a user specified number of blocks and block size. We don't store this information in the file or even attempt to guess it. We use the read() and write() UNIX system calls to access blocks from our simulated block device. These system calls are used for their simplicity.

We use a simple block_device data structure (Figure 1) to store information about our file. We are using the u64 data type so we can support large block devices. Hopefully, with the use of sparse files on a real system[4], I'll be able to test this.

[1]Currently the 2.5 series, specifically 2.5.70
[2]It's better to think of it as a block cache as that's it's real purpose
[3]Part of GTK, Glib provides functions for linked lists, hash tables etc
[4]Probably Linux running SGI's XFS

```
/*
  struct block_device
  -------------------
  A really simple block_device structure.
  $Id: testkit.tex,v 1.6 2003/06/05 00:27:11 stewart Exp $
 */
struct block_device {
  int file_on_disk;
  u64 block_size;
  u64 num_blocks;
  GList* read_blocks;
};
```

Figure 1: struct block_device

The block_dev_init function must be called before any other functions in testkit, it initializes some needed internal data structures, namely the hash of active buffers.

The block_dev_new function initializes a block_device structure, attepmts to open() the file and abort()s on failure. The initialized block_device can then be used as a parameter to other functions.

This is the extent of our block device simulation, the actual reading/writing of data is done in the buffer cache simulation (in functions such as bread).

2.2 Buffer Cache Simulation

The main function of the buffer cache is to cache disk reads. Where we would normally read in to a buffer, we read from the buffer cache. The main data structure for the buffer cache is the buffer_head structure (Figure 2). Ours is simplified from the Linux structure, mainly because we don't have to worry about other parts of the operating system.

Internally, the buffer cache tracks buffer_head structures in two indexes. One is a linked list, the other a hash. Hashing is used to allow extremely quick lookup when a buffer is requested and the list is used to allow operations on all buffers (such as flush dirty ones to disk) to be easy to implement.

```
/*
  buffer_head
  -----------
  straight from include/linux/buffer_head.h (kernel 2.5.69)
  We've comment out things we don't really care too much about.
  $Id: testkit.tex,v 1.6 2003/06/05 00:27:11 stewart Exp $
 */
struct buffer_head {

  unsigned long b_state;          /* buffer state bitmap (see above) */
  atomic_t b_count;               /* users using this block */
  struct buffer_head *b_this_page;/* circular list of page's buffers */
  //  struct page *b_page;            /* the page this bh is mapped to */

  sector_t b_blocknr;             /* block number */
  u32 b_size;                     /* block size */
  char *b_data;                   /* pointer to data block */

  struct block_device *b_bdev;
  //  bh_end_io_t *b_end_io;          /* I/O completion */
  //  void *b_private;                /* reserved for b_end_io */
  //  struct list_head b_assoc_buffers; /* associated with another mapping */
};
```

Figure 2: struct buffer_head

The API is kept remarkably similar to the Linux interface. Currently, bread() is implemented and working. A test driver program, blktest (see Appendix, Section 3.4) is designed to test the functionality of the simulation. I may implement this also as a Linux kernel module to help test that the simulation is a true one.

3 Appendix

This appendix includes the most up to date source files (as of document generation). The revisions may be different to those discussed in the body of the document.

3.1 types.h

3.2 block_dev.h

3.3 block_dev.c

3.4 blktest.c

```
/*
  blktest.c
  ---------
  Test driver for the block device simulator.

  $Id: blktest.c,v 1.2 2003/06/01 16:03:49 stewart Exp $

  (C)2003 Stewart Smith
  Distributed under the GNU Public License
 */

#include "block_dev.h"
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc,char* argv[])
{
  struct block_device b;
  struct buffer_head* bh;
  int a;
  int i;

  if(argc<3)
    {
      fprintf(stderr,"Usage:\n\t./blktest device blocksize blockcount\n\n");
      exit(0);
    }

  block_dev_init();
  block_dev_new(&b,argv[1],atoi(argv[2]),atoi(argv[3]));

  for(i=0;i<atoi(argv[3]);i++)
    {
      bh = bread(&b,i,atoi(argv[2]));
      a = write(STDOUT_FILENO,bh->b_data,bh->b_size);
    }

    for(i=0;i<atoi(argv[3]);i++)
    {
      bh = bread(&b,i,atoi(argv[2]));
      a = write(STDOUT_FILENO,bh->b_data,bh->b_size);
    }

  return 0;
}
```