Operating Systems: Practical Two

Mike Spivey

Michaelmas Term 1993

1 The Task

The MINIX file system maintains a cache of disk blocks that have been used recently. This reduces the amount of disk I/O, because blocks are frequently needed several times in a short period of time, and keeping them in the cache removes the need to read them from the disk again or to write updated blocks immediately. The aim of this practical is to assess the effect of different cache sizes and different replacement policies on the effectiveness of the cache.

You are provided with a file that records about 38,000 block requests made by the file system of a running MINIX system. This file was obtained by recording the activity of the file system as the machine executed a collection of I/O-intensive programs. (In fact, the machine was re-compiling MINIX itself). The sequence of blocks requested by the file system is independent of the way the cache is implemented, so it can be used as test data to evaluate different cache sizes and policies. Your task is to build a simulation of the block cache, collect relevant statistics, and make recommendations about improving the cache.

For the last part of the practical work, you will need a sheet of graph paper.

2 Block replacement policies

You will compare three replacement policies for blocks in the cache. When a requested block is not found in the cache, the replacement policy determines which block from the cache is evicted to make room for it. The first replacement policy is to choose the evicted block at random. The second is FIFO, to evict the block that has been in the cache for the longest time, so that the buffers are re-used in rotation. The third policy, LRU, is the one actually used in MINIX. Under this policy, the cache implementation keeps track of how recently each block in the cache has been used, and chooses the least recently used block for eviction.

In the simulation, a block replacement policy is represented by a set of functions that simulate the actions of the cache; examples of these functions for the RANDOM replacement policy are provided in the practical kit, and it is your job to add the functions for the FIFO and LRU policies. The practical kit also contains a driver program that reads the data file, calls the simulation functions, and prints statistics.

3 The Test Data

The file `data` contains about 38,000 lines, the first few of which look something like this:

```
R 1234
Z 2345
W 2345
R 1234
W 2345
...
```

Each line in this file records a request made of the block cache by the rest of the file system; it contains a letter, `R` or `W` or `Z`, and a block number. The block number is a natural number in the range 1 up to 65,535 (because I have a 64MB disk), and the letter shows what operation was requested on that block:

`R` indicates a read request. If the block in question is not in the cache, then this request causes it to be read from the disk.

`W` indicates a write request. If the block is not in the cache already, it must be read from the disk, because the data written may be only part of the block, and the rest is unchanged. Afterwards, the block is *dirty*: it must not be discarded from the cache without first writing it back to the disk.

`Z` indicates a zero-block request. This is a write request for a newly-allocated block. It is not necessary to read the block from the disk; the buffer can be filled with zeros instead, but afterwards the block is dirty.

This is a slightly simplified picture of the activity of the cache, because it assumes that each use of a block is a single event. In fact, requesting a block and returning it are separate events, and clients of the block cache may request and use several blocks at once, keeping them for some time before returning them. In practice, though, this simple picture is good enough, because ordinary disk reads and writes use only one block at a time. The file system requests certain special blocks (e.g., bitmaps) and keeps them for as long as their disk is mounted; for ordinary purposes, we can think of the buffers they occupy as being removed from the working cache space, so that the effective cache size is smaller by the number of such blocks in use.

The data file also assumes that all block requests are for a single disk, although the buffer cache in real MINIX keeps track of the device for each block as well as its block number.

4 Simulation structure

Each simulation is built from the driver module `driver.c` (see Appendix B), together with a module that simulates the replacement policy. One such module is `random.c` (see Appendix C), which implements the RANDOM replacement policy. The interface between the two modules is described by `sim.h` (see Appendix A). It consists of some shared global variables and three functions defined by the policy module.

- The function `init_cache` initializes the cache with `nbuffs` buffers. The driver module ensures that the value of `nbuffs` is between 1 and `MAXBUFS`, which is defined as 4096. Typically, `init_cache()` sets each buffer to contain the fictitious block `NO_BLOCK`, which is defined as 0 and guaranteed different from any genuine block. This function may also set up data structures such as a hash table and an LRU queue.

- The function `use_block` has this heading:

  ```
  PUBLIC void use_block(action, block)
  int action;
  int block;
  ```

  The `action` argument may be `READ`, `WRITE`, or `NEW`. The function simulates this action on block number `block`. For the simulation, there is no need to bother with the actual data that is transferred, but the function must determine which buffer to use and whether any blocks must be read or written. If the requested block is read from the disk, `use_block` must increment the global variable `nreads`, and if a dirty block is written to the disk in order to free its buffer, the function must increment `nwrites`.

- The function do_sync simulates the action of a sync system call by writing all modified blocks in the cache back to the disk, incrementing the global variable nwrites for each block written. It is called at the end of the simulation to make sure all simulation runs finish with the disk in the same state.

5 Practical instructions

In the practical, you will first build a supplied simulation of the RANDOM replacement policy, then improve it by adding a hash table to make large caches feasible. Next, you will modify the simulation to implement the FIFO and LRU policies. The code that implements each policy need be no longer than 100 lines of C, and much of that code is the same in each simulation. Finally, you will run your simulations on a selection of different cache sizes and analyse the results.

The directory /usr/local/opsys/prac2 contains the files you will need to carry out the practical:

Makefile	Project description file
sim.h	Header file for simulations
driver.c	Source code for simulation driver
random.c	Source code for RANDOM replacement policy
data	Test data
tabulate	Shell script to run simulations

You should make a new directory, and in it put copies of all these files. The file data is about 1/4 megabyte; you will not need to change it, so you could make a symbolic link instead of a copy if space is tight.

Your practical report should include listings of the code you write, the table and graph of simulation results from Section 5.5, and comments on the main features of the results.

5.1 Random replacement

The file random.c contains an implementation of the RANDOM replacement policy. There is an array buf of buffers, declared like this:

```
PRIVATE struct buffer {
     int b_block;
     int b_dirt;
} buf[MAXBUFS];
```

Each element of this array is a structure (of type `struct buffer`) containing a block number, and a flag, which is either `CLEAN` or `DIRTY` depending on whether the buffer needs to be written to the disk. In a real cache implementation, the buffer would also contain the data of the block, but that is not needed for the simulation.

In other parts of the program, pointers to buffers are declared like this:

```
struct buffer *b;
```

They are given values by assignments like this one, which sets `b` to the address of buffer number 3:

```
b = &buf[3];
```

After this assignment, the statement `b++` increases `b` by the size of a buffer, so that it now points to buffer number 4. The comparison `b < &buf[nbuffs]` is true if `b` has not run beyond the end of the cache. So a `for`-loop like this sets `b` to each buffer in turn:

```
for (b = &buf[0]; b < &buf[nbuffs]; b++)
        ...
```

There are three functions that define the replacement policy:

- The `init_cache` function initializes all the buffers so that they contain the fictitious block `NO_BLOCK` and are not marked as dirty.

- The `use_block` function searches the cache for the requested block. If the block isn't there, another block is chosen at random for eviction; we count one block write if the evicted block is dirty, and one block read unless the action is `NEW`. Finally, if the action is `WRITE` or `NEW`, we mark the requested block as dirty.

  Although there is a standard C library function for generating random numbers, the simulation uses its own function `randint(n)`, which returns a random integer in the range 0 up to $n - 1$. This function is probably inferior to the library function as a random number generator, but it generates the same pseudo-random numbers on all our machines, making the results of the simulation repeatable.

- The `do_sync` function examines each buffer, and counts a block write if the buffer is dirty.

Say `make random` to compile the driver program and the file of code that implements the RANDOM replacement policy, and link them together into an executable program called `random`. Now you can run the program on the test data using the command

```
    random 100 data
```

which uses 100 buffers. The output will be a single line like this:

```
        RANDOM      100    37888    14696    1045    15741
```

This means that the RANDOM policy with 100 buffers carried out a total of 37888 operations with 14696 reads and 1045 writes, a total of 15741 disk transfers.

5.2 Hashing on block numbers

Unfortunately, the supplied implementation of the RANDOM replacement policy uses a naïve linear search to find out whether a requested block is in the cache. This is almost workable for a cache of 100 buffers, but very slow for big cache sizes.

By copying the code in the MINIX buffer cache `fs/cache.c` (or otherwise), you should implement an open hash table for buffers. Add a pointer to each element of the `buf` array so that they all the blocks with the same hash code can be linked into a chain. Add an array of `HASHSIZE` pointers to the heads of these chains, choosing `HASHSIZE` to be (say) 128. Modify the simulation functions to set up and use the hash table, with the hash code of a block being its block number reduced modulo `HASHSIZE`:

- Make `init_cache` link all the buffers into a single chain, attached to the entry for `NO_BLOCK` in the hash table.

- Make `use_block` search the appropriate hash chain for the requested block, rather than looking at every buffer. If the hash chain is linked together by a field called `b_link`, the search is achieved by a loop like this:

  ```
      for (b = bufhash[block % HASHSIZE];
                           b != NULL; b = b->b_link)
            if (b->b_block == block) goto found;
  ```

  The `for` statement initializes `b` to the first buffer in the hash chain for `block`, then repeatedly moves to the next one using the `b_link` field, until either the right block is found, or the `NULL` pointer at the end of the chain is reached.

  When a block is evicted and replaced with another, you will need to delete the buffer from the hash chain for the old block and add it to the chain for the new one. This means another search of a hash chain, this time keeping track of the previous buffer in the chain, so its `b_link` field can be updated to skip the evicted buffer.

When you have done this, test the program again with 100 buffers; the results should be exactly the same as before. If you like, use the `time` command of UNIX to see the improvement in run time for your simulation.

Debugging suggestions

Add calls to `printf` so that your simulation outputs the decisions it makes. Try running it with a very small number of buffers on a small fragment of the data file: say

```
head -20 data >little
```

to make a file `little` that contains only the first 20 block requests, or make your own test data with a text editor. Make sure that your program is robust. For example, what happens if you try to delete a block from a hash chain and the block isn't there?

5.3 FIFO replacement

Make a new copy of your enhanced version of `random.c` called `fifo.c` and edit it to implement the FIFO policy. Hints: add a global variable `nextbuff` that contains the index of the next buffer to be (re-) used, and use it to select the buffer to be evicted, in place of the `randint` function. Make sure that `nextbuff` is incremented each time a block is evicted, and that it wraps round properly at `nbuffs`. You can also delete the function `randint` from the program, since it is no longer used.

The string `policy` defined near the top of the file is used by the simulation driver to label the line of statistics it prints: you should change it to say `"FIFO"` instead of `"RANDOM"`. Say `make fifo` to compile and link your new module into an executable `fifo.c`.

Sanity check

The command `fifo 100 data` produces the output

```
FIFO      100    37888    15099     1105    16204
```

Your program should produce exactly this answer. If not, go back and check things carefully. See the debugging suggestions in Section 5.2.

5.4 LRU replacement

Make another copy of `random.c`, this time called `lru.c`, and modify it to implement LRU replacement. You will find it useful to study the MINIX code on pages 595–600 of the book, and you will need to add more pointers to each buffer so that they can be made into a doubly-linked list. Change the `policy` string to say `"LRU"`. This time, say `make lru` to compile and link your program.

Sanity check

The command `lru 100 data` produces the output

```
          LRU     100   37888   14638    1009   15647
```

Again, there is no reason why your output should be any different from this, so if it is, you've done something wrong! See the debugging suggestions in Section 5.2.

5.5 Collecting statistics

The supplied file `tabulate` is a (Bourne) shell script that runs your three simulation programs for lots of different cache sizes. Here it is:

```
echo "  Method    Bufs    Reqs   Reads  Writes   Total"
echo
for n in 1 10 100 200 300 400 500 700 1000 1300 1600; do
        for m in random fifo lru; do
                $m $n data
        done
done
```

If you say `sh tabulate`, your simulations will be run a total of 33 times (11 times for each program), and you will get a table like this:

```
  Method    Bufs    Reqs   Reads  Writes   Total

  RANDOM       1   37888   99999    9999   99999
    FIFO       1   37888   99999    9999   99999
     LRU       1   37888   99999    9999   99999
  RANDOM      10   37888   99999    9999   99999
    FIFO      10   37888   99999    9999   99999
     ...      ..     ...     ...     ...     ...
```

This may take some time; but it would take longer without the hash table! Include this table in your practical report.

Now make a graph with cache size on the horizontal axis and total number of disk transfers on the vertical axis. Draw three lines, one for each replacement policy. Try to explain the obvious features of the graphs:

1. What happens with only one buffer?
2. What happens if the number of buffers is very large?
3. Why are the ‘rational’ policies sometimes significantly worse than the random one?

If the test data is typical of system load, what replacement policy and what cache size would you suggest for a reasonable compromise between memory space and performance?

Appendices

Appendix A `sim.h`

Appendix B `driver.c`

Appendix C `random.c`