p3q8
SMH
p10q9

OS Functions (1b): Tripos Questions 1998/99

1 Question 1

FIFO, LRU, and CLOCK are three page replacement algorithms:

a) Briefly describe the operation of each algorithm. [6 marks]

b) The CLOCK strategy assumes some hardware support. What could you do to allow the use of CLOCK if this hardware support were not present? [2 marks]

c) Assuming good temporal locality of reference, which of the above three algorithms would you choose to use within an operating system? Why would you not use the other schemes? [2 marks]

What is a *buffer cache*? Explain why one is used, and how it works. [4 marks]

Which buffer cache replacement strategy would you choose to use within an operating system? Justify your answer. [2 marks]

Give two reasons why the buffering requirements for network data are different than those for file-systems. [2 marks]

1.1 Answers to Question 1

1.1.1 Brief descriptions

Two marks each for FIFO, LRU and CLOCK.

FIFO — one maintains a queue of pages, ordered by the time at which they were brought into memory. When time comes to replace, choose the pages at the head of the queue (i.e. one which has been in memory the longest).

LRU — this is *least recently used.* I.e. one tries to maintain an ordering on pages according to their *use.* Assuming one has done this, then one replaces the page which has not been used for the longest amount of time.

Clock — this is an approximation to LRU. One maintains a queue of pages as per FIFO (i.e. ordered by time brought into memory). When replacing, however, check the *reference* bit. If this is set, the page has been referenced since while it was on the queue, and it is considered to be more useful, and is put back on the tail of the queue with its reference bit cleared.

1.1.2 Supporting Clock w/out h/w reference bit

The CLOCK strategy requires a reference bit to be set every time a page is "touched" (read, written, executed). This is cleared whenever the page is recycled to the tail of the queue. If the hardware does not support per-page reference bits, the operating system can simulate it by *unmapping* the pages it puts at the tail of the queue (i.e. clearing the valid bit of the PTE in the page table & flushing the entry from the TLB).

If any of these pages are referenced, a page fault will occur which can be immediately resolved since the page is still in memory. Hence if a page is still unmapped when it arrives at the head of the queue, it has not been referenced since being placed on the tail. This gives the same effect as a referenced bit, for slightly more cost (since the OS may take more page faults than it would otherwise).

1.1.3 Algorithm of choice

One would choose CLOCK because:

- FIFO is an inherently poor algorithm: it does not distinguish between "hot" (recently / highly used) pages and "cold" ones. It is cheap to implement, but CLOCK gives better performance with little additional overhead. FIFO does o.k. if we have no temporal locality of reference (but so does "random"!).
- LRU performs nearly optimally when we have lots of locality of reference. It also copes better than CLOCK if e.g. all the pages in the queue are continually being referenced, since it gives us an ordering on this set of referenced pages so that we can make a good choice to replace (which CLOCK does not). However it is prohibitively expensive to implement, requiring a large number of book-keeping memory references for every "normal" memory reference. Then benefits are typically not worth it.
- CLOCK approximates LRU, but with an implementation cost close to FIFO. This is a good trade-off.

A well-reasoned argument for LRU here (including a sensible sketch of a design to support it) might *win some marks. Given the caveat (*"Assuming good temporal locality of reference..."*) it would be difficult to give any marks if FIFO were chosen.*

1.1.4 What is a buffer cache?

A buffer cache is a cache of blocks/sectors of a disk. It is used to reduce the latency of disk reads and writes (including write merging). The buffer cache uses (typically) nailed-down OS memory which is partitions into n blocks of k bytes each. The block size is usually the same as that used by the file-system (i.e. the *logical* block size), e.g. 512 bytes – 8K.

Whenever a read occurs, the buffer cache is first searched to see if a copy of the relevant block is present. If not, the block is fetched into the buffer cache (In some systems, neighbouring (or subsequent) blocks will also be *prefetched.*). Once the block is resident, the read request is satisfied from the cached block.

On writes, the relevant block is first located in the cache (or read in if necessary). The modification is then made to the cached copy. In order to preserve multi-reader/single-write semantics, it may be necessary to copy the block first. Modified blocks are usually flushed back to disk asynchronously — e.g. `update` on Unix.

Since there only a finite number blocks can be resident in the cache, it is important that "useful" ones are pre-fetched (if any prefetching occurs), and that "useless" ones are replaced when it becomes necessary. It is also necessary to handle meta-data carefully: if it shares the same buffer cache as data, then it may be important to ensure that the meta-data is flushed more regularly and/or two-phase written. Some buffer caches (e.g. NT) are *unified* — the same pages are used to hold file-system and virtual memory buffers. The NT buffer cache also allows *write-through* behaviour to be forced for a given file: this is useful in handling concurrency issues.

1.1.5 Buffer Cache Replacement

Either LRU or LFU are reasonable candidates. The former keeps track of when each block is accessed and, when it comes time to replace something, chooses the block which has not been used for the longest amount of time. This can be implemented by keeping all the blocks in the cache on a queue, and placing blocks onto the tail of this queue whenever they are referenced. Thus the head of the queue is, by definition, the least recently used.

LFU (least-frequently used) is also plausible: this maintains a counter in each block and increments it every time that block is accessed. When a replacement is to be made, the block with the lowest counter value is chosen. It may be convenient to maintain an ordered list to prevent an $O(n)$ search at replacement, but since n is small it is not really required. LFU gives us a better idea of "hotness" of a block, but can suffer from "time-dilation" — i.e. a large number of small writes to the same block in a short time artificially inflates its value compared with another block where writes are coalesced in user-space.

The key thing here is that LRU/LFU are feasible to implement here since time-scales are so much bigger than the page replacement case. No reason to stoop to approximate LRU.

1.1.6 Network Buffering Differences

There are a bunch of these, in rough order:

1. Variable sized data is the norm (coz of protocol layering and lack of h/w "block size"). So fixed size blocks are not sufficient (although can be in terms of alloc, e.g.

`mbufs`).

2. Cacheing doesn't exist: data has a temporal aspect, and holding old versions of packets is not useful.
3. Allocation is sporadic: can't prefetch, or choose size, or whatever. Driven by the (external) network on receive.
4. Locality of reference doesn't exist (sim. to 2).
5. No "write-merge" benefit in delaying transmission.
6. May wish to treat a logical unit (e.g. packet) as a set of smaller logical units (e.g. fragments, headers, etc) $\Rightarrow$ need to be able to chain things together.

2 Question 2

The following are three ways which a file-system may use to determine which disk blocks make up a given file.

a) Chaining in a map.

b) Tables of pointers.

c) Extent lists.

Briefly describe how each scheme works. [2 marks each]

Give two benefits and two drawbacks of using scheme (c). [4 marks]

One can protect access to files by using *access control lists* or *capability lists*. Compare and contrast these two approaches. [6 marks]

You are part of a team designing a distributed filing system which replicates files for performance and fault-tolerance reasons. It is required that rights to a given file can be revoked within T milliseconds ($T \geq 0$). Describe how you would achieve this, commenting on how the value of T would influence your decision. [4 marks]

2.1 Answers to Question 2

2.1.1 Brief descriptions

Two marks each for chaining in a map, table of pointers, and extent lists.

Chaining in a Map — in this scheme we keep an array of e.g. 4-byte entries for every e.g. 4K block on the disk (partition). Each entry contains a "pointer' (usually an