CST Part IA: Operating System, SV 3
Joe Yan
2017-3-8

1 2006P1Q7

(a) (i) FIFO FIFO maintains a queue. When a page fault occurs the new page is offered to the back of the queue and the old page gets removed from the front of the queue to keep the size of such page table size constant.

LRU When a page fault occurs, the OS will discard the page which has been unused for the longest time. To implement this strategy, each page will need a time label which stores the latest time the page is touched such as referenced, read, executed or written so the OS can tell which page is not used for the longest time. To give such time label, a timer hardware will be needed to generate a time label for a page whenever the page is touched.

CLOCK Pages can be imagined to form a closed ring and each page will have a bit to record whether the page is touched. Define it called R bit and write 1 to R bit whenever the page is touched. A pointer will be needed to point to one of such pages and supports advancing to the next page in the page ring. When a page fault occurs the OS will check the current pointed page. If it has a R bit 0 the page is not recently touched and so get replaced with the requiring new page. Otherwise it has a R bit 1 the R bit will rewrite as 0 and the pointer will advance to the next page in the ring until it is pointed to some page with R bit 0. The process always terminates because there is a constant number of pages in the ring the pointer will always return to the page with R bit modified as 0.

(ii) One extra bit for each page is needed for recording whether the page is recently touched. This needs hardware support.
The OS can achieve similar effect by half-discard the page which means when the pointer reaches a page the pointed page will be set as invalid in the corresponding page table entry. As the pointer keeps being advanced if such page is referenced then OS will simply put the valid bit back to 1 and the page is still in the memory or cache anyway so the computer will not do the costly disk IO. Otherwise the page is not referenced until the pointer meets to again, if the page is still invalid in the page table entry, it will be fully discarded and replaced with the new requiring page.

(iii) FIFO will definitely be unsuitable because FIFO can not distinguish which page is not used recently used. FIFO judges the value of page by how old the page is but old page can also be used very frequently for example some dynamic linked library.
LRU is a reasonable candidate. It is a good approximation to the optimal page replacement algorithm by storing when each page is touched last time. But LRU is space costly for storing a time label for each page and also time costly for rewrite the time label whenever the page is touched.
CLOCK is a second chance algorithm. It gives every page a second chance if they are touched recently so it is actually a good approximation to the LRU and the implementation of CLOCK is much cheaper than the LRU in the aspect both hardware and time/space cost.
Overall LRU costs more but has a better effect than CLOCK. Both of them are possible choice.

(b) CPU computes very fast. To make it free of IO bound when it fetches the data from the memory people split the memory to register, buffer cache and memory. The IO speed if cache will be much faster than the memory but it is very expensive. When CPU asks

for some data or instructions it always looks for a copy in the buffer cache first. And so buffer cache should store the most frequently using page. If the page is not in the buffer cache, the requiring page will be copied to the cache from memory by some page evaluation algorithm which estimate the page is worthless to be replaced. CPU also writes directly to the page in the buffer cache to get the largest speed. When the page is finally removed from the buffer cache. If it is modified the modified page copy will rewrite the its prototype in the memory and then being discarded in the buffer cache to leave space for more important pages.

(c) As discussed in (b), how the OS evaluate the page value in the buffer cache is how frequently the page is used. The OS first needs a timer to generate a time unit, a counter counting how many times the page is used in the last time unit and a evaluation function to estimate how frequently a page will be used in the next time unit. e.g. $f_{t+1} = Af_t + (1 - A)f_{t-1}$ *where* $A \in (0, 1)$ Such recursion definition will be cheap to calculate and the older frequent record will keep fading away. The function is likely to give a close enough estimation how heavily the page will be used in the next time unit. So every time CPU asks for a page which is not in the buffer cache the page with lowest estimation value $f_t + 1$ will be discarded to leave enough space for the new page.

2 2006P1Q8

(a) (i) Address lines, data lines and control lines.

(ii) The CPU uses bus to make a request to the device and continues to do other processes while device is fetching data. Device will raise a interruption when it is ready to send data to CPU. CPU receives the interruption and vectors to the device handler to read data from the device. After finishing the data reading, CPU restore back to the context what it was doing.

(iii) DMA allows devices to read and write processor memory directly. So the bus needs some mechanics to determine the current master and the potential masters. Also the OS should provide memory protection against DMA devices.

(iv) There many different types of buses which have different bandwidth, parallel or in series, synchronous or asynchronous. Put everything in the same data bus may lose such flexibility. Also some times we want to add or remove devices from the bus, which will be restricted if there is only one bus.

(b) (i) We do not have source code but when the program gets executed, the ARM instruction will always stays in the memory. When the program runs, assuming to give the process some specific input to make it do some instruction. The the program counter will keep fetching different instructions in the memory. We can count for the instruction being used for such input then see which part of the memory addresses is frequently being fetched by program counter. Then we can say these instruction is used for these specific input. If the performance is bad, we can try use ARM or machine code to rewrite these instructions in the program in the disk.

(ii) A bit shift operation or an add operation will be cheaper than the multiplication.
Replace $x * 8$ with $x = x << 3$
Replace $x * 15$ with $x = (x << 4) - x$
These two replacement may improve the performance of the program if it is heavily used.

3 2012P2Q4

(a) (i) $70 < 2040$
Only the first block is read.

(ii) Notice the number of bytes in the first block and the direct block pointers pointing to is $2040 + 1024/4 * 4096 = 2^{20} + 2040$. The byte needed is 4 bytes after this which means it is in the first block indirect pointer pointing to. So 3 blocks are read. They are the first block, the first first indirect block then the block pointed by that pointer where the asked byte actually stays.

(b) The answer is partially answered in (a). All data access with up to 3 blocks read must be in the first block, the direct block or the first indirect block by the argument in (a). Sum of such three types of bytes is $2040 + 1024/4 * 4096 + 4096/4 * 4096 = 5244920$

(c) (i) The time of creation is stored in the index node because a file is represented uniquely by a index node.

(ii) A file has a unique index id. (Is this called name?) Or all possible paths from the root of the file system to the specific file are the name of this file.

(iii) The file access rights are stored in the index node by the reason of uniqueness again. That means every time one subject try to access some data from a file, such subject must pass the index node rights check.

(d) This method saves the space for block pointers and also need less block read to reach specific byte by adding the number of blocks to b. This read method is based on the contiguous property.
When write to the file. If the write operation does not change the size of data, it would be fine. But when the size of data increase the file will need some more blocks to store them. The case that needed N block after the last block of this file is free is easy to deal with. Otherwise, the OS will either find a whole large enough chunk to copy the whole file to and append or use a chain structure to make the last contiguous block contains a pointer to point to the next block. None of them are capable. Copy the whole file will be slow. A chain structure will be need much more times of block reading for those non-contiguous parts of the chain than the design previously. And also this will cause fragmentation in the disk. Maybe a idle process keeps cleaning up the mess in such file system will be useful.
If the process which creates the file can guarantee that the size of such file is constant once the file is created then such file system may be useful. But this is just putting unnecessary restriction on the file creation. Otherwise when creation files, there is no way for OS system to know how large the file will be in the future. And to maintain the compaction of such file system will be extreme expensive due to frequently moving file around in the disk.
Overall, this is a bad implementation if there is no good solution for either maintaining the contiguous property or build a efficient mechanics for an inner-file chain.