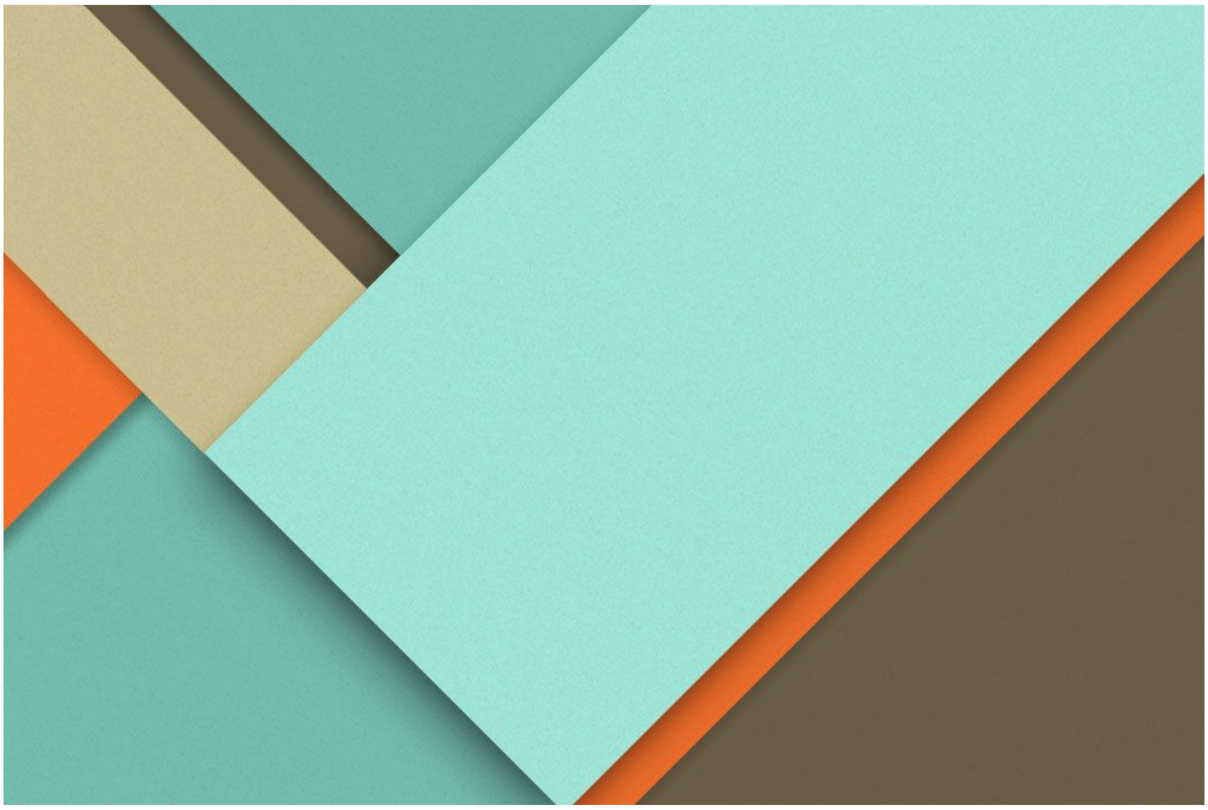


CS3210 Final Project Write-up

04/20/2017

— **Extend Paging System of JOS**

Chunyan BAI
Haoye CAI
Yanzhao WANG

Introduction

As we know, in JOS, the physical memory is limited to 256MB, which means the needed memory of a single process cannot exceed 256MB. One commonly implemented solution to the problem of limited memory is paging to disk. A paging system swaps memory pages between RAM and disk, allowing a process to use more memory than physically restricted. Since accessing permanent storage is much slower than accessing RAM, it is important for a good paging system to choose the right pages to swap out, or more specifically, to choose those pages that will rarely be used in the future. To accelerate this swapping process, we will also modify the process scheduler so that it can coordinate with our paging system more smoothly.

Out of all the options, this feature interests us because although the concept is simple and fundamental, lots of skills can be applied in order to improve the efficiency of the paging system.

Background

Paging to disk is widely used in different operating systems nowadays, and it has become a crucial feature in order to break the memory limit. In Unix and other Unix-like operating systems, it is common to dedicate an entire partition of a hard disk as swap space. Many systems even have an entire hard drive dedicated to swapping, separate from the data drives, containing only a swap partition. In terms of determining pages to be swapped out, which is one of our main focus, there are some existing page swapping selection heuristics, such as FIFO, LRU, NFU, etc.

Problem

We aim to extend the functionality of paging system in JOS, implementing paging to disk so that virtual memory could exceed RAM . Furthermore, we intend to propose a novel paging heuristic in order to increase the performance of paging system, and also explore the influence of different page replacement algorithms on paging system.

Approach

I. Paging to Disk

When a new page is needed, a page fault (trap) is generated to fetch a new page from disk. We implemented the paging library in a microkernel style, having a processing spinning and waiting for IPC calls to perform read/write to disk, while syscalls only send IPC requests and avoid doing real work in the kernel.

II. Extend system call

We need to extend system call including sys_page_alloc, sys_page_map, and sys_page_unmap, since they deal directly with page mapping and allocating. Other syscalls only need a small amount of wrapping code to ensure the virtual address passed in are not paged out.

sys_page_alloc is designed to return -E_NO_MEM when no more page can be allocated. To extend this, we created the page_alloc function which wraps sys_page_alloc, and calls page_out whenever sys_page_alloc returns -E_NO_MEM, followed by calling sys_page_alloc again.

sys_page_map is wrapped by page_map. In the special case where we map from a page that is paged out, we simply access the page to page it back in. In the special case where it would map over a page which is paged out, it sends an IPC to the paging server to discard the page. Otherwise, page_map behaves as sys_page_map normally does. Similarly, sys_page_unmap is wrapped to handle the case in which we unmap a page that was paged out.

The second functionality of the paging library is handling paging in when we try to access a page that has been paged out. In order to handle this, we implemented a user-mode page fault handler of its own which specifically checks for the paged out case and triggers paging in as needed.

III. Paging server

The paging server is a special environment (process) which continuously spins and processes incoming IPCs. We defined 4 possible codes for IPCs corresponding to page out, page in, discard page, and get page stats.

We dedicated a swap partition directly after the FS partition, for the storage of paged out pages. The paging server contains an in-memory bitmap, which marks which blocks of the partition are used. The paging server also provides functionality for doing the IDE writes/reads to/from the swap partition.

In order to page out a page, the paging server should be sent an IPC that includes the page to be paged out. The paging server then finds an empty block on the hard drive (using the bitmap), writes the page's contents to that block, then sends an IPC in return which contains the index of the block on disk to which the page was written. This index is 20 bits long, in order to fit in the mapping table system managed by the paging library code.

Page in does exactly the inverse: given a 20-bit index, the paging server reads the page from disk, shares the page with the requesting environment, and frees the block via the bitmap. Discarding skips the first of these two operations, and merely flips a bitmap bit to mark the given block as free.

The paging server keeps track of how many times the previously three IPC handlers have been called. Calling the fourth IPC handler, a stats function, will write this data into a page and share that page with the requesting user environment.

IV. Page replacement algorithms

For this part we intend to explore different page replacement algorithms. A page replacement algorithm selects a page from memory and writes it to disk (page out) when a page of memory needs to be allocated. We have implemented four regular algorithms and also come up with a novel method by ourselves. The paging functions are written in the lib/paging.c file, and we use a function pointer to select the page replacement policy adopted by the page server.

The details of the algorithms we have implemented are as follow:

1) Linear Probing: A simple paging strategy that iterates all page entries in the memory and evict the first available page directly, without taking other factors into consideration.

2) LRU (Least Recently Used) algorithm: Select the least recently used page in the memory to replace. To get the timestamp of a page we simply maintain an internal clock instead of reimplementing the “time.h” in C standard library. Although we didn’t get the exact time, this should still be a good approximation. This algorithm follows the assumption that a page which is not frequently used is less likely to be referenced in the future.

3) Random Page Replacement: A straightforward method that select a page in memory at random. A potential problem with this algorithm is that our paging server may reference a page that is not free for multiple times. To save the time cost, we locate the first free page in the memory through linear probing and then select a page in the rest of memory randomly.

4) NFU (Not Frequently Used) algorithm: A paging policy that is an approximation to LRU, which replaces the least frequently referenced page in the memory. A counter is set up in each page to record the time that a page has been referenced during an interval, and the paging server select the page according to value of the counters.

5) Novel method: modified version of NFU which take into account the time span of page references. It is basically a combination of NFU and Aging, with some elegent tricks. Firstly, we maintain an aging value in each page, and we incorporate the recent reference counter mechanism into the aging value. To

be more specific, we initialize the aging value to a maximum. If a page is not referenced, we deduce the aging value as time goes by. If a page is referenced, on the other hand, we increase the value instead. In other words, we utilize the reference times to delay the aging. Secondly, we increase the decrement value for the pages that are frequently referenced during a time interval. The reason for doing this is that, after conducting some experiments, we found that intensively used pages are more likely to be abandoned after usage. Thirdly, in order to improve time efficiency, instead of walking through the entire page table, we just scan a small portion of it. The percentage for scanning is determined by the aging value of a page: we walk through a large percentage of the table if the age value is large, and walk through a small portion if the aging value is small.

V. Performance metric

To compare the efficiency of different paging algorithms, our paging server records the number of page in and page out and calculate the ratio of page_in/page_out. Here page_in means the paging server fetch a page from the disk and free the block in the hard drive, and page_out means the paging server finds an empty block on the disk, writes the page to the block and evict the page in memory.

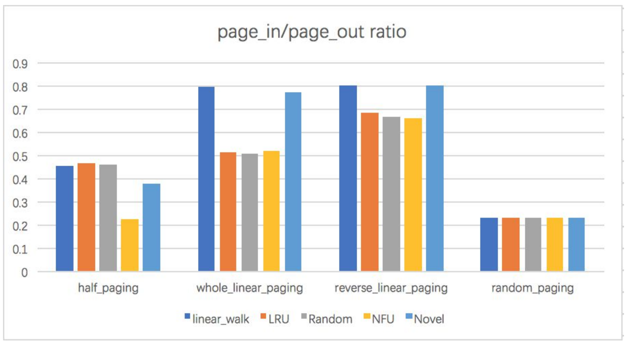
For a page selecting algorithm to be efficient, a page that is evicted should have rare chance to be fetched into memory again since the cost to read a page from hard drive is higher. Hence, a paging algorithm that performs well should have low page_in/page_out ratio.

Also, we record the time elapsed during handling page faults and use it to measure the speed of a page replacement algorithm. Here, linear probing and random replacement are set as a baseline for measuring performance due to their simpleness.

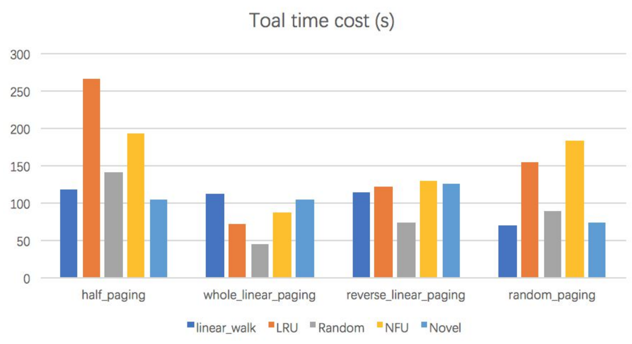
To test the extended paging system, we write four user programs that generate page faults. The four user programs perform differently in allocating and accessing page, as we have program that allocates whole page in memory and accesses half of it, allocates and accesses whole pages, allocates and accesses pages in reverse order and also accesses page randomly. For measuring the performance we record page_in/page_out ratio, time cost for handling page faults in each program and time cost per page faults.

Results

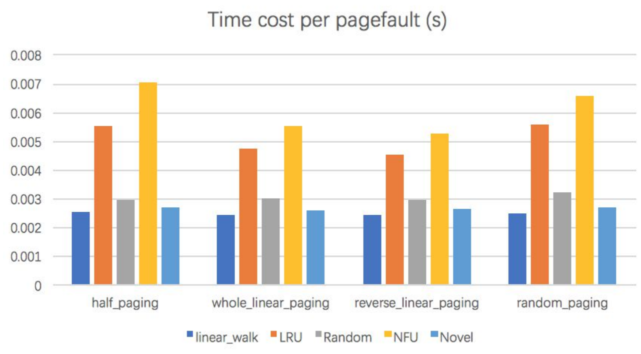
From the page_in/page_out ratio diagram below, we find that LRU, NFU and Random replacement have better performance over others. As we expected, linear probing has bad result since it simply evict the first page found. Our novel algorithm does not perform well too, possibly we did not implement the time span for page references well.



The time cost of different algorithms show that LRU and NFU has the most time cost on average, which meets our intuition since these two algorithm need to maintain a list or queue for page entries and update timestamps frequently. The other three methods have lower time cost in comparison.



We can see an obvious pattern in terms of time cost per page fault in the graph below. Again, LRU and NFU has the highest time cost for handling a page fault for the reason stated above. However, from the statistics we can see that the difference between time cost of page replacement algorithms is actually very little. Compared with the cost of reading a page from disk to memory, the additional time on handling page fault will not significantly affect the efficiency.



Conclusion

For this project, we have successfully extended the paging system of JOS to hard drive. What we have implemented include writing user programs triggering page faults for performance metrics, extend existing syscalls, implementing a paging server for reading and writing pages to hard drive, and also realizing different page replacement algorithms. We carried out experiments on paging algorithms and gained an overall understanding of existing algorithms such as LRU and NFU as well as their strengths and disadvantages.

Moreover, we proposed a paging algorithm which ends up in moderate overall performance. In both real world scenario and our experiments, LRU and its approximation NFU has been proven to best balance performance and speed, while Linear Probing and Random are too naive to perform well.

If we were given more time to work on this project, we will spend more time on the features that are not implemented perfectly. For example, when implementing LRU algorithm, we found it difficult to get the system in the kernel so we create a timestamp on our own, which may be not accurate as expected. Also, we noticed that our user program is not very stable as sometimes it will give very few page faults, which should be solved if time permits. In all, the project is interesting and worthwhile to work on and we have reinforced our knowledge about paging system through it.