Paging System, a final project for MIT 6.828, Fall 2012

Group 2 - members: Gurtej Kanwar (gurtej@mit.edu), Zixiao Wang (garywang@mit.edu), Jordan Moldow (jmoldow@mit.edu)

Introduction:

A commonly implemented solution to the problem of limited memory is paging. A paging system moves pages from the memory to permanent storage in order to free up more memory space. If this memory is ever needed again, the paging system moves the page back into memory. These two operations are termed paging out and in, respectively. Permanent storage is slower than memory, so a key component of a good paging system is choosing pages to page out such that they will rarely, if ever, need to be paged in again. Out of the all the projects, the paging system interested us because the fundamental concept was simple, but there was lots of room to make interesting improvements, such as the heuristic for choosing which page to page out.

Paging server:

We decided to abstract out writing pages to disk to a paging server (similar to the filesystem server). The paging server is a specially created environment which continuously loops 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 (explained in more detail later).

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.

Paging library:

Rather than doing our work in the kernel, we decided to use an exokernel-style library. This library provides the interface for “safe” allocation of pages (which would page out pages in order to make space if we hit memory limits) and provides a system for paging in (more details on this later). Fundamentally, the paging library exists to wrap several syscalls to make them either page out or page in pages as needed, and to handle page faults caused by accesses to paged out pages and resolve them by paging in the appropriate page. sys_page_alloc, sys_page_map, and sys_page_unmap need to be handled specially, since they deal directly with page mapping and allocating. The other syscalls only need a small amount of wrapping code to make sure addresses passed in aren't paged out (keeping in mind paging out should be transparent to user code, so the user code shouldn't have to check for itself whether the pages it passes into syscalls are paged in).

sys_page_alloc is designed to return -E_NO_MEM whenever the machine runs out of memory and cannot allocate more pages. In this situation, we want our paging library to try to page out pages to make space in memory. To do 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. To avoid hogging the processor, page_alloc yields after some number of failures before trying 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 touch 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. For every other syscall, we use a simple wrapper which “touches” (i.e. tries to read from) any virtual addresses used by the syscall. This is to make sure the page gets paged in before we send it to the syscall, otherwise it will fail the perm check.

The second component of the paging library is handling paging in when we try to access a page that has been paged out. A page that has been paged out is not present, so whenever any user code tries to access any of these addresses, it will page fault. In order to handle this, the paging library installs a user-mode page fault handler of its own which specifically checks for the paged out case and triggers paging in as needed. In order to avoid interfering with other page fault handlers, we modified the page fault handler code to allow setting multiple (up to a constant max) page fault handlers to be registered and be called until one returns 1, indicating the fault was handled.

Paging out calls a page choice function, sends an IPC to the paging server with the page, saves the returned index along with the virtual address, then unmaps the page from the current environment. If the page isn't shared, this will cause it to be freed. We currently don't handle shared pages due to complexity added by using the exokernel model, as explained later. On the flip side, paging in allocates a page at the virtual address we want to page in, retrieves the mapping index associated with the virtual address, then sends the IPC to the paging server which handles retrieving the page off of the disk.

In order to set up the mapping between virtual addresses in each environment and the mapping index for the paging server, we needed a struct that could map each page to some value. We noticed this was similar to the page directory that existed, but the page directory existed only readable in user space. Allowing the user code to write to the page directory in a controlled way would have required added some syscalls, and also would have been tricky, since we had only 1 PTE_AVAIL bit left to us – clearly not enough for a mapping index into the swap space. Our solution was to build a completely user-side data structure similar to the page directory. Our “mapping directory” was a similar two-level tree with the mapping directory page located at UMAPDIR, and the mapping tables dynamically allocated as needed. The top 20 bits of each MDE (mapping directory entry) pointed to the subsequent mapping table, and the top 20 bits of each MTE (mapping table entry) was the stored mapping index. The remaining 12 bits of each were reserved for flags, similarly to the paging directory. In the MTE only, we use all of the PTE_SYSCALL perm bits to store the corresponding perm bits of the actual page mapping, so that we can restore these when paging in. Because the mapping directory was located at a fixed location in each environment, we were able to then map it in when we were dealing with environments that were not the current one (for example in the page_map wrapper for the sys_page_map syscall, which takes in two arbitrary environments).

Page Choice, Metrics, and Shared Pages:

Our bare minimum project was developing a successful paging system, but the heart of our project was to come up with paging metrics and develop page choice functions which could be tested against those metrics to improve the page choice functionality as much as possible. In order to allow flexibility with this, we defined our page out system such that it used a function pointer, rather than a function definition, for getting the page choice. This meant that we could easily change one line to swap between page choice functions.

Before writing page choice functions, we had to decide how to rate their performance. There were two general categories of metrics which we cared about – *efficiency*: the page choice function shouldn't result a high rate of page ins vs page outs; and *fairness*: if a user program decided to allocate many pages, a smaller (memory-wise) program which later decides to allocate memory shouldn't be forced to constantly swap pages in and out just to get a small number of pages. Our metric for efficiency was simply the ratio of page ins to page outs – this was easily tracked on the paging server side, since it handled every page in and out.

One common feature all our page choice functions shared was that they did not choose shared pages. We currently don't handle shared pages because there's significant added complexity due to the exokernel style we're using. The paging library runs in user space, so it only has access to the current environment and its children. If there is a page shared with a non-child environment (for example due to an IPC), then we have no way of unmapping the page from that environment, and so can't successfully page out that page. Paging in the page provides similar obstacles. One potential solution to this would be to grant the paging server full privileges to use to sys_page_map on arbitrary environments, and then have the paging server deal with unmapping the pages. There is the still the complexity of building and maintaining a reverse-map that is user space accessible (we built a reverse-map for other reasons in kernel space).

Our initial page choice function was a simple environment-specific linear walk, in order to allow us to test the base system. This had the advantage of being easy to understand and debug, but was not very efficient in most cases, and was not fair at all, since the paging out was individually managed by each environment. Even if the environments were benign and willingly used the paging library, the library code only ever has the opportunity to page out in the case where an environment tries to allocate a page and fails. This allows environments to allocate many pages and hold them without being forced to give them up.

Once we had our base paging system in place, we focused on improving efficiency first (the easier of the two problems in the exokernel model). We looked up several existing paging algorithms in use – one of the commonly implemented ones is the Least Recently Used replacement algorithm. It is based on the simple idea of paging out the pages least recently used (and hopefully therefore least likely to be used). There were a few variants on the system – ultimately we selected one (based on the Linux kernel implementation, described at <http://www.linux-tutorial.info/modules.php?name=MContent&pageid=311>) in which we tracked a simple age value on each page. When a page is accessed, the hardware sets the PTE_A bit. On each clock interrupt, the kernel iterates through some pages (not all, since there are a lot of pages and this would take a long time), and rejuvenates pages that have PTE_A set (and clears the PTE_A bits), and ages pages that don't have it set. Since age is stored in PageInfo but PTE_A is stored in the page table, we maintain a reverse-map from each PageInfo object to all PTEs that point to the corresponding page, and check each of these PTEs to see if any of them have PTE_A set. The age field is exposed in the read-only PageInfo objects to user environments, so that user-space LRU replacement algorithms can choose older pages over newer pages. This system was significantly more efficient. It also partially handled the fairness issue, since an environment with a large number of pages would have a lower average age and would tend to page out more pages.

The data we collected for the efficiency metric (page ins / page outs) is shown below for the naïve linear probing system vs the LRU algorithm we implemented. The most important test program was the zigzag test program – a program that allocated all the memory and then only accessed the most recently used half of it. This simulated the locality of reference assumption that LRU attempts to exploit, and the results reflect this – LRU has a significantly lower page in rate, and is therefore much more efficient. The linear test programs set up the unrealistic situation in which the entire virtual address space was accessed, and therefore neither linear probing or LRU did very well on it. The random paging user test was implemented to act as a control – no page choice function should be able to do very well, since all pages are equally likely to be accessed next.

Figure 1: Paging efficiency metric

(page in/page out)	zigzag	linearpagein	reverselinearpagein	randompagein
Linear probing	14671/32070 = **0.46**	20317/25374 = 0.80	20727/25789 = 0.80	5301/22578 = 0.23
LRU	1474/18707 = **0.08**	19494/24541 = 0.79	23615/28710 = 0.82	5306/22583 = 0.23

Conclusion:

We successfully implemented a system to page memory to the hard drive and retrieve it on demand, allowing us to write user programs that use more memory than the amount of physical memory available on the computer. We accomplished this in exokernel style, allowing user programs and libraries to implement their own paging strategies and heuristics for choosing which pages to page out. We were partially successful in implementing and testing multiple heuristics for choosing pages to page out. We created two proof-of-concept heuristics, and tested them on various user test programs. We were unable to test fairness for multiple environments simultaneously allocating memory. Part of this difficulty was due to the exokernel model, which made it difficult to arbitrate multiple environments.

There are a few particular areas we would focus on for the future if we were to continue the project. Firstly, we would attempt to develop our fairness metric into a more quantitative measure. Our currently qualitative method did not give much specificity when attempting to improve. Secondly, we were unable to resolve issues with forking in low-memory situations, and would like to resolve those bugs. Thirdly, we occasionally ran into stack overflow bugs when attempting to page out, due to concurrency issues when multiple environments were running with lots of memory mapped – this situation occurred most frequently when attempting to run multiple user environments and was an obstacle to testing our fairness metric.