CS 377 - Operating Systems Lab

Design of Virtual Memory for the Pranali OS
Team A3

April 17, 2014

Team Members

Sagar Jha - 110100024
Raghav Gupta - 110100092
Guna Prasaad - 110050082
Yeshwanth C - 110050083
Vipul Venkataraman - 110050084
Mohit Gupta - 110050085
Introduction

In this project we focussed on the design and implementation of the virtual memory for Pranali. This is because the Pranali OS does not provide virtual memory and creates the entire address space of a process in the RAM. The various broad components we focused on implementing in our project are the following:

- Constituting the swap space of a program on the Sim_disk
- Allocating pages to processes
- Design of the page table :- Our group-specific requirement was to implement a multilevel page table
- Handling page faults by using an effective page replacement policy

Overview

A brief description of what we have implemented:

- Allocation of swap space to a process
- Loading of code of a process in its swap space
- Allocation of page frames to processes
- Non contiguous memory allocation
- Demand paging
- Multilevel page table (group-specific addon)
- Effective page replacement implementation
- Raising of page faults during operation of a process
Data Structures

Here we discuss the various new data structures, and changes to existing ones, that were needed for our project.

Changes to memory image mem_t

- The mem_t struct, which is the memory image of a process, now stores a pointer to a dir_table_t object instead of its old pages array. This constitutes the upper level of our page table.
- The number of pages of this process which are in memory, and the number of dir_page_t objects in memory, are maintained in variables num_pages_in_memory and num_dir_pages_in_memory respectively.
- A list of mem_page_queue_t pointers is also maintained to implement FIFO replacement policy for in-memory pages.
- Number of page faults incurred for this process.

Page table structures :- dir_table_t and dir_page_t

- Our page table is a two level page table with the higher level indexed on the 12 most significant bits of the virtual address, and the next level indexed on the next 8.
- Struct dir_table_t in turn contains:
  - A 4096-sized array of dir_page_t pointers
  - Where on disk the data of each of these dir_page_t is stored
  - The number of pages belonging to every dir_page_t that are currently in main memory.
- Struct dir_page_t stores a 256-sized array of actual page table entries.
- However, only a few dir_page_t objects out of the possible 4096 will actually be allocated (since the combinations of the most significant bits of the virtual address are sparsely used), which is where the multilevel page table implementation actually optimizes on storage.
- The page table is NOT stored completely in memory. Space is allocated for all created page table entries on the swap space portion of Sim_disk, which are sized 16B each.
- Thus, a bunch of 256 page table entries, all corresponding to one particular 12-MSB combination, constitute 4KB of data when stored on disk, equal to one page.
- This implies page table entries are stored alongside actual page data.
- More on page replacement policies to follow.

Page table entry :- mem_page_t

- The already available mem_page_t struct now doubles up as our page table entry, with slight modifications:
  - It no longer stores the tag or next fields. This is for storage requirement optimization.
  - The pointer to the page data doubles up as the physical address of the page, and is valid only for pages that are in memory and NULL for others.
  - It now stores the swap space location (basically a Sim_disk block) where the page’s data is stored.
  - The valid bit and dirty bit are integrated too.
  - Our implementation may contain mem_page_t objects for pages that are not even mapped (reason for this will be discussed later), hence the page table entry now includes a mapped bit, which is high when the page is actually mapped.
List of pages in memory: `-mem_page_queue_t`

- This is to maintain a list of pages that are in memory, for efficient implementation of FIFO page replacement policy
- This is a singly linked list of `mem_page_queue_t` pointers, with the earliest-fetched page at the head of the list for easy removal
- Stores the tag of the page, for identification
- Any newly-fetched page is placed at the end of the list

Interrupt list: `-mylist_struct` and `-node_struct`

- `mylist_struct` is a list of interrupts, storing a linked list of `node_struct` pointers
- `node_struct` stores the instruction number at which the interrupt is to be serviced, the context pointer of the relevant process, and a pointer to the next interrupt in the list

Global variables

- Maximum number of in-memory pages allowed for any process (`MAX_PAGE_IN_MEMORY`)  
- Disk block numbers of start and end of swap space
High Level Design Overview

Here we present the high level design overview of our different components. We have described them in more details afterwards.

Swap space management

We have a global swap space for all code and data, occupied by process data pages and page table entries on an FCFS basis. We assume a fixed total size of all of swap space. This will be a fixed part of the Sim_disk current implementation fixes this to the first half of Sim_disk).

Every page allocated via the existing mem_page_create is allotted 4KB on the swap space. Also, every required dir_page_t object, representing 256 page table entries totalling 4KB (same as the size of the data of a page), is allocated space on this swap space.

Wherever initialized, the data for every page and the page table entries for all allocated dir_page_t objects exists on swap space. So, when a page not in memory is needed, its data can be fetched from swap space. Same holds for page table entries which do not exist in memory.

Some finer points :-

- If the end of swap space is reached, the current implementation will terminate Pranali.
- Also, stray writes into swap space (by the read/write syscall implemented as part of assignment 8) are disallowed.

Demand paging

With the demand paging module in control, the complete address space of the process is not loaded into the physical memory when the process is started. It is rather loaded into the swap file of that process, by allocating space for all the required pages in the swap space, but only as and when the pages are needed.

The pages come into and go out of main memory by subsequent page faults. While mem_page_t entries exist for all pages which are in memory (and some more), the 4KB data space will be allocated on memory only for valid pages. Which pages will have space for their data allocated on memory will be decided by the page replacement policy

Non contiguous memory allocation and loading of code

Non contiguous memory allocation gives each process pages from the memory, and these do not have to be contiguous. A page request may be performed by any process for a data page or for storage of page table entries. The page tables of processes are also updated, whenever a new page is allocated to the process.

- A new dir_table_t may be created if needed i.e. if the first 12 bits of the new address have not been encountered before, in a request for allocation of a page
- If the relevant dir_table_t did not exist before, we create all the 256 dir_page_t objects for this dir_page_t.
- Why create all 256 page table entries? To exploit spatial locality of page requests.
- If we encounter a request for a page with the same index for the textit dir_table_t but a different on for the dir_page_t, just the relevant mem_page_t object has its mapped bit set.

Since page allocation requests can come in any order of address or process, the FCFS allocation model followed is hence non-contiguous
Page table lookup and replacement

Our implementation of the page table is a partly-in-memory multilevel one, with a page table per process. The page replacement policy is local FIFO i.e. a page will be replaced (if need be) by another page belonging to the same process, on a FIFO basis. At the same time, page table entries compete to be in memory. All are stored on swap space as well. However, the memory dedicated to page table entries is different from that dedicated to data pages i.e. separate limits have been set on their respective numbers in memory.

Data pages

\textit{mem\_page\_t} now carries a valid bit to indicate if the page is in main memory or not. Care is taken that the number of pages with a high valid bit does not exceed the maximum number allowed, per process; this number is stored in \textit{mem\_t}. If a page needs to be replaced to make room for another, its valid bit is set to zero and it is swapped out (written back to swap space) if its dirty bit is one i.e. its data has been modified since it was last fetched from swap space. Then the requested page is either created or its data fetched from swap space (if it already exists).

Page table entries

As discussed, a \textit{dir\_table\_t} object contains pointers to \textit{dir\_page\_t} objects, each representing one combination of the first 12 bits of the virtual address. As reasoned before, only a few of these will be used, saving space. However, we allocate space for each of the 256 \textit{mem\_page\_t} objects inside a \textit{dir\_page\_t}. We keep in memory only those \textit{dir\_page\_t} objects which contain a page which is in memory. Memory is not allocated for \textit{dir\_page\_t} objects which do not contain an in-memory page, and is deallocated for those who cease to have one.

To TLB or not to TLB?

In our case, for pages in memory, a successful page table lookup takes only two array accesses. Also, logically, TLB is a hardware component. Implementing it in software (which could be done by making every page table lookup perform a TLB lookup before) is contradictory to the design principles that led to the development of the TLB especially in our case where a reasonable speedup would not be achieved using a TLB.

Owing to these reasons, we did not implement a TLB, though this is a consideration worth investigating (part of possible future work).
Submodule Details and Functions Written

Here we discuss actual implementational aspects of the higher-level modules

Swap space allocation

In function `ld_load_exe`, the various sections (code, program headers, 8MB stack etc) of the executable are loaded. The addresses occupying these sections are mapped to pages (using function `mem_map`) which are allocated 4KB of swap space each, and `mem_page_t` entries are created for these. These are also inserted into the multilevel page table (discussed later)

As per this model, every page created has its space on the swap space. Also, the swap space is occupied in a linear order, but right now freeing some page from in between the swap space becomes difficult, hence the space allocated to a page remains on swap space even after it is freed. This is one thing we may seek to improve as an extension.

Initial loading of code in memory

All pages are allocated first on the swap space, with valid bit 0. This means that running the very first instruction of any process will cause a page fault.

Page table lookup

As discussed, our multilevel page table has two levels: `- dir_table_t` and `dir_page_t`. Their roles have been discussed.

The function `mem_page_get`, which is supposed to retrieve a page from the page table, will now look up the first 12 bits of the requested address in the `dir_table_t` array of its memory image `mem_t`: if a `dir_table_t` object has not been declared for this index, a NULL page is returned. Else, if the corresponding `dir_page_t` object (in the array inside `dir_table_t`), for the combination of the next 8 bits in the address as index, has mapped bit 1, we have our page/`mem_page_t` entry. Else we return NULL.

Page read/write

If the page being accessed is not even allocated yet (i.e. `mem_page_get` returns NULL), the page is created. After that,

- If the access is a read/execute
  - If the page is valid, and `page→data` is not null, return the requested data else return a string of 0s
  - Else if swap space has not been allocated for the block, return a string of 0s, else it is a page fault

- If the access is a write/initialization
  - If the page is valid, perform the write and set the dirty bit of the page to 1.
  - Else call the page fault handler to bring the requested page in memory (at the expense of another) and then perform the write.

Page fault handler

Our function `mem_page_allocate` functions as the page fault handler:

- It first checks if allocating another page in main memory is allowed, if so, then 4KB is allocated for the data of the process
- Else, it is a genuine page fault. In this case,
- It looks at the head of the `in_memory_queue` associated with the memory image `mem_t`, takes the tag at the head of the queue, finds that page and sets its valid bit to 0, and writes it back to disk if its dirty bit was 1.
- The 4KB memory previously used by the swapped out page is reused for the swapped in page, whose valid bit is now set.

Whichever new page is fetched is appended to the end of the `in_memory_queue` of the `mem_t`, since it is the most recently fetched page.

**Page table storage on disk**

Whenever a 12-MSB combination is first encountered, we create a `dir_table_t` object using function `mem_dir_allocate`, allocate it 4KB of swap space, and create the 256 `mem_page_t` objects. All of these may not be mapped, however, so the mapped bit for each of these is set accordingly.

We deallocate memory for a `dir_table_t` only when the number of in-memory pages contained inside this `dir_table_t` becomes zero (this count is maintained as part of the `dir_table_t` fields).

**Process blocking**

We block the running process in case of any read/write to/from `Sim_disk`. This can be due to a page fault or an explicit read/write syscall.

The penalty the process incurs is calculated as the number of instructions the process would be blocked for. Also, even if all processes are blocked, the processes are not fast-tracked into scheduling.
Incomplete Work/Future Additions

Dynamic allocation of swap space

Currently, for any page creation request, we allocate 4KB on the swap space and this space is not freed even if the page is freed from memory. This was done largely for ease of implementation, though a dynamic allocation scheme where there can be free areas in the middle of allotted space is a worthwhile update.

Global replacement policy

Currently our policy for page replacement is local-only i.e. the requested page can evict another page of the same process. However, local replacement MAY not always be successful. A better idea would be to maintain a global page table and implement a replacement policy that is global i.e. a page can be replaced by another page from any other process.

TLB

Even with our current implementation where it takes two array accesses for a successful page table hit, the speed up produced by a TLB could be negligible. Still, it could possible by a worthwhile exercise to explore the usefulness of a software-based TLB for our application.

Loading code in memory

Current implementation initializes no page on memory, and the very first instruction of the program will cause a page fault. It would take effort to study about contemporary OSes and discover how initial pages to be loaded into memory at the start are chosen.

Working set

Currently the limit on the number of pages of a particular process that can be in memory (which is MAX_PAGE_IN_MEMORY) is set to a global, common to all processes. One possible extension would be to implement a working set that would enable us to set this limit dynamically, and uniquely for each process based on each process’ memory requirement.

Comments

- Certain syscalls do not work (read, write etc)
- Overall, a nice experience, definitely one to learn loads from