OS Project: Theme A

IMPLEMENTING PAGE SWAP-IN/OUT IN PRANALI OS

April 19, 2014

Group A2
Deepali 11D170020
Sushant 110050009
Nilesh 110050007
Sanchit 110050035
Vipul 110050034
Rajlaxmi 110050087
Department of Computer Science and Engineering
IIT Bombay
INDIA

Contents

1 Overview

The current Pranali OS creates the entire address space of a process in host RAM, i.e., it does not provide virtual memory. The project aims at designing and implementing virtual memory for pranali which includes :-

- Allocation of swap space to process
- Loading pages in swap space
- Initial Loading of pages in pranali RAM
- Maintain page occupancy list and page free list
- Handling Page Faults
- Maintaining page replacement using combination of local and global replacement policy
- Free Pages when process exits
- Block the process during page fault handling
- Using TLB to improve the page lookup from RAM.
- Implement swap-in/swap-out of processe pages from RAM during IO operations

To implement these we implemented interrupt for swap-page-in which is introduced during swap-page-out. More information regarding this in the later sections.

2 Data Structures format of main data structures

Following new data structures or modications to existing data structures was used to implement Virtual Memory:

2.1 mem_t

This was originally in Pranali and was central for making any memory access by the process. This included a hashtable of pages of the process. It also has the host mapping made with the process if any. The changes we made to it include the following

- **pages** Hash table that stores the pointers to the pages in the swap space
- **ram_pages** Hash table that stores the pointers to the pages in RAM
- **total_faults** Total number of page faults the process has encountered
- **current_instruction_faults** Number of page faults encountered in the execution of the current instruction
- **pages_in_ram** Count of the pages of this process in the RAM
- **pages_swapped_out** Count of the pages of this process which have been swapped out
- **swapped_pages_addresses** Stores addresses of the pages that have been swapped out during IO

2.2 mem_page_t

Pranali has implemented the process data space in form of pages. We now use these pages both as pages in RAM and pages in swap space. Hence it includes fields for both. The new fields added are the following

- **dirty** It is used for RAM page. It indictaes that the page needs to be written back to the swap space
- **free_flag** It is used for RAM page. It indicates that this space in the RAM is free and hence can be used to insert a page
- **bytes_in_use** This is used for swap page. It indicates the bytes being used in the page

2.3 swap_mem_t

This is our implementation of Swap Space manager. We maintain a free list and an occupancy list. We store the free list's head and tail pointers and occupied list's head and tail pointers

2.4 ram_mem_t

This is our implementation of Pranali OS RAM. It contains the pointer to the first mem_page_t in the RAM.

2.5 kernel_t

In the current implementation of kernel_t, we have added the following functions

- **loading_in_progress** Set when loading of a process is taking place
- **instruction_no** Instruction number being executed in the current process
- **current_track** Current track being accessed
- **interrupt_list_head , interrupt_list_tail** Head and tail of the list of interrupts scheduled
- **ram** Points to the RAM

2.6 tlb_entry

This is the basic tlb entry which is used for faster access of pages. It includes the following entities:

- **valid** Checks whether the entry is valid.
- **context** The context of the process
- **ram_page** The pointer to RAM Page.

3 Functions

In general, most of the functions were changed to adopt to the Virtual memory implementation of Pranali. In addition to altering the existing functions quite a few new ones were also written. Following are the functions implemented in Pranali

3.1 Swap Space related functions

- **swap_initialize()** Initialises the swap space, allot space to every page of the process in the SimDisk
- **get_new_swap_page()** Returns a new swap page, called when the process is initially loaded onto the swap space
- **free_a_swap_page(struct mem_page_t *, struct mem_t *)** Frees a page from the swap space when the process is completed/swapped out
- **read_swap_page(struct mem_page_t *)** Reads a page in the swap space

- **swap_write_back_page(struct mem_t *, struct mem_page_t* ram_page, uint32_t)** Writes back a page in the memory to the swap space at a specified address
- **swap_write_bytes(struct mem_t, uint32_t, uint32_t,void *)** Writes size bytes from the buffer at a specified address in the swap space
- **add_occupied_page(struct mem_page_t*)** This is used by the Swap space manager to add a page to the occupied list
- **swap_mem_page_create(struct mem_t, uint32_t, int)** Creates a new page in the swap space
- **swap_out_process)** Swaps out all the pages of a process, if the page being replaced is dirty, it is written back to SimDisk
- **open_swap_disk()** Opens the SimDisk file in global variable swap_fd

3.2 RAM related functions

- **get_free_ram_page()** Returns the first free page found on RAM
- **ram_get_new_page(struct mem_t *)** Returns a new page on the RAM
- **get_page_to_be_replaced(struct mem_t *)** Returns the page to be replaced in RAM based on the combination of Local amd Global Page replacement policy
- **page_fault_routine(struct mem_t *, uint32_t)** Handles page faults
- **addInterruptForProcess(struct mem_t*)** Adds an interrupt set to the current instruction number + [100*(Number of page faults in the current instruction)]

3.3 mem_t related functions

- **mem_page_get(struct mem_t *, uint32_t)** Returns the page from RAM in case it is there, else page fault routine is called
- **swap_mem_page_get(struct mem_t *, uint32_t)** Returns the page from swap space at the specified address
- **mem_page_get_next(struct mem_t *, uint32_t)** Returns the next page from the swap space at the specified address
- **mem_map(struct mem_t *, uint32_t addr, int size, enum mem_access_enum)** Allocates space in the swap space when the process is initialised and loads them into RAM if needed, during execution

3.4 TLB Related functions

- **insert_tlb(struct mem_t* mem, struct mem_page_t* ram_page)** Inserts into TLB, entry corresponding to the RAM page.
- **find_tlb(struct mem_t* mem, uint32_t addr)** Finds in TLB whether the page corresponding to the address `addr` exists.
- **remove_tlb(struct mem_t* mem, uint32_t addr)** Removes the page corresponding to address `addr` from TLB.

4 General Code flow

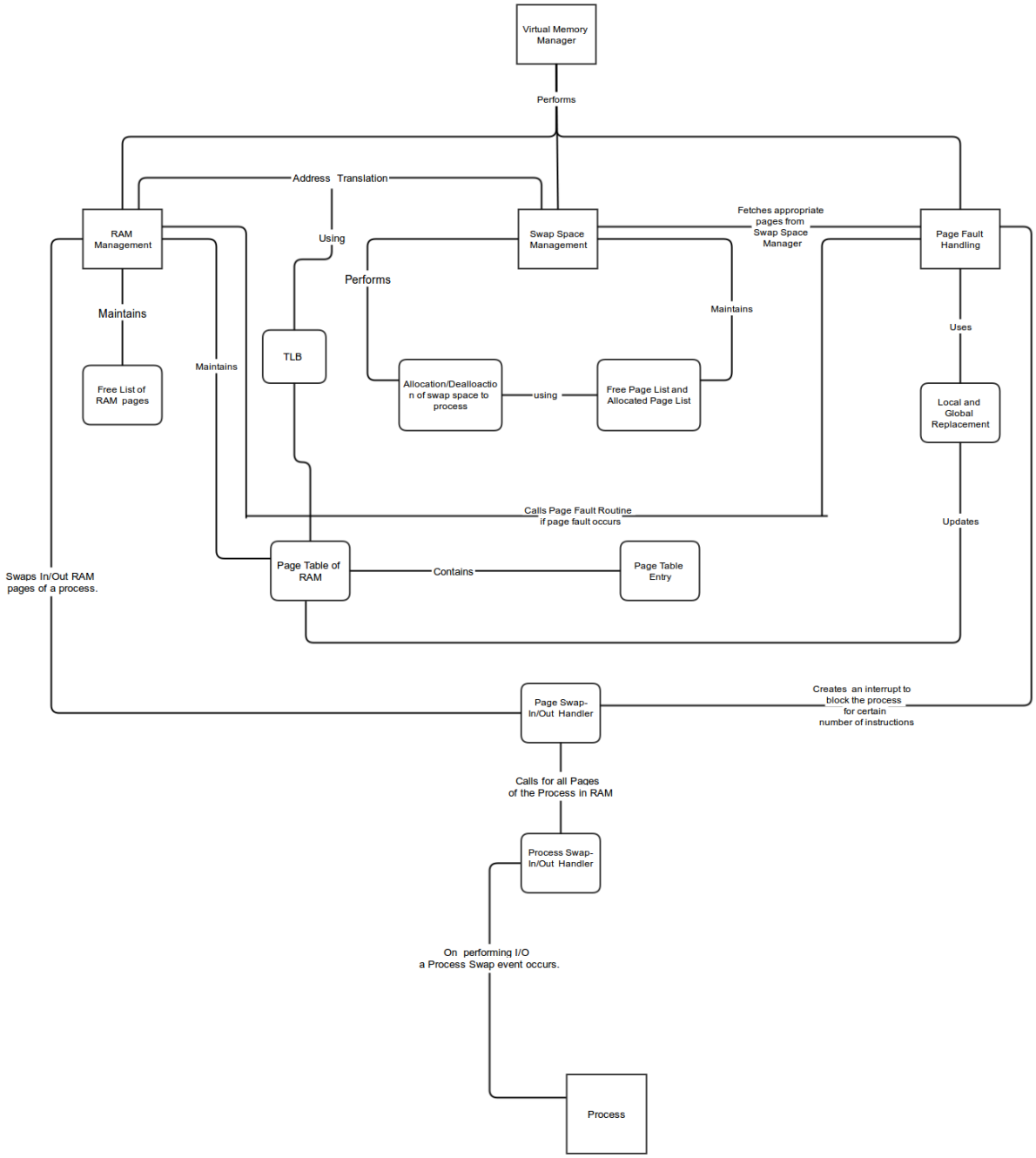
```
Initialise RAM
Initialize Swap Space
Load process
        Create process ctx
        Create process mem_t
Loads executable
  for all elf sections
             Create pages in Swap space
                Load section onto the Pranali RAM
                 if TLB fault
                     read from RAM
                        if RAM fault
                     Replace the page in RAM and TLB


Loop until the processes are finished
  Handle interrupts if any
  Execute Instruction
  Read instruction from TLB
                 if TLB fault
                     read from RAM
                        if RAM fault
                     Replace the page in RAM and TLB
                Swap out process in case of IO
                 for all pages
                     if dirty
                     write back
  Add interrupt in case of Page Fault
```

5 RAM and Swap space configuration

- Page size 4KB
- Size of the swap space 240MB (60000 pages)
- Size of the RAM 4MB (1000 pages)
- Space on RAM per process 40 KB (10 Pages)

6 High Level Design Overview

To make a complete module, we implemented a certain number of sub modules and linked them together, to avoid cluttering by making a single submodule. The following diagram shows a brief interlinking of the various submodules.



7 Submodule Details

7.1 RAM Management

RAM is implemented as a linked list of constant number of pages which are present in the memory of Pranali. The memory for the RAM is only initially allocated and never again. The RAM has structures to give information of the occupancy of the page, the dirty bit of page. Every process has an upper bound on the constant number of RAM Pages it can use.

7.2 SWAP Management

SWAP Memory Manager manages the swap disk as a structure of free lists and occupied lists. Every time a new swap page is demanded from the disk, the page from the head of the free list is given. The whole of the swap disk is divided into two sections, one is the classical swap area and the other part is a general purpose disk. The functionality provided here is to write to certain blocks and tracks on the disk.

7.3 PAGE Replacement

Page Replacement policy on a broad scale, has two implementations

- **Local Page Replacement Policy** This is the first preferred technique for replacement. In case all the RAM pages associated with the process are occupied, a randomly chosen page associated with the process is replaced.
- **Global Page Replacement Policy** This is the second preferred technique for replacement. This comes into picture when the current process has no pages in RAM. A random page associated with the process with the highest number of pages in the RAM is replaced.

7.4 Interrupt Handling

- **Process Wake Up Interrupt** This is the interrupt corresponding to the page fault. The interrupt is set to bring the process from suspended list to running list.
- **Swap In Interrupt** This is the interrupt corresponding to process swap in. All the pages which were earlier swapped out (due to I/O) are swapped in one by one. In case the RAM doesn't have enough space while inserting, then the pages are replaced locally. In the case when RAM had no space when the first page was being swapped in, then global replacement policy is used.

7.5 Page Fault

On a page fault, the current process is moved into suspended list. An event(interrupt) is inserted at instruction count corresponding to the the number of instructions after which

the process is to be brought back to running state. This instruction count is calculated by counting the number of page faults which occurred in the current instruction. The correct page is then found in the swap space. Finally this page is inserted back into the RAM using appropriate replacement policy.

7.6 Page Table Implementation

The memory object of the process contains the page table. Page table contains the page table entries (PTE) for all the pages of the process.

7.7 TLB Implementation

TLB is fully associative. Even without TLB, the virtual memory system works. However, implementing TLB facilitates faster address translation by avoiding page table lookup. It is implemented to facilitate faster address translation. It is global and contains only a few page table entries along with their process IDs.

8 Issues

8.1 Using predefined Linked List

Quite a few issues turned up, while using the predefined linked list. Importantly inserting into list head/tail and removing from list head/tail. We debugged quite a lot of errors occuring due to this.

8.2 Handling page faults

We are handling page faults per instruction and not per page. So, multiple page faults can occur in the single instruction. This is handled accordingly. Doing it per page was quite difficult as we would have to change inside isa etal. Also the page faults are handled after instruction occurs.is executed.

9 Incomplete Work/Future Additions

There is a lot of scope for improvement in the current implementation in terms of both performance and design. Due to time constraints, we could not code all of them. Following are some of the additions that we would have liked to do given more time.

9.1 Replacement Policy

Currently, our version uses random local replacement policy and if need be a global replacement policy(meaning when there are no local pages of the current process to replace). We would like to implement LRU(Least Recently Used), both local and global replacement policies and give the user the choice to choose between the two.

9.2 Storing some part of Page Table in RAM

Currently, the information about where the process pages are present in the swap space is stored on the physical memory. This is a bit inefficient in terms of space. A possible future add-on is to store some parts of it on the disk.

9.3 Shared Memory

Currently our version does not support shared pages between processes. Designing this would be tricky as the page may be updated by two processes simultaneously. It would involve making policy decisions about dealing with this. For example, Pranali might decide to allow only one process to access the page at a time or it might adopt a copy on write policy. It will also involve handling all protection issues pertaining to sharing correctly.

9.4 Inverted Page Table

We would like to code a more efficient Page Table like an inverted Page Table or a multi level Page Table.