Implementing Virtual Memory in Pranali OS

Anmol	Astha	Hardik	Kshitij	Nisheeth	Rishiraj
110050020	110050018	110050029	110050016	110050027	110040015

March 28, 2014

1 Introduction

This report outlines how the implementation will be done throughout the project.

2 Swap Space

The *swap space* is used to implement virtual memory. The idea is that in view of limited RAM sizes, there is a severe shortage of memory for processes. To counter this, swap space is given on the disk, such that each process has an illusion of a large memory. Inactive pages are repeatedly 'swapped' from RAM into this space while blocked processes are brought in when space gets available.

Static swap space allocation to a process is not an ideal solution, since that could lead to memory access problems when it asks for more memory than is allocated to it. On the other hand, barring future knowledge of memory requirements of processes, this can't really be solved quite efficiently.

Since this project deals with simulation of virtual memory on the *Pranali* OS, we would choose to allocate a constant swap space on the *Sim_disk* to each process. This can be handled based on the number of processes that are going to be simulated, based on the initial configuration file(ignoring child process creations, ofcourse!).

3 Page Table

The *page table* is an auxiliary data structure that helps identify which pages are in the main memory and in which page frames they are located in. It is actually a mapping from the 'virtual' address to an actual physical address on the disk or the RAM. Such a page table is required for each process to check where its pages are allocated memory.

The problem with keeping one page table for each process is that the memory requirement for storing page tables themselves exceeds the current disk capacities. Even a conservative estimate of current architecture specifications yields total storage size in hundreds of terabytes. An alternative(and highly space-conserving) strategy is to use *inverted page tables.*

Here, instead of the above mapping, a mapping is stored from physical page addresses to virtual page addresses. For this, a single linear shared array suffices. Physical page numbers are not stored, since the index in the table corresponds to them. Process IDs are obviously needed to be stored, which would be used when a linear search is done to get the physical address mapping of a virtual address. While this solves the space problem by a huge margin, it creates a new problem in time, since searching through the whole array for each memory access is expensive.

To mitigate that, we will use *hashed inverted page tables.* Here, an additional level, in the form of a hash table is added that maps process IDs and virtual page numebrs to page table entries. Chaining will need to be done to resolve collisions. Since hash tables enable constant search time, it will lead to shorter time.

4 Page Replacement Strategy

When a page needs to be swapped in, a proper page replacement startegy needs to be used. The most popular such strategy is the Least Recently Used, which will send the page that has the lowest timestamp, out of the RAM.

The implementation is using linked lists and a hash table. A list will be kept that will have pages sorted in ascending order of timestamps, while a hash table will keep a pointer to a page's location in that list. When a page that was in the list is accessed, it will be entered at the end of the list, while removing the old entry with pointer redirection. This will take up quite a small fraction of a page size, and cause just a small time overhead, which will not matter much in simulation.

5 Advanced Features

There is a virtual memory manager which maintains a free frames list and tries to keep a few page frames in this list at all times.

The virtual memory manager will have 2 threads. One thread called 'free frames manager' is activated by the virtual memory manager when the number of free page frames drops below threshold which is predefined. The 'free frames manager' scans pages to identify few pages which can be freed, and adds the page frames occupid by these pages to the free page frames list and if the page frame is dirty it will mark the page frame dirty in the free page frames list. The free page frames manager puts itself to sleep whine the number of page frames exceeds another threshold of the virtual memory manager.

Another thread called 'page IO handler' performs page out operations on dirty page frmaes in the free page frames list, resets the dirty bit of a page frame on page-out operation.

6 Work Division

The project has been divided into the following sections (each assigned to one team member):

- Implementation of data structures (hash tables, linked lists, fixed arrays) required for page tables, swap space etc. *[Hardik Kothari]*
- Implementation of *Inverted Page Tables [Kshitij Singh]*
- Implementation of *Swap Space* as part of the *Sim_Disk* and initial allocation of swap space to processes. *[Astha Agarwal]*
- Implementation of LRU page replacement policy *[Nisheeth Lahoti]*
- OS thread for free space management *[Anmol Garg]*
- Writing the function for converting virtual address to physical address, and raising page faults if required *[Rishiraj Singh]*

7 Current Status

The following things have been done as of today:

- Read and understood the internals of the original *Pranali OS* code
- Finalized the design and decided the work distribution
- Coded the custom data structures needed for the VM implementation
- Coded considerable part of inverted page table
- Started implementation of replacement policy