**Project Report (Group a\_5)**

**Theme A : Virtual Memory**

**Overview:**

Objectives Completed:

* Non-Contiguous Memory Allocation
* Swap File
* Demand Paging
* LRU Page Replacement

**Data Structures:**

VM Manager:

This is the main VM manager which is used to handle page faults and TLB misses. It maintains a table of page entries.

* VM Manager Page Entry : Each entry of VM manager is of this type
* Number of Page Frames allocated to VM Manager

VM Manager Page Entry:

Each entry of VM manager is of this type. All this information comes into play when we have to replace a page from the VM manager space.

* Physical Address
* Virtual Address
* Thread ID
* Dirty Bit
* Valid Bit
* Reference Bit
* is\_free : Boolean to check whether page is free/allocated
* Address Space : Stores pointer of the addrspace of the thread, whose id = Thread ID

VM Metrics:

This data structure is used to observer the performance of the VM system.

* Number of TLB misses
* Number of TLB misses where page was found in memory
* Number of Page Faults
* Number of Page Faults where VM Manager found a free page
* Number of Page Faults where LRU page replacement was used

Page Table:

This is the page table of the process. It is included in the address space of the process. It contains

* Number of pages in the address space of the process
* Array of the Page Table Entry

Page Table Entry:

This represents each entry of the page table of each process. It is used to give the virtual address to physical address translation.

* Virtual Bit
* Reference Bit
* Dirty Bit
* Virtual Address
* Physical Address

Swap File:

We are maintaining swap file for each user program. This struct represent a logical view of a swap file of a process

* File name: swap file name for the process
* Number of pages that would be stored in the swap file
* Array of Struct Swap Page

Swap Page:

It is a mapping of the pages present in swap file to their offset (i.e. The location of that page in the swap file). For given user virtual address it tells the offset of the page containing that virtual address. This offset is the index of the this swap page in array of Swap Page in Swap File

* Virtual Address: Virtual address of the page
* Size: length of content in that page. else is zero substituted

address space:

It represents the address space of the process. It contains the swap file and page table of the process.

**High Level Design Overview:**

We have implemented 4 basic modules performed by the virtual memory manager

1. Non-Contiguous Memory Allocation

* Non-Contiguous Memory Allocation gives each process non-contiguous pages from the memory
* It performs allocation of single pages to a process, whenever demand paging module calls for it
* It keeps the TLB updated with the latest pages
* It also updates the page table of the process, whenever a new page is allocated to the process

1. Swap File Management
   * Swap File Management module handles page traffic
   * It performs swap-in and swap-out actions on the pages of the process
   * swap-in module are called by on-demand paging module when to bring in the page into the physical memory
   * swap-out module are also used by Page replacement module when there is no free page frame in physical memory and a new page is to bought in. It replaces one of the page using page replacement technique and swap-out that page
2. Demand Paging
   * The main task which demand paging module handles is that it does not load the complete address space of the process into the physical memory when the process is started.
   * It loads the address space into the swap file of that process.
   * The pages come into the memory by subsequent page faults.
3. LRU Page Replacement
   * This page replacement policy takes the least recently used page from the memory, and replaces it with the page to be loaded into memory.
   * It keeps a reference bit in the virtual manager page entry, to implement this policy

**Submodule Details:**

Well, before we go into the details of each function let us understand the system flow when a user program is executed and memory structure of the os161.

When a program starts running, the following things happen

* First, a thread is created using the thread\_fork function
* Then, the runprogram function is called.
* It opens the executable file, using vfs\_open.
* Now, the address space of the process is set up, i.e. the page table and stack base of the thread are set up. This is done using the as\_create function.
* Then, as\_activate is called to activate the current address space.
* After this, load\_elf function is called which loads the code area and the data area at the entrypoint, using as\_define\_region and as\_define\_stack.
* Now, the executable file is closed, using vfs\_close function.
* Then, as\_define\_stack is called which loads the stack at the stackptr.
* Finally, enter\_new\_process is called which begins the execution of the process.

Following are the major breakthroughs in understanding the memory structure of os161 which helped in writing swap modules.

* From the 4 divisions of the mips memory, first division is targeted to be used by user program as stack-heap area. Second division is kernel space area.
* calling kmalloc and defining any object in kern code gives them space from kernel space area.
* physical pages or physical memory: There is no as such physical memory in os161. The page of physical memory are stored in kernel space area. So writing anything in physical memory, you end up writing in kernel space whose address is obtain by applying macro PADDR\_TO\_KVADDR on physical address.
* Read/Write from file: To read/write from file one uses vfs functions. We need to initialize uio and iovec structs for that. iovec base represent the base address from where to read or write. If this base address is kernel space address ie.8000.... then use UIO\_SYSSPACE flag and if it is user space address then use UIO\_USERSPACE flag.
* VOP\_READ and VOP\_WRITE functions uses uiomove function.

Now, let us explain the functions that we have implemented.

vm\_bootstrap:

* This function initialises the VM manager of the system.
* It steals some pages from the physical memory and sets up the virtual memory pages.

vm\_fault:

* This function is called whenever there is a TLB miss.
* It calculates the page containing the virtual address, which raised the TLB miss.
* Now, the page table of the process is checked to find the physical address where the page actually exists.
* If the valid bit is true, then that physical address is mapped to that virtual address in the TLB.
* If the valid bit is false, then this function calls the getpage(1) function. This function returns the physical address of 1 page to the function. Now, the vm\_fault function writes this mapping to the TLB.
* It also updates the page table of the process so as to change the physical address corresponding to the virtual address which raised vm\_fault.
* It also updates the virtual manager page entry maintained by the VM manager.
* The dirty bit is set in this function by the faulttype with which this function is called. If faulttype is VM\_FAULT\_WRITE, then dirty bit is set to true.

getpage:

* This functions gives the control to the VM manager.
* VM manager searches through its entries to find a page
* If a free page is found (is\_free bool entry in the page entry), then that page is returned.
* If no free page is found, then the reference bit of each page is checked.
* If some page has reference bit as false, then the dirty bit is checked. If the dirty bit is true, then that page is written to swap file.
* If all pages have reference bit as true, then the first page in the VM manager is replaced. This is a random choice, as we have no way to find the least recently used page.

timerclock:

* In this function, a counter is maintained to check if a particular time interval has passed. If yes, then the reference bit of all pages is set to false, else the counter is incremented.

load\_segment:

* This function is called when we start reading segments from the executable. To perform on-demand paging complete data and code area read from prog executable is being wrote to swap file. Segments are read page by page from program file to a buffer in kernel space and then wrote to swap file at their appropriate offsets.
* Now catch here is that there can be pages in prog executable which are to be given memory but do not contain anything initially. ie memsize > filesize. so to handle it we fill the swap page with zeros.
* Major problem while writing this function was that we do not know what base addresses to give to iovec object to read/write. This got clear when we understood difference in implementation of user space and kernel space.

read\_page\_from\_swap:

* arguments: physical address , virtual address of user program, and address space of process
* given the physical address of page we apply PADDR\_TO\_KVADDR macro to get kernel address of the physical memory page and write there. Data is read from swap file from offset determine by virtual address through Swap File object present in address space.

write\_page\_to\_swap:

* same technique is used as in above function but conversely.

as\_create:

* Create the address space of the process. It initialises page table entries and swap file entries.

as\_copy:

* It copies the address space from the old pointer into the new pointer.

as\_destroy:

* print the statistics regarding use of VM
* frees the kernel space used by address space structure.

as\_activate:

* invalidate the tlb registers. Its kind of flushing tlb before loading new process.

as\_define\_region:

* This is main function which sets up the page table entries with virtual address and swap file structure with offsets.
* This function is called each time for each segments in prog executable.
* This increase the size of page table each time by the # of pages in the segment and insert for each page the virtual address it points to. The physical address of this virtual address is set up in vm\_fault function when a physical page is allocated to it.
* In swap file, for each index i we store virtual address and size of the page. The index is the offset of the page having that virtual address.

as\_prepare\_load:

* This function initialises the page table entries of the page table of the process.

as\_complete\_load:

* Does nothing.

as\_define\_stack:

* This function sets up the stack of the process.
* It adds the entries for the stack to the page table.
* It increases the number of pages in the page table by the number of pages required for the stack, i.e. 12.
* It also sets up the mapping from offset to virtual address in the swap file.
* It also initialises the data in the stack area of the swap file, with zeros, using the bzero function.

**Incomplete Work / Future Additions:**

* We were thinking about implementing 2-level page table, for faster translation from virtual address to physical address.
* We were thinking of implementing free\_kpages just like alloc\_kpages, so that once a process completes, the address space present in the kernel area of physical memory is also freed.

**Comments:**

* System calls have not been implemented. Unknown syscall error on most testbin files.
* Overall, a good project, and a good experience.