PROJECT-2 VIRTUAL MEMORY IMPLEMENTATION IN PINTOS

Creator:

KEYUR JOSHI - keyurjos@buffalo.edu

Preliminaries:

Use the 'src' folder after extracting the tar
A copy of this document is enclosed in 'vm' folder

References:

1.
http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

2.
http://inux.die.net/man/1/readelf

3.
http://flint.cs.yale.edu/cs422/doc/ELF_Format.pdf

Paging Management:

This project implement basic paging functionality in pintos. It supports on demand paging, swapping, and stack growth.

Data Structures-

SUPPLEMENTARY PAGE ENTRY

```
 struct page
 {
   void *addr;              /* User virtual address. */
   bool writable;           /* Is writable; */
   struct frame *frame;     /* Page frame. */
   struct pageinfo *pinfo;  /* Additional info */
   struct hash_elem elem;
   enum pgtype pgtp;        /* File Location */
   size_t idx_;             /* Swap Start Sector Index */
 };
```

```
enum location  //Location of page
{
PHYMEM,
SWAP,
DISK
};



enum pgtype   //Tells whether page is stackpage or a file
{
STACKPG,
FIL
};

struct pageinfo  //Additional pageinfo for files
{
struct file *fname;
size_t offset_;
size_t readbytes;
enum location loc;
};
```

This data structure defines the supplementary page table entry. The field **'addr'** refers to the page address,**'pgtp'** tells whether the page is a data page or stack page. The nested struct **'pageinfo'** gives additional information about the page like the file to which it belongs the offset in file and number of bytes in page. It also contains enum **'loc'** that gives current location of the page.It telss whether page is in disk,swap or RAM.

```
struct list *lst;
```

FRAME TABLE ENTRY

```
struct frame
{
   void *base;                /* Kernel virtual base address. */
   void *paddr;               /* Mapped process page, if any. */
   struct page *pageptr;
   struct thread *t;
   struct lock frlock;
   struct list_elem el;


   /* ...............          other struct members as necessary */
};
```

The above data structure represents an entry in frame table. It stores the frame address , corresponding page address ,pointer to page and list element. List has been used here to implement frame eviction when full.

Algorithms-

Whenever a page fault occurs the **page_in()** function is called which locates the entry corresponding to the fault address(page) in the supplementary page table .If the page is located on disk/swap in the file then **do_page_in()** is called to fetch it into the RAM.
It then allocates a frame to the page and makes an entry in the page table the base address of which is stored in MMU.

<u>Paging and Disk Access & Synchronization:</u>

A page fault in user VM cause the **page_in()** to be called where the corresponding supplementary page table entry is searched. If no entry if found **page_for_addr()** is called if the address is below PHYS_BASE (3 GB) but not more than 1MB(STACK_SIZE) below PHYS_BASE the requested is treated as stack and page table entry is made.

When a frame is needed but none is free in that case **frame_evict()** is called. The eviction algorithm scans the access bit of each frame entry in frame table(**list lst**) and sets it to 0 if 1.If it encounters a bit that is 0 then that frame is evicted. The **page_accessed_recently()** function uses .If all frames have their bit set to 1 then algorithm runs once more from the beginning of list.

In order to avoid inconsistencies and race conditions during eviction the frame entry is locked for use by using the **lock_acquire()** function .The function then removes the page table entry for the frame and also resets the used pointers to **NULL**. Then lock is released. Another lock **(llock)** is used to lock the list to remove the element and is shared with code where frame entry is inserted into the list. The entry element is then freed. A new frame entry is created and added to the list with frame address**(frame->base)** of the deleted entry and page data is cleared.

When eviction occurs the corresponding page is moved to swap or to disk if the page belongs to file and swap is full. Also a change is made to (**p->pinfo->loc**) to indicate that page has been moved out of memory to swap or disk.

In order to implement swapping a swap device has been assumed to be used. This is treated as a block device with sector size of 512 bytes. The free space is tracked using a bitmap where a true value indicates occupied sector and false represents empty sector.

The **bitmap_scan_and_flip()** function returns the index of first consecutive free blocks and inverts them. Just as in case of frames here also another lock called scan lock is used before changing the bitmap and is released afterwards. This prevents another process from writing at the same location since the process that gets access first will change it and then the next one will get the updated bitmap. This will prevent race conditions while writing to the disk. The **idx** value stores the starting sector of the page location on swap.

Invalid addresses-

When page fault occurs the **page_fault()** handler in 'exception.c' is executed. Here a check is performed to see if the page belongs to User Virtual Memory else kill process.

<u>Design :</u>

This design provides coding simplicity and modularity. It effectively implements the required functions. Use of list makes it easy to scan frame table for recently accessed frames taking **O(n)** time. Hash table has been used to search for entries in supplementary page table since there are larger entries to process and use of hash table implies O(1) search time.