CS342
Design Document
Project 3: Vritual Memory

Anil Kag
10010111
a.kag@iitg.ernet.in

0ctober 28, 2012

1 Data Structures

1.1 Struct Thread

```
 struct thread
{
   /* Supplementary Page Table. */
   struct hash supplement_pt;
   /* Lock for accessing supplementary page table. */
   struct lock supplement_lock;
   /* Till this the user stack can be accessed without growing. */
   uint8_t *user_stack_limit;
   /* Number of stack pages which can be allocated (Limit 64 pages).*/
   int num_stack_pages_left;
 };
```

1.2 # defines

```
#define SECTOR_ERROR SIZE_MAX
#define SLOT_SIZE (PGSIZE / DISK_SECTOR_SIZE)
```

1.3 Global Locks & Variables

```
/* Frame Table. */
static struct hash frame_table;
/* Lock for synchronizing access to frame table. */
static struct lock frame_table_lock;
     - Frame table is organized in the form of hash table.
       System wide frame table & access to it is synchornized with the help of
           frame_table_lock
```

```
/* Lock for accessing the swap table. */
static struct lock swap_table_lock;
/* Swap table implemented as bitmap. */
static struct bitmap *swap_table;
       - Swap table is a bitmap which keeps track of used swap slots in system.
```

1.4 Frame Table Entry

```
struct frame
{
  /* Kernel Virtual address of physical frame. */
  void *kpage;
  /* Address of the page residing at this frame. */
  void *upage;
  /* Struct thread * of the thread. */
  struct thread *t;
  /* Is frame free? */
  bool free;
  /* Hash element to be embedded in hash table. */
  struct hash_elem elem;
};
```

1.5 Supplementary Page Table Entry

Supplementary page table is implemented as hash table, key is upage . It's a per thread table & page eviction may require to change some other thread's supplemental page table.

```
struct page
{
  /* Type of page. */
  enum page_type_t page_type;
  /* Frame's kernel address. */
  void *kpage;
  /* User virtual page. Also used for hashing. */
  void *upage;
  /* Start sector number of the swap disk, where page is available. */
  disk_sector_t sector;
  /* Is the page writable? */
  bool writable;
  /* Name of the file. */
  char file_name[20];
  /* Offset within the file. */
  int32_t file_ofs;
  /* Bytes to read from page (< PGSIZE). */
  uint32_t read_bytes;
  /* Element to be inserted in hash table. */
  struct hash_elem elem;
};
```

2 Algorithms

1. In a few paragraphs, describe your code for locating the frame, if any, that contains the data of a given page.
   Ans: Given a upage, check the supplementary page table of a process for it's presence. You can lookup the page table via 'supplementary_lookup ()". You need to acquire the supplement_lock for accessing a thread's supplementary page table. The lookup function provides 'struct page *', if this return value is NULL, means the page is not mapped to thread's virtual address space. Otherwise the page is mapped to the address space & if the kpage field of this page is not NULL, then the page resides in main memory (frame table too).
   If kpage is NULL, then any access to the page results in a page fault & the page fault handler allocates space to the valid upages according to the 'enum page_type_t'.

2. How does your code coordinate accessed and dirty bits between kernel and user virtual addresses that alias a single frame, or alternatively how do you avoid the issue?
   Ans: My implementation works only with the user virtual address alias to avoid consistency problems with the access & dirty bits of the two page table entries both pointing to the same frame.

3. When a frame is required but none is free, some frame must be evicted. Describe your code for choosing a frame to evict.
   Ans: 'allocator_get_page ()' checks the presence of a free frame, if it's not available, then it searches the frame table for the presence of a not accessed page. If a not accessed page is found then that page is victimized i.e. the upage residing at that address is flushed out. If no page with not_accessed bit set to false, is found, then accessed bits of all the frames is cleared & the first page in the frame table is victimized.

   Writing back the victim upage

   (i) If the upage residing at the victim frame is a read only page then it can be discarded & replaced by another page.

   (ii) If the upage is an ALL_ZERO page then it is written to swap disk (the swap slots are limited when the swap slots gets finished, Kernel Panics) & the supplementary page table of the thread whose upage is going to be replaced is updated, page type is changed to IN_SWAP & sector is updated to the one returned by the 'swap_write ()'.

   (iii) If the upage is a writable page & IN_FILE, then ideally the page should be written back to the disk but currently due to some synchronization issues, my implementation doesn't take that into account.

4. When a process P obtains a frame that was previously used by a process Q, how do you adjust the page table (and any other data structures) to reflect the frame Q no longer has?
   Ans: When a page is evicted then it's entry in the corresponding supplementary page table of the process is removed, i.e. the kpage entry is initialized to NULL in the victim process & the pagedir_clear_page () is called for that page, which means further access to that upage will result into page fault.

While the process which has called 'allocator_get_page ()', if it gets a kpage (ideally it should unless the system runs out of swap space in which case the kernel panics), the kpage is installed in the process's page directory & inserted into it's supplementary page table.

5. Explain your heuristic for deciding whether a page fault for an invalid virtual address should cause the stack to be extended into the page that faulted.
   Ans: *Stack Growth:*
   I have put a restriction on the maximum stack pages to be allocated to a process (currently the limit is 64 pages = 256KB). A stack page is allocated to the process at the beginning in setup_stack call. Each thread struct has a 'limit' & 'num_pages_to_allot' variables. Limit tells that if the stack pointer is greater than that, then the stack access is fine, otherwise it checks the stack pointer to lie within the [limit - num_pages_to_allot * PGSIZE, limit], if it does lie within this range then it means that stack needs to grow otherwise something phishy is going on & you terminate the process.
   In page fault, it checks the stack pointer for the growth & if stack size is increased then page fault handler returns, which means if the page fault was because of stack growth, then it'll not occur next time.
   Else if the stack growth was not the reason for page fault, the fault will occur one more time & this time it checks for the fault address to lie between (esp -4)(PUSH instruction) to (esp -32) (PUSHA instruction), if yes then allocate a new stack page, if process doesn't exceed the limit of 256KB stack size.

3 Synchronization

1. When two user processes both need a new frame at the same time, how are races avoided?
   Ans: Only a single user process can access frame table at any time, as for accessing the frame table it needs to acquire the frame table lock. A page is allocated to process on a call to 'allocator_get_page ()'. "allocator", in my implementation is a manager for allocating frames to user processes. It's initializer 'allocator_init ()' is called by 'main thread' & in the initialization it requests all the pages from the user pool via 'palloc_get_page (PAL_USER)' & creates the frame table (a hash table) where the key is kpage.

2. How do you avoid race between the access to the swap table?
   Ans: Swap table can be accessed only after acquiring the swap table lock which is handled internally by the swap table design. Both 'swap_write ()' & 'swap_read_and_free ()' acquires the lock then only perform the changes to the swap table.

4 Structure Design & Functional Interface

1. Allocator (page allocation handler) design
   "allocator", in my implementation is a manager for allocating frames to user processes. It's initializer 'allocator_init ()' is called by 'main thread' & in the initialization it requests all the pages from the user pool via 'palloc_get_page (PAL_USER)' & creates the frame table (a hash table) where the key is kpage.
   Further the interface provides 'allocator_get_page (void *upage, enum page_type_t, bool

writable)' which returns the frame's address & adds the upage to frame table & add an entry to thread's supplemental page table with kpage.
'allocator_free_page (void *kpage)' frees up the frame table's entry i.e. sets the free bit for the frame table entry.
'allocator_exit ()' is called when the system exits & at that time all the space allocated to the frame table is free'd up.

2. Supplementary page table design
It's a per process hash table used to manage the virtual address space mapping. Initialized in start_process. All the pages which were loaded previously by load segment are just inserted as entries in supplementary PT & the first access itself leads to a page fault (= demand paging).
Interface also provides 'supplementary_insert_zero_page ()' & 'supplementary_insert_kpage ()' which helps in updating the supplementary page table & 'supplementary_exit ()' helps in clearing the space used by the supplementary PT.

3. Swap table design
Swap table is implemented as Bitmap for the fixed number of page slots, created by 'swap_init ()' & destroyed by swap_exit ().
'swap_write ()' helps in writing the upage to the swap slot & returns the start sector number of the 8 consecutive sectors which is saved in page-¿sector.
'swap_read_and_free ()' reads the page from the sector number into the upage given to it & frees up the swap space.

5 Rationale

1. Frame table as Hash Table
Current frame table implementation is somewhat costly as compared to an Array implementation of frame table. Since system knows about the memory size at the time of startup we can allocate array of that size & store the indices(of the frames in supplemental page table) in place of the kpage values, which can be extracted with the help of the index. While implementing i thought that insertion of upage in frame table will be much & hence searching so i thought of going with hash table but now some consistency problems are there with the iterators of frame table. Will be moving to array implementation at later point in time.

2. Is kpage really required in supplementary page table ? Or is upage required in frame table entry?
One can implement the frame table & page tables without these entries but inorder to make implementation simpler, i introduced these variables. But these variables create overhead as well as race conditions, for eg. each call to 'allocator_get_page ()' needs to update a frame table entry & a supplementary table entry (sometimes of some other processes too). During a page eviction process holds frame table lock & it goes to evict another process's page & hence calling 'lock_acquire ()' on that process's supplement_lock & that process might be holding this lock & caused a page fault, hence waiting for frame table lock. Thus creating a deadlock situation.

3. Still needs to solve some race conditions arising in the page-parallel test.