VIRTUAL MEMORY IMPLEMENTATION IN PINTOS DESIGN DOCUMENT OPERATING SYSTEM PROJECT 2

By
Amit Pratap Singh
Person No : 50097261
UBIT Name : amitprat
Email Id: amitprat@buffalo.edu
CSE 521 (Operating System)
Fall 2013

File Structure

1.1 Vm

Page.c : This file is the main executable file in VM directory for this project. This file provides the functions to record file read/write information , page fault handling and page replacement algorithm. Apart from these main functions, there are some utilities functions also which deals assist these main functions and also interact with other functional modules from frame.c and swap.c .
Functionalities and Interactions :
1) **Update Supplemental Page Table :** To record the information in page table to further assist dynamic loading of page information from file. This information may be recorded from *load_segment/stack_setup/stack_extend* in *process.c* or *sysmap/unmap* in *syscall.c* . (*function name : update_suppl_page_table*).
2) **Page Fault Handler :** It handles the page fault and identify the category of page fault (1. MMAP 2. Page Load from executable 3)Stack Growth) and then calls the corresponding functional call to handle it. Then it will either install page or uninstall it in page directory based on above outputs and will return true/false to *page fault* function in *exception.c* .(*function name : page_fault_hdlr*)
3) **Page_replacement_alg :** Here is the implementation of page replacement algorithm (second chance algorithm). When frame allocator tries to allocate a frame for user page and it finds no free kernel virtual address space for frame, then this function is called. In this function, we find a unlocked available frame with reference bit 0. Then we free write the content of frame back to file or swap based upon where it comes from (if it is stack page then write it swap sector else write back to file). Then the frame is freed and control returns to frame allocator. Then, the frame allocator can continue allocating the new frame. *(function name : page_replacement_alg).*
4) **Stack Growth :** This function checks for valid stack expansion checking limit for stack so that it doesn't write over code/data segment and also doesn't go beyond certain limit (here it is 8Mb). A single page is allocated for stack expansion for each request/page fault.
5) **free_all_resources :** It is to free the corresponding page and frame table resources related to a page directory. The corresponding page table/frame entries are deleted when process exits. *(function name : free_all_resources).*

frame.c : This file mainly deals with the frame allocating and freeing the frames on user demand and stores their base address and corresponding page table entry.
Functionalities and Interactions :
1) **frame_alloc_and_lock :** It simply gets a frame from user space and returns the case address to caller.
2) **frame_free :** It frees the frame from page table if not needed further or a entry is need to be freed for new frame space.

swap.c : This file deals with the swap slots for page writing and reading. It uses bitmap to identify free slots on swap space. The space of swap is allocated page aligned . The corresponding read/write sector entry is entered/deleted from page table.
Functionalities and Interactions :
1)**swap_in :** It reads the data from the specified swap sector to frame base address again page is needed to be reused. The corresponding bitmap for the swap sector is cleared after readback.

2)**swap_out :** If a frame is need to be evicted and its data has been modified (dirty bit is set) , then we need to write it back to some storage otherwise changes will be lost. Swap space is used as secondary fast space for swap out/in operations. We write back the data from frame base address to swap sector and frees that frame.

myhash.c : It is wrapper file over normal hash/list operations. Here all the hash/list operations are done using locks and hence we can ensure that all hash/list operations from main files are done sequentially.

1.2 UserProg

Process.c : Here the modifications were done to delay the executable code loading. Lazy page loading is implemented in which a page will be loaded only when it is required. When an interrupt occurs for page fault, page is loaded and instruction is restarted.
Functionalities and Interactions :
1) load_page_data : It reads the data from executable file/data segment into frame when page fault happens.
2) write_back_data_to_file : It writes back the data form frame to file again if the frame's data was modified.

Exception.c : Here in the funcion frame fault , earlier it was just displaying the message for page fault and killing the running process. In the current implementation, page fault is handled and if it is valid page entry then corresponding frame is allocated otherwise process in killed.
Syscall.c : Here the modification where done to implement sys_map/sys_unmap system call. These system calls maps the file data over frame address space and then file can directly be read from memory which results in fast access.
Functionalities and Interactions :
sys_mmap : It maps the open file with supplied fd to an individual map_id (obtained by incrementing the current available map id) . It simply stores the information regarding file into supplemental page table.
sys_unmap: It writes back the frames data to file if the file data was modified.
sys_load_page_runtime : When page fault happens , the data is read from file to frame address space. Now data can directly be accessed from frame address space instead of file.

PRELIMINARIES

Passing Test cases

- Regular (total 5)

    pass tests/vm/pt-grow-stack
    pass tests/vm/pt-grow-pusha
    pass tests/vm/pt-big-stk-obj
    pass tests/vm/pt-write-code2
    pass tests/vm/pt-grow-stk-sc

- Seldom (total 2 or 3 but sometime stuck, unknown issue)
  pass tests/vm/page-linear
  pass tests/vm/page-merge-stk
  pass tests/vm/page-merge-seq

- Bonus Credit (total 13)
  pass tests/vm/mmap-read
  pass tests/vm/mmap-unmap
  pass tests/vm/mmap-overlap
  pass tests/vm/mmap-twice
  pass tests/vm/mmap-write
  pass tests/vm/mmap-shuffle
  pass tests/vm/mmap-bad-fd
  pass tests/vm/mmap-misalign
  pass tests/vm/mmap-null
  pass tests/vm/mmap-over-code
  pass tests/vm/mmap-over-data
  pass tests/vm/mmap-over-stk
  pass tests/vm/mmap-zero

References

- *http://www.cse.buffalo.edu/faculty/tkosar/cse421-521/projects/project-2/WWW/pintos_1.html*
- *http://www.cse.buffalo.edu/faculty/tkosar/cse421-521/projects/project-2/WWW/pintos_6.html*
- *http://www.cse.buffalo.edu/faculty/tkosar/cse421-521/projects/project-2/WWW/pintos.pdf*

Data Structures

2.1 Structures/Unions:

2.1.1 Structure Name : SUPPL_PAGE_TABLE

Purpose : This structure is stores the information regarding what type of page it is and what information need to be read on page fault. It hols information about all type of page loading (Code/Data Segment , MMAP, Stack Pages, PageDir). When page fault happens, one can consult this table to fetch the information regarding which file to read/write and how many bytes are read bytes or zero bytes. Supplementary page table is updated from load_segment, stack_setp/extend and sys_mmap.

Date Filed Name	Date Filed Type	Description	Default Value
Lock	Bool	This is hold the information if the this page's frame is locked and can't be evicted now.	false
Writable	Bool	This stores if the page's frame is writable or	False

		read-only.	
Addr	Void *	This the user virtual address.	NULL
Frame	Frame_struct	The pointer of corresponding frame.	NULL
File	Struct file *	This stores the name of the file from which data to read/write(executable).	NULL
Ofs	Uint32_t	The file offset from where it is read/write.	0
Read_bytes	Uint32_t	The number of bytes to be read from file	0
Zero_bytes	Uin32_t	The number of zero bytes set need to set in page.	0
Reference	Uint8_t	The page reference bit to store if page is recently referenced or not.	0
Mapi_id	Uint8_t	The id of the mmap page.	0
Fp	Struct file *	The file pointer of mmap file.	NULL
Fd	File Desciptor		
Pd	Uint32_t *	The pagedir of corresponding page entry.	0
Esp	Uint8_t*	Stack pointer	NULL
Uint32_t	Stack_pgs	Current number of stack pags	0
Avl	Bool	To check if current page's frame is available or not	False
In_swap	Bool	To check if current page's frame in swap or not	False
Hash_elem	Struct hash_elem	The structure for hashing the list	0

(file name : page.h)

2.1.2 Structure Name : frame_struct

Purpose : This structure holds the information regarding physical memory address and their corresponding page table pointers to refer them back. The hash/list structure is used to implements and its corresponding hash/list utility functions are defined in *my_hash.c* .

Data Field	**Description**	**Default Value**
Base	The physical address of a page	NULL
Page	The pointer of supplemental page table to refer it back whenever a frame's page is need to be consulted.	NULL
Hash_elem	Struct hash_elem	0

(file name : frame.h)

2.2 Variables/Defines/Enumerations:

2.2.1 Enumeration Name : result_enum

Purpose : This enumeration specifies the access type for stack. It simply identifies if it is valid access within the limit for stack growth.

Data Field	Description	Default Value
NOT_REQ_OP	This request is not for stack growth reject it.	0
OK	Stack expanded.	1
ILLEGAL_ACCESS	This stack growth access tries to modify code/data segment.	2
NO_PHY_MEM	No memory is available for further stack growth.	3
REACHED_MAX	The stack growth has reached it max limit.	4

(file name : page.h)

2.2.1 Define Values

Purpose : This enumeration specifies the access type for stack. It simply identifies if it is valid access within the limit for stack growth.

Data Field	Description	Value
STACK_MAX	The max size of stack	8*1024*1024
USER_VADDR_BOTTOM	User virtual address range. Used to identify if faulted address is above it.	0x08048000
STACK_HEURISTIC	The possible stack fault pointer below current stack position	32

2.2.2 Lock Variables

Purpose : The below listed locks were used for synchronization. There can be multiple locks possible to increase the level of parallelism but it will increase complexity and also some deadlock scenarios also. Here i used five different locks for each file (specifically identifying the operation type).

Lock Field	Description	Value
Fs_lock1	To manage the frame operations sequentially.	unset
Page_lk	To synchronize the page lock/unlock operations.	unset
Hash_lock	To synchronize the hash/list operations	unset
File_lock	Lock for file read/write operations. (code and data segment)	unset

Fs_lock	To synchronize read/write from memory mapped files (already existing lock in syscall.c).	unset
Swap_lock	Swap lock to synchronize read/write operations over swap space.	unset

2.2.3 Global Variables

Field	Description	Value
struct page *p	Supplemental page table pointer variable	NULL
struct hash pages	Hash structure for supplemental page table	0
struct hash_iterator iter	List variable for supplemental page table	0
suppl_page_table *victim_page	Victim page in supplemental page table	NULL
Struct _frame_struct *frame;	Frame table pointer	NULL
struct hash frames	Frame hash structure	0
static struct block *swap_device	Initial pointer to swap partition.	NULL
static struct bitmap *swap_bitmap	Initial pointer to allocated swap bitmap for identifying empty/filled swap sectors.	NULL

Algorithm

PAGE TABLE MANAGEMENT

3.1 >> A2: In a few paragraphs, describe your code for locating the frame, if any, that contains the data of a given page.

1) The supplemental page table is looked via user virtual address using hash data structure.
2) If the page entry in page table exist, the frame for this page need to be allocated which is done though frame_alloc_and_lock function call.
3) Then, the user virtual address is mapped to the physical frame address
using the install_page function call.

4) The recorded information in supplemental page table helps to find from where we should read data for a page. If it belongs to executable or mapped region then it is read from file other if the frame is available in swap then it is read from swap space.
5) Whe reading/writing/evicting frame, the frame is locked with lock flag to synchronize the frame access and eviction process.

3.2 >> A3: 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?

In the current implementation, frame aliasing is not implemented and all references to frame are made through user virtual address. Also by default, kernel tries to access the frames through user address if there exists no mapping through kernel virtual address.

PAGING TO AND FROM DISK

3.3 >> B2: When a frame is required but none is free, some frame must be evicted. Describe your code for choosing a frame to evict.

The implemented page eviction algorithm is 'second chance' algorithm. When a page is first brought into memory , the reference bit is set to 1. Also whenever the page is accessed, the reference bit is also updated to 1(through page_set_acccessed). Now when a page fault occurs, it check for victim page if its reference bit is 0 or 1. If victim page is not set(first time), then it is first one. Then if the victim page reference bit is 0, then the page's frame is evicted otherwise we search linearly for a page with reference bit 0. All the pages during this search set to zero until we find a page with reference bit already zero (i.e the page has already been given second chance). If it reaches to the end of list and no victim page is found, it searches again in circular fashion. (It will certainly find a 0 reference bit page as it turned off the bits while traversing in first go). While selecting a page's frame for evection, the lock flag is checked if it is off or on. A page can be evicted only if its lock bit is set to false.

3.4 >> B3: 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?

When a process obtains a frame that was evicted by another process, the new allocated frame is all zero frame and this frame entry neither exists in previous nor in current process page table. While evicting a frame, we set its 1)availability set to zero 2) lock bit to false 3)Remove the frame entry from page table, delete the frame entry from frame table and 4) free the frame. When this frame is assigned to the current process, now the current process holds this frame and corresponding entry for page table pointer and frame table pointer are made .

When allocating space on swap, it uses bitmap to identify free/used spaces over disk. Whenever a page is written to swap disk, the corresponding bitmap is set true. The read/write operations over swap space are page aligned i.e we get a page size space from bitmap and then divide the page into blocks and write each block to a sector. Similarly, we read block by block and set the bitmap for that page_idx as false so that it can used again for another swap allocation.

3.5 >> B4: 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.

To identify if a stack growth fault is not an invalid address request, we need to identify few things. First it is not writing beyond user address space. Second, it is not writing over code/data segment for which we can check always if it is within 32 bytes below the current stack pointer. We can find the current stack pointer from faulted interrupt structure and compare with addressed request to identify its validity. Third thing is need to identify if the process is in not infinite recursion, in that case it will always keep on asking for more stack space. And hence we have to limit it somewhere. The max stack size is 8Mb as per our design which can be further adjusted.

MEMORY MAPPED FILES

3.6 >> C2: Describe how memory mapped files integrate into your virtual memory subsystem. Explain how the page fault and eviction processes differ between swap pages and other pages.

The memory mapping process is similar to virtual memory demand paging but it differs in frame eviction backing store. In memory mapping, mapped file is treated as backing store and data is written back to mapped file. Again while loading the frame, the data is read back from file.

3.7 >> C3: Explain how you determine whether a new file mapping overlaps any existing segment

To identify valid memory mapping and its overlapping with stack and code/data segment, we can check out here for few conditions. First, the memory mapping for address zero is disallowed. Second, it checks the memory alignment of the requested address (i.e. it should be page aligned). Third check it for empty mapping and also identify if the mapping doesn't already exists. These condition avoids the overlapping writing as it would be page aligned and hence the entry will already exist in page table and it will return error.

Synchronization

PAGE TABLE MANAGEMENT

4.1 >> A4: When two user processes both need a new frame at the same time, how are races avoided?

- A frame need to be allocated in following cases :
  - stack growth
  - Memory mapping
  - Reading/writing from executable

All these cases are synchronized using locks so that they don't try to request for a frame at same time.

PAGING TO AND FROM DISK

4.2 >> B5: Explain the basics of your VM synchronization design. In particular, explain how it prevents deadlock.

- Acquire lock for every file system operation (file read/write/seek) to implement the sequential access otherwise it may cause read/write bad data.
- Lock the page's frame for which we need to read/write data so that another thread in parallel can't use this frame for eviction.
- When evicting a frame, check if the page is not locked earlier to ensure that we don't end up evicting a frame that is currently being accessed by another process.
- When allocating a frame, a lock is acquired to ensure that palloc_get_page is accessed sequentially and also frame table entry is added sequentially.
- All hash/list operations are done using locks to ensure that hash/list operations executes sequentially.
- Using above locks, deadlocks situation can be avoided as no two process will be allocating/freeing the frame at same time.

4.3 >> B6: A page fault in process P can cause another process Q's frame to be evicted. How do you ensure that Q cannot access or modify the page during the eviction process? How do you avoid a race between P evicting Q's frame and Q faulting the page back in?

a. When we are loading frame's data from file/swap, we lock the frame and hence it can't selected for eviction while process is allocating a frame for it and loading data into frame.
b. When a page is selected for eviction then it is checked if the page's frame is locked or not. It will selected for eviction only if frame is not locked.
c. When we select a page's frame for eviction, we mark is as locked and hence another process Q can't select this frame for eviction until frame eviction for this page is complete.
d. Also while selecting frame for eviction, we remove the entry from pagedir (i.e its mapping is removed for the attached user address) and hence user process will not be able to access or modify till we are complete with eviction.
e. While removing the frame, we check if this frame's data has been modified or not. If modified then it will be written back to backing store.
f. Backing store for executables/mapped files be the file system itself while for stack pages, we can use swap memory.
g. When Q needs this frame again, it will cause page fault and same steps will repeat.
h. Now if the frame is available in swap then it will be read back from swap otherwise it will be loaded from file system.
i. Again the above steps will repeat.

4.4 >> B7: Suppose a page fault in process P causes a page to be read from the file system or swap. How do you ensure that a second process Q cannot interfere by e.g. attempting to evict the frame while it is still being read in?

- The page's frame is locked while we are reading it and hence another process can't select it for eviction while we are in process of reading the page's frame. Page's lock for the frame is hold in following cases :
  - File/Swap read/write
  - Frame Eviction
  - Stack Growth

4.5 >> B8: Explain how you handle access to paged-out pages that occur during system calls. Do you use page faults to bring in pages (as in user programs), or do you have a mechanism for "locking" frames into physical memory, or do you use some other design? How do you gracefully handle attempted accesses to invalid virtual addresses?

- Not implemented in this. System calls pages are treated as same as normal virtual memory pages and same eviction algorithm applies but they will be write back to mapped files instead.

Rationale

PAGE TABLE MANAGEMENT

5.1 >> A5: Why did you choose the data structure(s) that you did for representing virtual-to-physical mappings?

- Supplemental page table contains following main categories data:
  - Executable File information
    - User virtual address pointer
    - file pointer from where we need to read/write
    - read bytes/zero bytes to ensure we read and write only read bytes data and data is pages can be kept page aligned
  - Mapped file data
    - map id - to identify corresponding mapping
    - Mapped file pointer
    - Mapped file descriptor
  - Stack Record
    - esp - current stack pointer
    - stack_pages - number of stack pages assigned.
  - Page Directory Information
    - pd - to remove page table entries when process is removed and also to identify if the request/response belongs to correct pagedir entry.
    - To set/clear mapping of user/kernal virtual address while swapping in/out.
  - Flags

- Reference bit - to store if page is accessed or not
- writable bit - to identify if the page is writable or not
- lock bit - to synchronize page eviction/read/write

The above mentioned data are required for each subcategory to hold all the information so that we can manage every frame safely.

PAGING TO AND FROM DISK

5.2 >> B9: A single lock for the whole VM system would make synchronization easy, but limit parallelism. On the other hand, using many locks complicates synchronization and raises the possibility for deadlock but allows for high parallelism. Explain where your design falls along this continuum and why you chose to design it this way.

- Yes it is better to have more locks to separately lock an individual section to increase the level of multiprogramming but in order execution of them and handling the deadlock can be big issue and can complicate operating system design. Here in the current design, four locks are using essentially to identify different categories where separate lock can be handled without causing much issue.

MEMORY MAPPED FILES

5.3 >> C4: Mappings created with "mmap" have similar semantics to those of data demand-paged from executables, except that "mmap" mappings are written back to their original files, not to swap. This implies that much of their implementation can be shared. Explain why your implementation either does or does not share much of the code for the two situations.

- Yes the implementation of both is almost same. It differs only in storing data types and some extra checks for memory mapping (like no null address mapping). The implementation for special frame locking for system calls is remained and that may require separate treatment of two codes and hence separate modules are used to implement them.

Survey Questions

6.1 >> In your opinion, was this assignment, or any one of the three problems in it, too easy or too hard? Did it take too long or too little time?

- Yes of course this assignment was much hard in comparison to previous assignment. The synchronization issues were the real difficulty while dealing with the implementation.

6.2 >> Did you find that working on a particular part of the assignment gave you greater insight into some aspect of OS design?

- This project really helped in understanding virtual memory concepts and synchronization issues.

6.3 >> Is there some particular fact or hint we should give students in future quarters to help them solve the problems? Conversely, did you find any of our guidance to be misleading?

- I think, this part is the most difficult part of all three projects of pintos. I think it will be better to give out all the three projects in beginning. This will give students plenty of time for being more familiar with all levels of implementation and dealing with the issues.
-

6.4 >> Do you have any suggestions for the TAs to more effectively assist students, either for future quarters or the remaining projects?

- For such type of projects, I think continuous recitation is necessary regardless of whether they are undergraduate or graduate.