CS342: Operating Systems Lab.

Project 3: Virtual Memory

Submitted By:

Pradeep Bansal

1000143

ASSIGNMENT 1: Frame Table, Supplementary Page Table Implementation and page reclamation on exit.

CHANGED FILES

1. ~/pintos/src/threads/thread.h
2. ~/pintos/src/threads/thread.c

Changed Functions

a. void thread_exit (void)
b. static void init_thread (struct thread *t, const char *name, int priority)

3. ~/pintos/src/userprog/process.c

Changed Functions

a. static bool load_segment (struct file *file, off_t ofs, uint8_t *upage,uint32_t read_bytes, uint32_t zero_bytes, bool writable)

4. ~/pintos/src/userprog/exception.c

Changed Functions

a. static void page_fault (struct intr_frame *).

DATA STRUCTURES

1.

```
struct supplementary_page_table_entry
{
uint32_t *va; /* Virtual address entry.*/
int type; /* 0->in MM,1->in file,2->all zeroes,3->in swap.*/
struct file *file; /* file pointer only useful if type=1.*/
int offset; /* offset into the file.*/
int swap_slot_no; /* Swap slot number only useful if type=3.*/
uint32_t page_read_bytes; /* number of bytes to be read from the disk
applicable only when the type=1.*/
uint32_t page_zero_bytes; /* number of bytes to be zeroed applicable
only when the type=1.*/
bool writable; /* applicable for type=1.*/
struct list_elem elem;
};
```

2.

```
struct frame_table
{
int pid; /* Process pid.*/
```

uint32_t *pte; /* Page Virtual Address.*/
bool free_bit; /* Whether the frame is free or not.*/
};

ALGORITHMS

Working of Code

Since the pages of the process are to be loaded lazily (pure demand paging), Therefore in load_segment() we remove code for loading pages and is shifted to page_fault() in ~/pintos/src/userprog/exception.c.
The supplementary page table is implemented in the load_segment() function. The function loads the segments of the process (code, data) one by one which may themselves span pages.

Supplementary page table has a type field (0->in MM,1->in file,2->all zeroes,3->in swap) which indicates where the page will be found.
Using the page_read_bytes and page_zero_bytes we find which type of page it might be at the start of the process, there are only two possibilities for the type either it could be 1 or 2.
After using the above mentioned local variables we update the supplementary page table accordingly.
When page fault occurs then we check the cause by examining the fault_addr (local variable in page_fault() in ~/pintos/src/userprog/exception.c) if is NULL then we exit the thread if it is above the user accessible memory then also we exit the thread otherwise the page_fault could have occurred due to page not present in the main memory.
We look into supplementary page table of the process and find where the faulting page is if it in file we demand a page from the user pool if we are able to get a page then we just read from the disk and update the page table of the process and subsequently update the frame table as well. If we are not able to get a page, then we have to use page replacement algorithm (second chance LRU) to replace a page and make a free frame then load our faulting page into the Main Memory.

ASSIGNMENT 2: Page Eviction and Stack Growth.

Page Eviction Algorithm:

It is basically an implementation of second chance algorithm. A clock hand is kept for the frame table, which is pointing to a frame number where eviction happened last time. We iterate the clock hand at each frame checking whether the accessed bit of the virtual page in it is set or not. If the bits are set, then we move ahead and unset the accessed bits of the virtual page in this frame.
We do the above until we find a frame which has a virtual page such that it's accessed bit is unset. This page is the one which will be evicted.

Before we actually load the page which faulted we make sure the following things.
1. Page to be replaces is copied to swap.
2. Make supplementary page table entry of type and swap_slot_no of the corresponding page in the supplementary page table of the process to which this page belongs is updated.
3. Invalidate the page table entry of this page in the page table of the process this page belongs to.
4. Update the frame table so as to show this frame as free.

After this has been done we call insert_in_free_frame() function which loads the faulting page into this frame which was freed.
The above mentioned function basically examines the type of the page to be loaded and based on the type takes a correct action .In a case the page has to be loaded from the swap then we make sure the following things are done.
1. Read from the swap slot.
2. Update the page table of the process to which this page belongs to.
3. Swap slot is made free since the data from the swap has been brought onto the frame.
4. Update the frame table.
The other case is in which we read from the file into the free frame's kpage and then update the process's page table accordingly. In case it is a zeroed page we just zero the free frame's kpage and update the corresponding process's page table.
Finally we update the frame table. Whenever the page of the process is loaded into the memory, the supplementary page table is updated so that the type of the page is set to 0(that is the page is in Main Memory)

Stack Growth

Hueristic to find the “stack accesses”
Whenever a page fault occurs then we first examine whether the faulting address is NULL or is it invalid access(access to kernel virtual space) or whether the faulting address is in the process's supplementary page table if the demand is for another stack page then certainly the faulting address will not be found.
In this case, it is a stack growth access if and only if faulting address is above the address being pointed by the 32 bytes below the current stack pointer.
Otherwise we terminate the process.

Allocation of pages to Stack.

The pages if available in the user pool are allocated in same manner as described above.
But if they are not available in the user pool, then we do the following.

1. Find a free frame.
2. If not available, call evict_page() (from pintos/src/userprog/exception.c) and allocate the kpage of the frame to the stack also since this is a new page also update the supplementary page table and update the page table of the current process and also update the frame table.
3. If it is available then we do 2 as mentioned above except for calling evict_page().

SYNCHRONIZATION

No synchronization was needed for this project.

RATIONALE

Advantages:

1. The pure demand paging has been implemented with second chance algorithm for eviction and also stack growth is supported.