Lab3b Design Doc

Project 3b: Virtual Memory

Preliminaries

Stack Growth

ALGORITHMS

> A1: 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.

New member in `struct thread` for page fault in kernel.

```
uint8_t* cur_esp;               /**< Current stack pointer. */
```

```
struct thread* t = thread_current();
void*esp = user ? f->esp : t->cur_esp;
```

In `page_fault` ,first extract the current stack pointer.If page fault is caused by user processes,we can get stack pointer from `struct intr_frame` .If page fault occurs in kernel, `t->cur_esp` records the current stack pointer.

```
bool stack_growth = (esp < fault_addr) || (fault_addr == esp - 4) || (fault_addr == esp - 32);/*If stack_growth is true,stack should be extended.*/
```

Consider the cases that lead to stack growth.
(1)Allocate local variables.Stack pointer will decrease. But because of lazy loading,when user process try to access the allocated space at first time,page fault will happen.In this case, `esp` will be lower than `fault_addr` .
(2)CALL ,PUSH and PUSHA instructions.
fault_addr == esp - 4 or fault_addr == esp - 32.

Memory Mapped Files

DATA STRUCTURES

> B1: Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.

Add new member in `struct thread` .

```
    struct list map_list;          /**< List of mapping between files and virtual pages.*/
    int next_mapid;                /**< The next valid mapping id,initialize it to zero.*/
```

```
struct mmap_inode
{
    mapid_t mmap_id;          /**< The mapping id./
    struct file* file;                /**< The mapped file.*/
    void* start_upage;           /**< The virtual address the file mapped to*/
    int file_size;                    /**< The length of the file*/
    struct list_elem elem;        /**<Defined for map_list.*/
};
```

ALGORITHMS

> B2: 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.

(1)How memory mapped files integrate into your virtual memory subsystem?
In system call mmap,I install all the pages of the file to be mapped on this process's supplemental page table,then build and initialize a node of `struct mmap_inode` and put it on this process's `map_list` .

(2) How the page fault and eviction processes differ between swap pages and other pages.
(a)page fault:If the faulted page is stored in swap block device,I first read the page back into frame and then clear the associated bit in `swap_bitmap` .If the faulted page isn't a swap page,I allocated a frame for this page and then either load the page from file system or initialize all bits of the page to zero.For all faulted pages,thier supplemental page table entry will be modified.
(b)eviction:To evict a swap page out of our memory subsystem,we first need to read it back into a frame,and then we can read the page back into file system if the page is modified.For the page on the frame,we can directly read the modified page back into file system without doing anything special.For pages in file system,there is nothing to do.

> B3: Explain how you determine whether a new file mapping overlaps any existing segment.

```
/** Return true if supplemental page table has an entry
 * whose virtual address is UPAGE.Otherwise return false.
*/
bool vm_supple_has_entry (struct supplemental_page_table *stable, void *upage)
{
    struct supple_table_entry*supple_entry = vm_supple_lookup(stable,upage);
    return supple_entry != NULL;
}
```

`vm_supple_has_entry` will scan the supplemental page table `stable` and find if there is a page whose virtual address is `upage` .

All the allocated virtual pages of a process has an entry on this process's supplemental page table.So if we find an existing entry for the virtual address to which the new file will be mapped,this new file mapping overlaps another existing segment.

RATIONALE

> B4: 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.

My implementation does not share much of the code for the two situations,but the main functions used are generally the same.
Reasons:
Although the main part of the two situations is both to install the pages of the file on supplemental page table,a small difference is that before installing the mapped file page,it should be checked that if there is overlap.That's the one reason my code doesn't share a lot.What's more,in part `mmap` I use a new data structure `map_list` ,so I think the independence of code will do benefit to readability.