Project 3b: Virtual Memory

Preliminaries

> Fill in your name and email address.

Jin Yang 2000012935@stu.pku.edu.cn

> If you have any preliminary comments on your submission, notes for the TAs, please give them here.

> I have deleted some functions that are not frequently used and separated some functions to make each one only do one job. And I have fixed some horrible synchronization bugs.

> Some of the functions are renamed, but they still have do the same job.

> Please cite any offline or online sources you consulted while preparing your submission, other than the Pintos documentation, course text, lecture notes, and course staff.

> "Pintos Guide" written by Stephen Tsung-Han Sher, his website is https://www.stsher.com/

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.

> Firstly, the virtual address must be a user address between **0xc0000000** and **0xbf800000**, since the max stack size is 8MB. Any process tries to grow beyond 8MB will be killed.
> If the spte is not found in the current thread's supplement page table, then the virtual address could be a stack access.
> Since the PUSHA instruction could cause an access to be 32 bytes below the stack pointer, any user memory access between PHYS_BASE and esp - 32 is valid.

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.

> In process.h:

```
struct mmap_file
{
    int mapid;                          /**< Mmap id. */
    struct sup_page_table_entry *spte; /**< Page table entry. */
    struct list_elem elem;              /**< List element. */
};
```

> A mmap file.

```
struct process
{
    ...
    int mapid;          /**< Max mapid of the process. */
    struct list mmaps; /**< Mmap files of the process. */
    ...
};
```

> Record all mmap files for a user process.

> In syscall.h:

```
void *_esp;
```

> For check whether the given address is valid.

```
typedef int syscall_function(int, int, int);
```

> A syscall function.

```
struct syscall
{
    size_t arg_num;
    syscall_function *func;
};
```

> A syscall struct, got the idea from block read and write.

> Also, the syscall function table is now as:

```
static struct syscall syscall_table[] = {
    ...
    {(int)argument_num, (syscall_function *)syscall}
    ...
};
```

> In frame.h:

```
struct frame_table_entry
{
    void *kpage;                        /**< Physical address of the page. */
    struct sup_page_table_entry *spte; /**< Supplyment page table entry. */
    struct thread *owner;               /**< The owner thread of the frame. */
    struct list_elem lelem;             /**< List element. */
};
```

> Remove **pinned** and **helem**, change **upage** into **spte**.

> In frame.c:
> Remove hash table **frame_table**, Since list is enough and easier to debug and do synchronization.

> In page.h:
> Add **pinned** to spte, and add a **page_lock** to spt.

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.

> Mmap files are just like executables: lazy load, read from files and loaded into physical memories. The main difference is mmap file must be written back to the corresponding file during eviction and munmap. In munmap, the process iterates through its mmap_list, writes dirty pages back to disk, closes the maped file and does other cleanup. During a process exit, all mapped files are munmapped, while other pages are simply 'thrown away'.

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

> The file is mapped in page-by-page to the corresponding address. For each page, its spte is added to the thread's spt. If the spt detects duplicate entries, then the new file mapping overlaps with an existing segment. Then, all the previous mappings are unmapped and the mmap fails.

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.

> Since they have similar behaviors, I use **load_file** function to load pages from file and mmap, and the add_spte function is almost the same except the page status and add_mmap also add the mmap file to a process's mmap_list. And in frame eviction, the mapped dirty pages written to their corresponding file, while the clean ones are ignored. This makes implementing the mmap files an extension of executables when it comes to loading into memory.