Project 3b: Virtual Memory

Preliminaries

> Fill in your name and email address.

Hao Wang tony.wanghao@stu.pku.edu.cn

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

I achieved full score on my local laptop.

> 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.

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.

My heuristic is pretty simple. First, the faulting page should be within the possible stack space, which is `PHYS_BASE - STACK_SIZE ~ PHYS_BASE`. Additionally, we make an assumption that all valid stack pointers should be at most 4096 bytes lower than the stack pointer. That is, `upage >= esp - 4096`.

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 `mmap.h`:

```
struct mmap_file {
    struct file *file;      /**< pointer to the file. */
    struct hash_elem elem;  /**< hash element. */
    mapid_t mapid;          /**< map id for this entry */
    void *addr;             /**< mapping address */
};
```

In `process.h/struct process`:

```
    struct hash mmap_table;             /**< The table for mmap entries */
    int next_fd;                        /**< Next fd assigned. */
```

In `page.h/enum page_type`:

```
PG_MMAP
```

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.

Each process keeps a `hash` table of mmaped files `mmap_table`. Each time we `mmap` a file, the corresponding infos are stored into a new entry in `mmap_table`. Additionally, Mmaped files have a special type `PG_MMAP` in the supplementary page table.

Each time a mmap page triggers a page fault, same as `PG_FILE` and `PG_MISC` pages, we read from file the content of the page. Note that this sort of page is never read from swap device, since each time mmap pages are evicted, they are written directly to the original file. If the page is not dirty, both swap pages and mmaped pages would not be saved.

When a process exits, it would unmap all its mmaped files.

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

I iterate through all the pages, during which checks the supplementary page table for the current process if the same page has been assigned. If there is an existing `SPE` in `SPT` with the same `upage`, the file mapping overlaps with another 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.

I only shared much of the code in page_fault. I think the reason is that from the abstraction level, the file mapping table is an independent system to the other components of the kernel, so I opened a new `mmap_table` for the mapped files. However, when it comes to the demand paging part, there are a lot of similarities, like how the content of the page should be partly read from files and partly set to zero. The only difference is when doing eviction, the mapped pages need special care. Thus, much of the codes are shared in paging part, but not the part where I take care of the mapping-specific information.