Lab 3b Design Doc

Preliminaries

> Fill in your name and email address.

Zhang Lingxin 张龄心 2100013018@stu.pku.edu.cn

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

> 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.

https://static1.squarespace.com/static/5b18aa0955b02c1de94e4412/t/5b85fad2f950b7b16b7a2ed6/1535507195196/Pintos+Guide

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.

These are the conditions that must be met before the stack grows:

- The faulted address is above current `esp` or equals `esp - 4` or `esp - 32`
- The faulted address is below `PHYS_BASE` and above `PHYS_BASE - MAX_STACK_SIZE`
- There's no entry in the supplemental page table for the faulted page

If all the conditions are satisfied, we grow the stack.

Here is the code:

```
bool stack_fault = fault_addr >= esp || fault_addr == esp - 4 ||
                     fault_addr == esp - 32;
bool stack_to_grow =
         (uint64_t)fault_addr >= ((uint64_t)PHYS_BASE - MAX_STACK_SIZE);

/* If OK, make the stack grow with a new zero page */
if (stack_fault && stack_to_grow && fault_addr < PHYS_BASE) {
   if (!supp_page_to_spte(cur, fault_page)){
      supp_install_page_zero (cur, fault_page);
   }
}
```

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.

```
/* syscall.h */
```

```
typedef uint32_t mmapid_t;

/* Record the information in mmap */
struct mmap_entry{
    mmapid_t mmapid;        /**< mmap id. Identifies each mmap pair uniquely. */
    struct file* f;         /**< The file mapped from. */
    void *addr;             /**< The starting address of mmap. */
    size_t size;            /**< The size of mapped file. */
    struct list_elem elem;
};

/* thread.h */

struct thread{
    ...
    struct list mmap_list;  /**< A list of struct mmap_entry. */
    size_t mmap_id;         /**< The next mmap_id. */
    ...
};
```

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?

   If a thread invokes a system call to map a file, the corresponding function `syscall_mmap()` in `syscall.c` will:

   - call `supp_install_page_file()` in `page.c`, which creates a `struct supp_page_table_entry` tagged `IN_FILESYS` and insert it into the supplementary page table of current thread
   - create a `struct mmap_entry` to store the information of memory mapping, and insert it into the `mmap_list` of current thread

   When the memory mapping is unmapped, the two entries are removed from the corresponding lists.

   So the memory mapping mechanism works as part of the whole virtual memory subsystem, managing pages mapped from files.

2. How the page fault and eviction processes differ between swap pages and other pages?

   Swap faulting and eviction differs from file faulting only in the way that we restore/store the pages. When a page fault occurs, the page fault handler will bring in the page according to the page type:

   - A swapped-out page: get a free frame (by finding a free one or evicting one) and swap it in.
   - An all-zero page: get a free frame and set it as all zero.
   - A page from file(mapped): get a free frame and read the information from the file in

   Eviction only move a page in frame to the swap slots. Pages mapped from file and all-zero pages do not use the frame, so they are never evicted.

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

Before the mapping, the function will check if the mapping request is valid. This includes checking the supplementary page table for all the pages of the new file - if any page already exists in the supplementary page table entry, the system call fails and return -1.

```
/* 14. mmap */
static void
syscall_mmap(struct intr_frame *f){
  ...
```

```
  /* Check if the pages are already mapped */
  for(ofs = 0; ofs < size; ofs += PGSIZE){
    ptr = addr + ofs;
    if(supp_page_to_spte(cur, ptr) != NULL){
      on_failure(f);
      return;
    }
  }
  ...
}
```

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 shares much of the code and only separates the system call part of memory mapping, storing `mmapid`, a pointer to the source file and other necessary information for system call handler in `struct mmap_entry` defined in `syscall.h`. Most of information is stored in supplementary page table like other pages.

I choose the implementation because it simplifies the design and reduces the debugging workload. As I know the supplementary page table works correctly in Lab 3a, I don’t need to adjust its overall structure when a bug occurs.