Project 3: Virtual Memory

Operating Systems — National University of San Marcos

Group

- Damaris Del Carpio damaris.delcarpio@unmsm.edu.pe
- Paolo Flores paolo.flores2@unmsm.edu.pe

Preliminaries

> 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://en.wikipedia.org/wiki/Page_table
- https://courses.cs.vt.edu/cs5204/fall15-gback/lectures/Project3Help.pdf
- Starting with Project 3 by Tahmid Haque https://www.youtube.com/watch?v=PVKZkkiyXD0

Page Table Management

Data Structures

> **A1:** 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.

Frame table entry An entry of the page table. Maps a kernel page to a user page, storing the thread that owns it.

```
struct fte
  {
    void *kpage;
    void *upage;

    struct thread *t;

    struct list_elem list_elem;
  };
```

Supplemental Page Table Entry per-process data structure that tracks supplemental data for each page, such as location of data (frame/disk/swap), pointer to corresponding kernel virtual address, active vs. inactive, etc.

The "supplemental" is a concept defined Pintos itself.

Reference: https://www.cs.utexas.edu/~ans/classes/cs439/projects/pintos/WWW/pintos_4.html#SEC65

```c
struct spte
{
  void *upage;
  void *kpage;

  struct hash_elem hash_elem;

  int status;

  struct file *file;    // File to read.
  off_t ofs;            // File off set.
  uint32_t read_bytes;  // Bytes to read or to set to zero.
  uint32_t zero_bytes;
  bool writable;        // whether the page is writable.
  int swap_id;
};
```

Algorithms

> **A2:** In a few paragraphs, describe your code for accessing the data stored in the SPT about a given page.

We used the hashing utilities located at `src/lib/kernel/hash.c`.

```c
struct spte *
get_spte (struct hash *spt, void *upage)
{
  struct spte e;
  struct hash_elem *elem;

  e.upage = upage;
  elem = hash_find (spt, &e.hash_elem);

  return elem != NULL ? hash_entry (elem, struct spte, hash_elem) : NULL;
}
```

The `get_spte()` function retrieves the Supplemental Page Table entry for a given user page (upage). It creates a temporary spte with the upage, searches for it in the hash table using hash_find(), and returns the corresponding spte if found. If not found, it returns NULL.

> **A3:** How does your code coordinate accessed and dirty bits between kernel and user virtual addresses that alias a single frame, or alternatively how do you avoid the issue?

We address this issue by not storing the dirty/access bits in the frame, but rather in the page. This way, every alias will share the same dirty/access bits.

Synchronization

> **A4:** When two user processes both need a new frame at the same time, how are races avoided?

We avoid races using two main mechanisms. First, the frame table is protected by a lock, ensuring only one process can access and modify it at a time.

Second, the `palloc_get_page` function uses an exclusive lock (pool's lock) to prevent races during frame allocation. It performs a `bitmap_scan_and_flip` to find a free page index, and only one process can access the bitmap at a time.

Once a page is allocated, its bit is set to in-use, and other processes must wait until the first one completes before accessing the bitmap to get another frame.

Rationale

> **A5:** Why did you choose the data structure(s) that you did for representing virtual-to-physical mappings?

Aside from the already defined PTE (Page table entry) we added another structure recommended by the Pintos reference called the Supplemental Page Table Entry (SPTE).

The supplemental page table serves two main purposes.

- During a page fault, the kernel refers to it to determine the data associated with the faulting virtual page.
- When a process terminates, the kernel uses the supplemental page table to identify which resources need to be freed.

This information is beyond the capacity of the page table entry, which only contains the physical address of the frame. If we stored this information, the page table would end up being too large.

It also adds lookup efficiency, as we used a hash table to store the SPTEs,

Stack Growth

Data Structures

1. ESP: In order to dynamically grow the stack, it is necessary to record the stack pointer for each thread. Therefore, espa field was added to store the stack pointer.

```
// File: threads/thread.h
struct thread {
  // ...
    void *esp; // Stack pointer.
  // ...
}
```

Algorithms

Stack growth during a page fault is handled in page_fault(). If a fault occurs due to insufficient stack space and the address is within a close range (e.g., 32 bytes) of the current stack, a Supplemental Page Table Entry with a value of 0 is created to allocate additional space.

```
//File: userprog/exception.c
static void
page_fault(struct intr_frame *f) {
  //...
  upage = pg_round_down (fault_addr);

  if (is_kernel_vaddr (fault_addr) || !not_present)
      sys_exit (-1);

  spt = &thread_current ()->spt;
  spe = get_spte (spt, upage);

  esp = user ? f->esp : thread_current ()->esp;
   if (esp - 32 <= fault_addr && PHYS_BASE - MAX_STACK_SIZE <= fault_addr) {
    if (!get_spte (spt, upage))
      init_zero_spte (spt, upage);
   }

  if (load_page (spt, upage))
    return;

  sys_exit (-1);
  //...
}
```

The stack setup process has changed. Instead of directly allocating pages with palloc_get_page(), pages are now allocated through falloc_get_page(). Additionally, the allocation of the Supplemental Page Table Entry (SPTE) has been integrated into the process of allocating space near PHYS_BASE.

```
// File: userprog/process.c
static bool
setup_stack (void **esp)
{
  uint8_t *kpage;
  bool success = false;

  // kpage = palloc_get_page (PAL_USER | PAL_ZERO);
  kpage = falloc_get_page (PAL_USER | PAL_ZERO, PHYS_BASE - PGSIZE);
  if (kpage != NULL)
```

```
    {
      success = install_page (((uint8_t *) PHYS_BASE) - PGSIZE, kpage, true);
      if (success)
        {
          init_frame_spte (&thread_current ()->spt, PHYS_BASE - PGSIZE, kpage);
          *esp = PHYS_BASE;
        }
      else
        {
          // palloc_free_page (kpage);
          falloc_free_page (kpage);
        }
    }
  return success;
}
```