گزارش سوال ۲

حسنا بشیریان

محمدرضا حسینی

امیرمسعود شاکر

دانیال علی عظیمی

سید عباس میرقاسمی

در پیاده سازی قبلی سیستم عامل pintos این شکل است که page table ها و Supplementary table ها به عنوان بخشی از کرنل در حافظهٔ اصلی قرار دارند. اما در سیستم عامل های واقعی برای افزایش فضای حافظهٔ اصلی خود این جداول نیز به شکل Virtual در حافظهٔ اصلی قرار می‌گیرند؛ یعنی بخشی از آن‌ها در حافظهٔ جانبی قرار دارد و در صورت نیاز در حافظهٔ اصلی بارگذاری می‌شوند. حال برای پیاده سازی این موضوع وارد عمل میشویم.

فرضیات:

در این بخش فرض کردیم که نیاز است در هنگام بلاک شدن یا صبر کردن هر thread، pagetable های ذخیره شده در page directory را به حافظه مجازی (جانبی) منتقل شوند و در هنگامی که قرار است thread دوباره در وضعیت اجرای running قرار گیرد، page table های مربوطه به آن thread به حافظه اصلی باز گردد.

ابتدا برای این موضوع از توابع زیر که قبلا ایجاد شده است، استفاده میکنیم:

```c
bool vm_load_page(struct vm_page *page, bool pinned)
{
  /* Get a frame of memory. */
  lock_acquire(&load_lock);/* If we have a read-only file try to look for a frame if any that contains the same data. */
  if (page->type == FILE && page->file_data.block_id != -1)
    page->kpage = vm_lookup_frame(page->file_data.block_id);
  /* Otherwise obtain an empty frame from the frame table. */
  if (page->kpage == NULL)
    page->kpage = vm_get_frame(PAL_USER);
  lock_release(&load_lock);
  vm_frame_set_page(page->kpage, page);
  bool success = true;
  /* Performs the specific loading operation. */
  if (page->type == FILE)
    success = vm_load_file_page(page->kpage, page);
  else if (page->type == ZERO)
    vm_load_zero_page( kpage: page->kpage);
  else
    vm_load_swap_page( kpage: page->kpage, page);
  if (!success)
  {
    vm_frame_unpin(page->kpage);
    return false;
  }/* Clear any previous mapping and set a new one. */
    pagedir_clear_page(page->pagedir, page->addr);
  if (!pagedir_set_page(page->pagedir, page->addr, page->kpage, page->writable))
  {
    ASSERT(false);
    vm_frame_unpin(page->kpage);
    return false;
  }
  pagedir_set_dirty(page->pagedir, page->addr, false);
  pagedir_set_accessed(page->pagedir, page->addr, true);
  page->loaded = true;/* On succes we leave the frame pinned if the caller wants so. */
  if (!pinned)
    vm_frame_unpin(page->kpage);
  return true;
}
```

```c
void vm_unload_page(struct vm_page *page, void *kpage)
{
  lock_acquire(&unload_lock);
  if (page->type == FILE && pagedir_is_dirty(page->pagedir, page->addr) &&
      file_writable(page->file_data.file) == false)
  {
    /* Write the page back to the file. */
    vm_frame_pin(kpage);
    sys_t_filelock(true);

    file_seek(page->file_data.file, page->file_data.ofs);
    file_write(page->file_data.file, kpage, page->file_data.read_bytes);
    sys_t_filelock(false);
    vm_frame_unpin(kpage);
  }
  else if (page->type == SWAP || pagedir_is_dirty(page->pagedir, page->addr))
  {
    /* Store the page to swap. */
    page->type = SWAP;
    page->swap_data.index = vm_swap_store(kpage);
  }
  lock_release(&unload_lock);

  pagedir_clear_page(page->pagedir, page->addr);
  pagedir_add_page(page->pagedir, page->addr, (void *)page);
  page->loaded = false;
  page->kpage = NULL;
}
```

```c
palloc_free_page (void *page)
{
  palloc_free_multiple ( pages: page,  page_cnt: 1);
}

```

```
117  void
118  palloc_free_multiple (void *pages, size_t page_cnt)
119  {
120    struct pool *pool;
121    size_t page_idx;
122
123    ASSERT (pg_ofs (pages) == 0);
124    if (pages == NULL || page_cnt == 0)
125      return;
126
127    if (page_from_pool (&kernel_pool, pages))
128      pool = &kernel_pool;
129    else if (page_from_pool (&user_pool, pages))
130      pool = &user_pool;
131    else
132      NOT_REACHED ();
133
134    page_idx = pg_no (pages) - pg_no (pool->base);
135
136  #ifndef NDEBUG
137    memset ( Dst: pages,  Val: 0xcc,  Size: PGSIZE * page_cnt);
138  #endif
139
140    ASSERT (bitmap_all (pool->used_map, page_idx, page_cnt));
141    bitmap_set_multiple (pool->used_map, page_idx, page_cnt, false);
142  }
```

حال تغییراتی که در کد به وجود آورده ایم را نشان میدهیم:

در ساختار thread مان لازم است تغییراتی به وجود بیاوریم تا آدرس page directory و حافظه مجازی هر thread را در آن ذخیره کنیم.

```c
struct thread
{
  /* Owned by thread.c. */
  tid_t tid;                 /* Thread identifier. */
  enum thread_status status; /* Thread state. */
  char name[16];             /* Name (for debugging purposes). */
  uint8_t *stack;            /* Saved stack pointer. */
  int priority;              /* Priority. */
  struct list_elem allelem;  /* List element for all threads list. */

  /* Shared between thread.c and synch.c. */
  struct list_elem elem; /* List element. */

  int base_priority;    /* Base priority of a thread. */
  bool donated;         /* If a thread has donated priority. */
  struct list locks;    /* List of locks hold by a thread */
  struct lock *blocked; /* The lock blocking the thread */

  int nice;          /* Nice value. */
  int32_t recent_cpu; /* Recent CPU value in 17.14
                         Fixed Point representation. */
  // our changes-----------------------------------------------------------------------------------------------------------------------------
  uint32_t *pagedir; // keep page directory address
  uint32_t *vm_address; // keep page address in virtual memory
  // ----------------------------------------------------------------------------------------------------------------------------------------
#ifdef USERPROG
  ...
#endif
```

در بخش بلاک شدن هر thread، page table های هر thread را به حافظه جانبی میبریم و حافظه اختصاص یافته به آن در memory را آزاد میکنیم.

```c
void thread_block(void) {
    struct thread *cur = running_thread();

    ASSERT(!intr_context());
    ASSERT(intr_get_level() == INTR_OFF);
    // swap out page table
    thread_current()->status = THREAD_BLOCKED;
    //our change-----------------------------------------------------------------------------------------------------------------------------
    for (int i = 0; i < VM_INDEX; ++i) {
        vm_unload_page(cur->pagedir + i , true)
        palloc_free_page(cur->pagedir + i)
    }
    //---------------------------------------------------------------------------------------------------------------------------------------
    schedule();
}
```

در این بخش قبل از اینکه thread به حالت running رفته و اجرا شود، دوباره page table های آن thread را از حافظه جانبی به حافظه اصلی میاوریم. فرآیند دادن حافظه جدید اصلی نیز همان طورکه در توابع بالایی به آن اشاره شد، نیز به صورت خودکار صورت

میپذیرد.

```
707  void thread_schedule_tail(struct thread *prev) {
708      struct thread *cur = running_thread();
709
710      ASSERT(intr_get_level() == INTR_OFF);
711
712      /* Mark us as running. */
713      cur->status = THREAD_RUNNING;
714      //our change----------------------------------------------------------------------------------------------------
715      for (int i = 0; i < VM_INDEX; ++i) {
716          vm_load_page(cur->vm_address + i, true)
717      }
718      //--------------------------------------------------------------------------------------------------------------
719
720      /* Start new time slice. */
721      thread_ticks = 0;
722
723  #ifdef USERPROG
724      ...
726  #endif
```