Projeto 2: User programs

Equipe:

Rodrigo Pontes de Oliveira Lima (rpol)

Thiago Henrique Rezende Brito (thrb)

Vitor Manoel de Melo Silva (vmms)

Parte 1: Page Fault

Na parte 1 do projeto, é tratado o caso onde um processo acessa um endereço virtual que não está no diretório de páginas. Isso causa um PF (Page Fault) e o dever do sistema operacional é alocar a página necessária caso o endereço acessado seja válido. O tratador de PF é a função `page_fault()` no arquivo `exception.c`.

Primeiro é necessário realizar a validação do endereço. No código, é verificado as 4 situações de invalidação: endereço nulo, endereço do kernel, endereço fora da stack de usuário ou se o acesso está a mais de 32 bytes da posição atual da stack. As seções de código do tratador que realizam essa validação são as seguintes:

```
if(user){
    if (fault_addr == NULL || !is_user_vaddr(fault_addr)) {
        sys_exit(-1);
    }
        (...)
    if(not_present && spt_entry != NULL){
        (...)
    }
        (...)
    }
    else if(fault_addr >= f->esp - 32 && fault_addr < PHYS_BASE){
        (...)
    }
    sys_exit(-1);
```

Foram utilizadas duas estruturas de dados adicionais: A *frame_table* e a *supplemental_page_table*. Os elementos dessas estruturas são representados pelos seguintes structs:

```
struct frame_table_entry {
    uint32_t* frame;
    struct thread* owner;
    struct sup_page_table_entry* page;
    bool pinned; // indica se o frame está sendo lido por um processo e não
pode sofrer swap
    struct list_elem list_elem;
};
```

```
struct sup_page_table_entry {
    uint32_t* vaddr;
    struct file *file;
    off_t offset;
    uint32_t read_bytes;
    uint32_t zero_bytes;
    size_t index;
    bool writable;
    bool dirty;
    bool accessed;
    bool swap; // indica se a página está na região de swap
    bool in_memory; // indica se está na memória principal
    struct hash_elem hash_elem;
};
```

A *frame_table* armazena informações sobre os frames na memória principal enquanto a *supplemental_page_table* armazena informações sobre as páginas alocadas para o gerenciamento por parte do SO.

A função `load_segment` em `process.c` foi modificada para criar uma entrada na *supplemental_page_table* ao invés de alocar uma página física.

```
load_segment (struct file *file, off_t ofs, uint8_t *upage,
              uint32_t read_bytes, uint32_t zero_bytes, bool writable)
{
  ASSERT ((read_bytes + zero_bytes) % PGSIZE == 0);
  ASSERT (pg_ofs (upage) == 0);
  ASSERT (ofs % PGSIZE == 0);

  struct thread* curr = thread_current();

  while (read_bytes > 0 || zero_bytes > 0)
    {
      /* Calculate how to fill this page.
         We will read PAGE_READ_BYTES bytes from FILE
         and zero the final PAGE_ZERO_BYTES bytes. */
      size_t page_read_bytes = read_bytes < PGSIZE ? read_bytes : PGSIZE;
      size_t page_zero_bytes = PGSIZE - page_read_bytes;

      struct sup_page_table_entry *spt_entry = (struct sup_page_table_entry
*)malloc(sizeof(struct sup_page_table_entry));
      if(!spt_entry) return false;

      spt_entry->file=file_reopen(file);
      spt_entry->offset=ofs;
      spt_entry->vaddr=(void *) pg_round_down(upage);
      spt_entry->read_bytes=page_read_bytes;
      spt_entry->zero_bytes=page_zero_bytes;
      spt_entry->writable=writable;
      spt_entry->dirty=false;
```

```
      spt_entry->accessed=false;
      spt_entry->swap = false;

      sup_page_insert(&curr->sup_page_table, spt_entry);



      /* Advance. */
      read_bytes -= page_read_bytes;
      zero_bytes -= page_zero_bytes;
      upage += PGSIZE;
      ofs += page_read_bytes;
    }
  return true;
}
```

A função `setup_stack` em `process.c` foi modificada para criar uma entrada na *supplemental_page_table* e na *frame_table* para a pilha do usuário.

```
static bool
setup_stack (void **esp, char **argv, int *argc)
{
  uint8_t *kpage;
  bool success = false;
  struct thread *curr = thread_current();

  kpage = palloc_get_page (PAL_USER | PAL_ZERO);
  if (kpage != NULL)
    {
      success = install_page (((uint8_t *) PHYS_BASE) - PGSIZE, kpage,
true);
      if (success) {
         struct sup_page_table_entry *spt_entry = (struct
sup_page_table_entry *)malloc(sizeof(struct sup_page_table_entry));
         spt_entry->vaddr = (void *)((uint8_t *) PHYS_BASE) - PGSIZE;
         spt_entry->writable = true;
         spt_entry->in_memory = true;
         spt_entry->swap = false;
         spt_entry->file = NULL;
         spt_entry->offset = 0;
         spt_entry->read_bytes = 0;
         spt_entry->zero_bytes = PGSIZE;
         spt_entry->index = 0;

         sup_page_insert(&curr->sup_page_table, spt_entry);

         lock_acquire(&frame_lock);
         frame_table_insert(spt_entry, kpage);
         lock_release(&frame_lock);
        (...)
```

Parte 2: Swap, Eviction e Reclamation

Na parte 2 do projeto, é tratado o caso em que um processo pode acessar um endereço virtual que não está mapeado na memória principal, mas a memória principal está cheia. Nessa situação, entram em ação os algoritmos de `eviction()` e de `reclamation()`.

Eviction

Todo o algoritmo de `eviction()` está centralizado em `frame.c`

À priori, o eviction percorre a tabela de molduras tratando-a como uma lista circular. Para isso, uma variável global do tipo `struct list_elem *` é inicializada, para que a posição da última moldura acessada seja "lembrada" e outro processo, ao chamar `eviction()`, continue de onde o último parou. Em paralelo, há implementação do LRU(Last Recently Used) através do algoritmo de segunda chance. Ou seja, ao verificar um frame que foi acessado recentemente(`pagedir_is_accessed(curr->pagedir, upage) == true`), o algoritmo põe esse valor como *false*(`pagedir_set_accessed(curr->pagedir, upage, false);`). Assim, ao passar por essa moldura novamente, caso não tenha sido acessada, ela será o alvo e retirada da memória principal para a área de *swap*.

```
void
frame_table_init(void){
    (...)
    frame_elem = NULL; // Inicializa o list_elem global como nulo
}

void
frame_table_insert(struct sup_page_table_entry *sup_page_table_entry, void*
frame_addr){
    (...)
    if(frame_elem == NULL) frame_elem=list_begin(&frame_table); // Depois
da primeira inserção, coloca o frame_elem como o primeiro da lista

}

void *eviction(){
  lock_acquire(&frame_lock);

  while(true) {
    if(list_empty(&frame_table)) {
      lock_release(&frame_lock);
      return NULL;
    }

    if(frame_elem == NULL || frame_elem == list_end(&frame_table))
frame_elem = list_begin(&frame_table); // Se chega no fim da lista, retorne
ao começo
    struct frame_table_entry* fte = list_entry(frame_elem, struct
frame_table_entry, list_elem); // Pega um frame da frame table

    frame_elem = list_next(frame_elem); // Avança para o próximo logo após
pegar o frame, já que tem possibilidade de free(fte)
```

```
    if(fte->pinned) continue; // Caso o frame seja utilizado no momento,
continue

    struct thread* curr = fte->owner;
    void *upage = fte->page->vaddr;


    if(pagedir_is_accessed(curr->pagedir, upage)){ // Verifica se foi
acessado recentemente
      pagedir_set_accessed(curr->pagedir, upage, false); // Coloca como
false(segunda chance)
    }
    else {
      (...)
    }

  }
}
```

Caso o frame não foi acessado recentemente, ele será o alvo para ser retirado da memória principal. Assim, surgem duas outras novas verificações: a página foi alterada? ou se o frame vem de uma página anônima? Nesses dois casos, a página deve ser salva na área de *swap*. Caso nenhuma dessas verificações sejam satisfeitas, isso significa que a página já está atualizada(não há discrepância com o que está armazenado no *swap*), assim, só é necessário retirar a página da *frame_table*.

Por fim, é feito uma limpeza do frame e ele é removido da _frame_table.

```
void *eviction(){
  lock_acquire(&frame_lock);

  while(true) {

    (...)

    if(pagedir_is_accessed(curr->pagedir, upage)){
      pagedir_set_accessed(curr->pagedir, upage, false);
    }
    else {
      if(fte->page->file == NULL || pagedir_is_dirty(curr->pagedir, upage))
{ // Caso de páginas anônimas ou frame dirty
        size_t index = swap_out(fte->frame);


        fte->page->index = index; // Salve o índice do bipmap na spt
        fte->page->in_memory=false;
        fte->page->swap=true;

      }
      else{ // Nesse else, é só retirar da memória
        fte->page->in_memory=false;
```

```
        }
        void *kpage = fte->frame;
        pagedir_clear_page(curr->pagedir, upage); // dá o page_clear
        list_remove(&fte->list_elem); // remove da frame_table
        free(fte);  // dá um free(fte)
        lock_release(&frame_lock);


        return kpage; // retorna o frame do table
      }

    }
}
```

Para colocar um *frame* na área de *swap*, temos o auxílio de uma função chamada `swap_out(void *kpage)`, definida em `swap.c`. Para encontrar posições livres na área de *swap*, utiliza-se um *bitmap*.

```
void
swap_init(void) {
    global_swap_block = block_get_role(BLOCK_SWAP); // Cria um bloco de
swap
    lock_init(&swap_lock);

    size_t total_sectors = block_size(global_swap_block);
    size_t total_slots = total_sectors / SECTORS_PER_PAGE;

    if (total_sectors % SECTORS_PER_PAGE != 0)
        PANIC("Swap block size not multiple of page sectors");

    swap_bitmap = bitmap_create(total_slots); // Cria um bitmap através do
total_slots(mapeia um índice para uma página)
    if (!swap_bitmap)
        PANIC("Failed to create swap bitmap");
}
void
write_from_block(uint8_t* frame, block_sector_t start_sector) {
    // esscreve no bloco: como cada índice é um início de uma página e cada
página tem 8 setores, então escreve num loop de 0 a 7
    for (size_t i = 0; i < SECTORS_PER_PAGE; ++i) {
        block_write(global_swap_block, start_sector + i,
                    frame + (i * BLOCK_SECTOR_SIZE));
    }
}

size_t
swap_out(void *kpage) {
    lock_acquire(&swap_lock);


    size_t slot = bitmap_scan_and_flip(swap_bitmap, 0, 1, false); // pega o
primeiro slot livre
```

```
    if (slot == BITMAP_ERROR) {
        lock_release(&swap_lock);
        PANIC("ESTOUROU O SWAP");
    }

    block_sector_t first_sector = (block_sector_t)(slot *
SECTORS_PER_PAGE);
    write_from_block((uint8_t*)kpage, first_sector); // escreve no bloco

    lock_release(&swap_lock);
    return slot;
}
```

A função `eviction()` é chamada sempre que um `palloc_get_page()` returna um `NULL`. Por exemplo, em *page_fault*:

```
static void
page_fault (struct intr_frame *f)
{
  (...)
  if(user){
      (...)
      if(not_present && spt_entry != NULL){
         //verifica se o endereço é válido
         void* kpage = palloc_get_page(PAL_USER);
         if(kpage == NULL){
            kpage = eviction();
         }
        (...)
      }

      else if(fault_addr >= f->esp - 32 && fault_addr < PHYS_BASE){ // Caso
em que não tem tabela na SPT(crescimento de pilha)
         void *kpage = palloc_get_page(PAL_USER|PAL_ZERO);
         if(kpage == NULL){
            kpage = eviction();
         }
         (...)
      }
      sys_exit(-1);

    (...)

    }
}
```

Reclamation

O *reclamation* é o processo inverso do *eviction*: um processo tenta acessar um endereço mapeado na área de swap e precisa trazê-lo de volta para a memória principal. Para isso, será utilizado o `spt_entry->index` para acessar a posição do bitmap, e, posteriormente, ler do `swap_block`.

```
void
read_from_block(uint8_t* frame, block_sector_t start_sector) {
    for (size_t i = 0; i < SECTORS_PER_PAGE; ++i) {
        block_read(global_swap_block, start_sector + i,
                   frame + (i * BLOCK_SECTOR_SIZE));
    }
}

void
reclamation(void *kpage, size_t slot) {
    block_sector_t first_sector = (block_sector_t)(slot *
SECTORS_PER_PAGE); // pega o setor em que está a página no swap
    lock_acquire(&swap_lock);

    read_from_block((uint8_t*)kpage, first_sector); // lê esse setor e o
coloca no kpage


    bitmap_set(swap_bitmap, slot, false); // coloca aquele setor como
livre(false)

    lock_release(&swap_lock);
}
```

Essa função é chamada sempre que há uma *spt_entry* indicando que a página está salva na área de swap. Por exemplo, em *page_fault*, a função é chamada nas seguintes condições:

```
static void
page_fault (struct intr_frame *f)
{

  (...)
  if(user){

      (...)

      if(not_present && spt_entry != NULL){
         //verifica se o endereço é válido
         void* kpage = palloc_get_page(PAL_USER);
         if(kpage == NULL){
            kpage = eviction();
         }

         if(spt_entry->swap){ // Caso em que a página está na área de swap
            reclamation(kpage, spt_entry->index); // Chama o reclamation,
passando a página para escrever e o índice
```

```
            spt_entry->swap = false;
            spt_entry->index = 0;

          }
        (...)
      }

      else if(fault_addr >= f->esp - 32 && fault_addr < PHYS_BASE){ // Caso
em que não tem tabela na SPT(crescimento de pilha)
        (...)
      }
      sys_exit(-1);
  }
  (...)
}
```

Parte 3: Memory Mapping

O grupo, infelizmente, não conseguiu finalizar a parte 3.

Últimos comentários

É importante ressaltar que não passamos em todos os testes.