Laboratorio 2 - Sistemas Operativos

—Integrantes—

Angela Jara < angela.jara@unmsm.edu.pe >

—Preliminares—

https://www.scs.stanford.edu/24wi-cs212/pintos/pintos.html
https://grail.eecs.csuohio.edu/~cis345s/PintosCSU_Ref.pdf

==========

ACCESO A MEMORIA Y SYSCALL WRITE

==========

—ALGORITHMS—

syscall_handler y verificación de acceso a memoria

```
static void
syscall_handler(struct intr_frame *f)
{
  static uint32_t *esp;
  esp = f->esp;

  if (!is_valid_ptr(esp) || !is_valid_ptr(esp + 1) ||
      !is_valid_ptr(esp + 2) || !is_valid_ptr(esp + 3))
  {
    exit(-1);
  }
  else
  {
    int syscall_number = *esp;
    switch (syscall_number)
    {
    case SYS_HALT:
      sys_halt();
      break;
    case SYS_EXIT:
      sys_exit(*(esp + 1));
      break;
    case SYS_OPEN:
      f->eax = sys_open((char *)*(esp + 1));
      break;
    case SYS_CLOSE:
      sys_close(*(esp + 1));
      break;
    case SYS_WRITE:
      f->eax = sys_write(*(esp + 1), (void *)*(esp + 2), *(esp + 3));
      break;
    default:
      break;
    }
  }
}
```

El manejador de llamadas al sistema (syscall_handler) identifica el tipo de syscall que se ejecutará según el número leído desde el stack del usuario (esp). Antes de procesar la syscall, valida que los punteros involucrados sean

seguros para evitar errores de segmentación. Si los punteros son válidos, ejecuta la función correspondiente al syscall.

```
bool is_valid_ptr(const void *usr_ptr)
{
  struct thread *cur = thread_current();
  if (is_valid_uvaddr(usr_ptr))
  {
    return (pagedir_get_page(cur->pagedir, usr_ptr)) != NULL;
  }
  return false;
}
```

Esta función valida si un puntero proporcionado por el usuario es seguro.

Comprueba que la dirección esté dentro del espacio de usuario y que esté

mapeada en el directorio de páginas actual. Si alguna condición no se cumple,

se considera inválido.

```
static bool
is_valid_uvaddr(const void *uvaddr)
{
  return (uvaddr != NULL && is_user_vaddr(uvaddr));
}
```

```
static inline bool
is_user_vaddr (const void *vaddr)
{
  return vaddr < PHYS_BASE;
}
```

Estas funciones verifican que las direcciones virtuales estén dentro del rango permitido para direcciones de usuario. PHYS_BASE define el límite entre el espacio de usuario y el espacio del kernel.

syscall write

```
int sys_write(int fd, const void *buffer, unsigned size)
{
```

```
  struct file_descriptor *fd_struct;
  int status = 0;

  unsigned buffer_size = size;
  void *buffer_tmp = buffer;

  while (buffer_tmp != NULL)
  {
    if (!is_valid_ptr(buffer_tmp))
      exit(-1);

    if (buffer_size > PGSIZE)
    {
      buffer_tmp += PGSIZE;
      buffer_size -= PGSIZE;
    }
    else if (buffer_size == 0)
    {
      buffer_tmp = NULL;
    }
    else
    {
      buffer_tmp = buffer + size - 1;
      buffer_size = 0;
    }
  }
  lock_acquire(&fs_lock);
  if (fd == STDIN_FILENO)
  {
    status = -1;
  }
  else if (fd == STDOUT_FILENO)
  {
    putbuf(buffer, size);
    status = size;
  }
  else
  {
    fd_struct = get_open_file(fd);
    if (fd_struct != NULL)
      status = file_write(fd_struct->file_struct, buffer, size);
  }
  lock_release(&fs_lock);

  return status;
}
```

La syscall write gestiona la escritura en archivos o la salida estándar.

Primero, verifica que las direcciones del buffer sean válidas, iterando por páginas si es necesario. Luego:

- Si el archivo destino es STDIN, no permite escribir.
- Si es STDOUT, utiliza putbuf para escribir en la consola.
- Para otros descriptores, intenta obtener el archivo asociado y escribe en él. Se asegura la concurrencia utilizando un candado (fs_lock) para evitar conflictos de acceso simultáneo.

—RATIONALE—

El diseño maneja adecuadamente punteros proporcionados por usuarios no confiables. Las verificaciones de acceso a memoria aseguran que todas las operaciones trabajen únicamente con direcciones válidas, mientras que el manejo de errores protege al kernel de comportamientos indeseados.

==========

PAGINACIÓN PARA SEGMENTOS CARGADOS DE FORMA *LAZY*

==========

—DATA STRUCTURES—

Supplied page table

```
struct thread
  {
   ...
    struct hash suppl_page_table;
   ...
  };
```

Se almacena como una tabla hash en la estructura thread. Esta tabla gestiona la información de las páginas virtuales que podrían estar respaldadas por almacenamiento, memoria compartida o swap.

```
enum suppl_pte_type
{
  SWAP = 001,
  FILE = 002,
  MMF  = 004
};


struct suppl_pte
{
  void *uvaddr;
  enum suppl_pte_type type;
```

```
  union suppl_pte_data data;
  bool is_loaded;
  size_t swap_slot_idx;
  bool swap_writable;
  struct hash_elem elem;
};

union suppl_pte_data
{
  struct
  {
    struct file *file;
    off_t ofs;
    uint32_t read_bytes;
    uint32_t zero_bytes;
    bool writable;
  } file_page;
};
```

Suppl Page Entry (suppl_pte): Representa la información de una entrada en la tabla de páginas complementarias, indicando detalles como la dirección virtual, el tipo de almacenamiento (swap, archivo, memoria compartida), y su estado de carga.

VM Frames

VM Frames (vm_frame): Representa un marco físico de memoria utilizado por un proceso, manteniendo información sobre el marco, el proceso propietario, la página de usuario asociada y un puntero de tabla de páginas.

```
struct list vm_frames;
```

```
struct vm_frame {
  void *frame;
  tid_t tid;
  uint32_t *pte;
  void *uva;
  struct list_elem elem;
};
```

Swap slots

Bitmap que controla el espacio de almacenamiento swap, representando la disponibilidad de cada posición de almacenamiento para swap.

```
static struct bitmap *swap_map;
```

—ALGORITHMS—

```
static bool
load_segment_lazy (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);

  while (read_bytes > 0 || zero_bytes > 0)
```

```
    {
      size_t page_read_bytes = read_bytes < PGSIZE ? read_bytes : PGSIZE;
      size_t page_zero_bytes = PGSIZE - page_read_bytes;

      if (!suppl_pt_insert_file (file, ofs, upage, page_read_bytes,
                                 page_zero_bytes, writable))
      return false;

      read_bytes -= page_read_bytes;
      zero_bytes -= page_zero_bytes;
      ofs += page_read_bytes;
      upage += PGSIZE;
    }
  return true;
}
```

Carga una sección de un archivo en la memoria virtual de manera *lazy*. Divide el segmento en páginas, valida su lectura, y usa suppl_pt_insert_file para insertar cada página en la *Supplied page table*.

```
bool suppl_pt_insert_file(struct file *file, off_t ofs, uint8_t *upage,
                          uint32_t read_bytes, uint32_t zero_bytes, bool writable) {
  struct suppl_pte *spte = calloc(1, sizeof *spte);
  if (spte == NULL) return false;

  spte->uvaddr = upage;
  spte->type = FILE;
  spte->data.file_page = (struct {
```

```
    struct file *file;
    off_t ofs;
    uint32_t read_bytes;
    uint32_t zero_bytes;
    bool writable;
  }){file, ofs, read_bytes, zero_bytes, writable};
  spte->is_loaded = false;

  struct hash_elem *result = hash_insert(&thread_current()->suppl_page_table,
&spte->elem);
  return result == NULL;
}
```

Crea una *entry* en la *Supplied page table* del thread. Asigna la dirección virtual, el archivo, el offset, los bytes de lectura, bytes para cero inicialización, y el permiso de escritura.

—RATIONALE—

La implementación de load_segment_lazy y suppl_pt_insert_file sigue un enfoque eficiente para la gestión de memoria virtual mediante carga perezosa (lazy loading), lo que permite cargar solo las páginas necesarias en el momento oportuno para optimizar el uso de recursos. Al separar la lógica de lectura de archivos en la memoria virtual a través de la tabla de páginas complementarias, se facilita el manejo de memoria y el acceso bajo demanda.

==========

STACK GROWTH

==========

—ALGORITHMS—

```
static void
page_fault (struct intr_frame *f)
{
  bool not_present;
  bool write;
  bool user;
  void *fault_addr;
  struct suppl_pte *spte;
  struct thread *cur = thread_current ();

  asm ("movl %%cr2, %0" : "=r" (fault_addr));

  intr_enable ();
  page_fault_cnt++;

  not_present = (f->error_code & PF_P) == 0;
  write = (f->error_code & PF_W) != 0;
  user = (f->error_code & PF_U) != 0;

  if (!not_present)
```

```
    exit (-1);

  if (fault_addr == NULL || !not_present || !is_user_vaddr(fault_addr))
    exit (-1);

  spte = get_suppl_pte (&cur->suppl_page_table, pg_round_down(fault_addr));

  if (spte != NULL && !spte->is_loaded) {
    load_page (spte);
  }
  else if (spte == NULL && fault_addr >= (f->esp - 32) &&
           (PHYS_BASE - pg_round_down (fault_addr)) <= STACK_SIZE) {
    grow_stack (fault_addr);
  }
  else {
    if (!pagedir_get_page (cur->pagedir, fault_addr))
      exit (-1);

    printf ("Page fault at %p: %s error %s page in %s context.\n",
            fault_addr,
            not_present ? "not present" : "rights violation",
            write ? "writing" : "reading",
            user ? "user" : "kernel");
    kill (f);
  }
}
```

La función page_fault se encarga de manejar las interrupciones de página que ocurren cuando un proceso intenta acceder a una dirección de memoria que no está actualmente en la memoria física. Primero, obtiene la dirección que causó el fallo de página mediante el registro cr2. Luego, verifica el tipo de fallo para determinar si es un error de escritura, lectura o un fallo por ausencia de página. Si el acceso es válido, busca la entrada correspondiente en la tabla de páginas complementarias (suppl_page_table) para ver si la página necesita ser cargada. Si la dirección es cercana al límite de la pila, intenta expandirla dinámicamente con la función grow_stack. Si no es válido, termina el proceso para evitar errores de acceso.

```
void grow_stack(void *uvaddr) {
  struct thread *t = thread_current();
  void *spage = vm_allocate_frame(PAL_USER | PAL_ZERO);

  if (spage == NULL) return;

  if (!pagedir_set_page(t->pagedir, pg_round_down(uvaddr), spage, true)) {
    vm_free_frame(spage);
  }
}
```

La función grow_stack extiende la pila dinámicamente cuando un acceso fuera de límites es válido, como parte del diseño de la arquitectura de memoria virtual. Intenta asignar un nuevo marco de memoria física y lo vincula con la dirección virtual para permitir el acceso. En caso de fallo durante la asignación de memoria, la función finaliza sin hacer cambios, evitando posibles errores. Si la asignación es exitosa, la dirección es configurada en la tabla de páginas para reflejar su estado cargado.

—RATIONALE—

El diseño de las funciones page_fault y grow_stack es eficiente y modular, ya que maneja los fallos de página verificando condiciones críticas, como la validez de accesos, expansión dinámica de la pila y carga de páginas bajo demanda. Esto mejora la robustez del sistema de gestión de memoria al manejar adecuadamente casos de acceso inválidos y situaciones de crecimiento de

memoria.