Respuestas a Preguntas Argumentativas

Laboratorio 1: Memoria Virtual

Pregunta 1: Manejo de Fallos de Página

¿Qué es un fallo de página?

Un **fallo de página** (page fault) es una excepción o trampa que ocurre cuando un proceso intenta acceder a una página virtual que no está actualmente presente en la memoria física (RAM). Es un evento normal y esperado en sistemas de memoria virtual con paginación bajo demanda.

Causas comunes de fallos de página:

1. **Página no cargada**: La página nunca ha sido traída del disco a RAM
2. **Página desalojada**: La página estuvo en RAM pero fue reemplazada por otra
3. **Primera referencia**: Primera vez que el proceso accede a esa dirección
4. **Página inválida**: Intento de acceder a memoria no asignada al proceso

¿Qué hace el SO (y nuestra simulación) cuando ocurre un fallo de página?

Pasos del Sistema Operativo:

1. **Detección**: La MMU detecta que el bit `present` en la entrada de la tabla de páginas es `False` y genera una interrupción
2. **Validación**: El SO verifica su tabla interna para determinar si la referencia de memoria es válida:
   - Si es inválida → Segmentation Fault (termina el proceso)
   - Si es válida → Procede con la carga
3. **Búsqueda de marco libre**:
   - Si hay marcos disponibles → Asigna uno
   - Si no hay marcos → Ejecuta algoritmo de reemplazo de páginas
4. **Reemplazo (si es necesario)**:
   - Selecciona una página víctima usando el algoritmo (FIFO en nuestro caso)
   - Si la víctima está sucia (`dirty=True`) → Escribe de vuelta al disco
   - Libera el marco de la víctima
5. **Carga de página**:

- Lee la página requerida desde el disco al marco asignado
- Actualiza la tabla de páginas: `present=True`, `frame=marco_asignado`, `dirty=False`

6. **Reanudación**: El proceso continúa su ejecución desde la instrucción que causó el fallo

En nuestra simulación:

Nuestro método `_ensure_in_ram()` implementa este comportamiento:

python

```python
def _ensure_in_ram(self, page_no: int) -> None:
    entry = self.page_table.get_entry(page_no)

    if entry.present:
        return  # No hay fallo

    # PAGE FAULT detectado
    frame_no = self.physical_memory.allocate_frame()

    if frame_no is None:
        # Reemplazo FIFO
        victim_page = self.fifo_queue.pop(0)
        victim_entry = self.page_table.get_entry(victim_page)

        if victim_entry.dirty:
            # Write-back
            self.backing_store[victim_page] = copy(frames[victim_frame])

        # Liberar marco
        physical_memory.free_frame(victim_frame)
        frame_no = physical_memory.allocate_frame()

    # Cargar página
    if page_no in self.backing_store:
        frames[frame_no] = copy(backing_store[page_no])
    else:
        frames[frame_no] = bytearray(PAGE_SIZE)  # Nueva página

    # Actualizar tabla
    entry.present = True
    entry.frame = frame_no
    entry.dirty = False
```

Importancia:

Los fallos de página permiten la **paginación bajo demanda** (demand paging): solo se cargan en RAM las páginas que realmente se necesitan, optimizando el uso de memoria.

Pregunta 2: Bit Sucio (Dirty Bit)

¿Por qué se necesita el bit sucio?

El **dirty bit** (bit de modificación) es esencial para optimizar el rendimiento del sistema de memoria virtual. Se necesita por las siguientes razones:

1. Optimización de E/S:

- Escribir al disco es una operación muy costosa (miles de veces más lenta que RAM)
- Si una página no fue modificada, su copia en disco está actualizada
- No tiene sentido escribir de vuelta algo que no cambió
- **Ahorro**: Reduce el tiempo de E/S a la mitad para páginas no modificadas

2. Identificación de cambios:

- Permite al SO distinguir entre:
  - Páginas **limpias**: Datos idénticos a la copia en disco
  - Páginas **sucias**: Datos modificados que difieren del disco

3. Decisiones de reemplazo:

- Algunos algoritmos avanzados prefieren desalojar páginas limpias primero
- Es más rápido reemplazar una página limpia que una sucia

¿Qué sucede si desalojamos una página sucia?

Consecuencias si el dirty bit está activo (`dirty=True`):

1. **Write-back obligatorio**: El SO DEBE escribir la página al backing store antes de liberar el marco
2. **Latencia adicional**: El fallo de página que causó el reemplazo se demora más
3. **Preservación de datos**: Los cambios del proceso se guardan en disco
4. **Consistencia**: El backing store se mantiene sincronizado con las modificaciones

Ejemplo en código:

```python
if victim_entry.dirty:
    # Página modificada - DEBE escribirse
    self.backing_store[victim_page] = bytearray(
        self.physical_memory.frames[victim_frame]
    )
    self.write_backs += 1  # Operación costosa
else:
    # Página limpia - simplemente descartar
    pass  # Sin E/S necesaria
```

¿Qué pasaría si no rastreáramos la suciedad?

Escenario catastrófico sin dirty bit:

Opción 1: Escribir siempre (conservador)

- **Problema**: Duplicamos el tiempo de reemplazo innecesariamente
- Cada desalojo requiere escritura al disco, incluso si la página nunca cambió
- Impacto: Sistema 50% más lento en promedio
- Desgaste innecesario del disco/SSD

Opción 2: Nunca escribir (optimista)

- **PROBLEMA CRÍTICO**: ¡Pérdida de datos!
- Las modificaciones del proceso se pierden cuando la página es desalojada
- Si la página se recarga, aparecen los datos antiguos del disco
- **Resultado**: Corrupción de datos, comportamiento impredecible, crashes

Ejemplo de fallo sin dirty bit:

```python
# Sin dirty bit - opción 2 (desastrosa)
vm.write_byte(100, 42)  # Modificar página 0
# ... llenar RAM con otras páginas ...
# Página 0 desalojada SIN write-back (perdemos el cambio)
value = vm.read_byte(100)  # Página 0 recargada desde disco
# ¡value es 0, no 42! - PÉRDIDA DE DATOS
```

Conclusión:

El dirty bit es absolutamente esencial. Sin él, debemos elegir entre:

- Rendimiento pésimo (write-back siempre)
- Pérdida de datos (write-back nunca)

No hay alternativa razonable. El dirty bit es un ejemplo perfecto de cómo un solo bit de metadata puede tener un impacto masivo en el rendimiento y correctitud del sistema.

Pregunta 3: Reemplazo FIFO

¿Cuál es una fortaleza de FIFO?

Fortaleza principal: Simplicidad y eficiencia

FIFO es extremadamente simple de implementar y tiene un overhead computacional mínimo:

Ventajas específicas:

1. **Implementación trivial:**
   - Solo requiere una cola (o lista)
   - O(1) para seleccionar víctima: `queue.pop(0)`
   - O(1) para agregar página: `queue.append(page)`
   - Sin estructuras de datos complejas
2. **Overhead mínimo:**
   - No necesita rastrear información adicional (tiempos de acceso, contadores)
   - No requiere actualización en cada acceso a memoria
   - No depende de hardware especial (bits de referencia)
3. **Predecible y determinista:**
   - Fácil de analizar y depurar
   - Comportamiento completamente predecible
   - Útil para sistemas con requisitos de tiempo real
4. **Justo:**
   - Todas las páginas tienen "tiempo igual" en RAM
   - No discrimina basado en importancia percibida
   - Evita inanición de páginas

Comparación con LRU:

- FIFO: Una simple cola, sin tracking de accesos
- LRU: Requiere actualizar timestamp/posición en CADA acceso a memoria
- FIFO: ~10 instrucciones por reemplazo
- LRU: ~100+ instrucciones si se implementa correctamente

Código ilustrativo:

```python
# FIFO - Selección O(1)
victim = self.fifo_queue.pop(0)  # Trivial

# LRU - Requiere búsqueda O(n)
# O estructura de datos compleja (heap, doubly-linked list)
victim = min(pages, key=lambda p: p.last_access_time)
```

¿Cuál es una limitación de FIFO comparado con LRU?

Limitación principal: Ignora patrones de uso

FIFO reemplaza páginas basándose únicamente en el orden de llegada, ignorando completamente cuán frecuentemente se usan las páginas.

Problema específico:

1. **Anomalía de Belady:**
   - Fenómeno contraintuitivo donde agregar más RAM puede aumentar los page faults
   - Ejemplo: Con 3 marcos puede tener menos fallos que con 4 marcos
   - Solo ocurre en FIFO (LRU no sufre esto)
2. **Páginas "calientes" desalojadas:**
   - Una página usada constantemente puede ser desalojada por ser antigua
   - Páginas raramente usadas pueden quedarse en RAM por ser recientes
   - No refleja el "working set" real del proceso
3. **Rendimiento subóptimo:**
   - Tasa de page faults típicamente 20-50% mayor que LRU
   - Especialmente malo con patrones de acceso no uniformes

- Ignora localidad temporal de referencias

Ejemplo concreto del problema:

```python
# Secuencia de acceso: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
# Marcos: 3

# FIFO:
# [1] [2] [3] [4] [1*] [2*] [5] [1*] [2*] [3*] [4*] [5*]
# F   F   F   F   F    F    F   F    F    F    F    -
# 11 page faults - Páginas 1,2 desalojadas repetidamente

# LRU (óptimo):
# [1] [2] [3] [4] [1] [2] [5] [1] [2] [3] [4] [5]
# F   F   F   F   -   -   F   -   -   F   F   -
# 8 page faults - Retiene páginas frecuentes (1,2)
```

Comparación de tasas de fallo:

Algoritmo	Complejidad	Fallos típicos	Overhead
FIFO	O(1)	Alto (~100%)	Mínimo
LRU	O(n) o O(log n)	Bajo (~60%)	Alto
Óptimo	Imposible	Mínimo (~40%)	N/A

Trade-off fundamental:

```
FIFO: Simplicidad + Velocidad <-> Mala tasa de aciertos
LRU:  Complejidad + Lentitud <-> Buena tasa de aciertos
```

Conclusión:

FIFO es ideal para:

- Sistemas pequeños con recursos limitados
- Prototipos y educación
- Casos donde la simplicidad es prioritaria

LRU es mejor para:

- Sistemas de producción con carga real

- Aplicaciones donde el rendimiento es crítico
- Workloads con localidad temporal fuerte

En nuestro laboratorio, FIFO es perfecto para aprender los conceptos fundamentales antes de abordar algoritmos más sofisticados.

Pregunta 4: Aislamiento

¿Cómo ayuda la memoria virtual a aislar procesos?

La memoria virtual es fundamental para el **aislamiento de procesos**, que es un pilar de la seguridad y estabilidad de los sistemas operativos modernos.

Mecanismos de aislamiento:

1. Espacios de direcciones separados:

- Cada proceso tiene su propia tabla de páginas
- Las direcciones virtuales de un proceso son completamente independientes de otros
- Dos procesos pueden usar la misma dirección virtual (ej: 0x1000) sin conflicto
- La MMU traduce usando la tabla del proceso actual

2. Mapeo independiente:

```
Proceso A:          Proceso B:
Virt → Físico       Virt → Físico
0x00 → Frame 5      0x00 → Frame 2
0x01 → Frame 8      0x01 → Frame 9
0x02 → Frame 1      0x02 → Frame 7

Mismas direcciones virtuales → Diferentes marcos físicos
```

3. Imposibilidad de acceso cruzado:

- Un proceso NO puede construir una dirección virtual que apunte a marcos de otro
- La tabla de páginas solo mapea sus propias páginas
- Intentar acceder a memoria no mapeada → Page Fault → Segmentation Fault

4. Protección por hardware:

- La MMU verifica permisos en cada acceso (lectura/escritura/ejecución)

- Bits de protección en cada PTE (no implementados en nuestro simulador)
- Modo kernel vs. modo usuario (páginas del SO inaccesibles a procesos)

Beneficios del aislamiento:

1. **Seguridad:**
   - Un proceso malicioso no puede leer secretos de otro
   - No puede modificar datos de otros procesos
   - No puede ejecutar código en memoria ajena
2. **Estabilidad:**
   - Un bug (buffer overflow) en un proceso no corrompe otros
   - Crashes localizados → el SO y otros procesos continúan
   - Más fácil depurar: errores de memoria son detectables
3. **Simplicidad para programadores:**
   - Cada proceso ve un espacio de direcciones limpio desde 0
   - No necesitan preocuparse de dónde están otros procesos
   - Programación como si tuvieran toda la memoria

¿Cómo se manejarían múltiples procesos aunque nuestra VM solo soporte uno?

Extensión teórica de nuestro simulador:

Aunque nuestra implementación actual solo soporta un proceso, extenderla a múltiples procesos requeriría los siguientes cambios:

1. Tabla de páginas por proceso:

```python
class VM:
    def __init__(self):
        # En lugar de una sola tabla:
        # self.page_table = PageTable()

        # Diccionario de tablas: PID → PageTable
        self.page_tables: Dict[int, PageTable] = {}

        # Proceso actualmente ejecutándose
        self.current_pid: Optional[int] = None
```

2. Cambio de contexto (Context Switch):

Cuando el SO cambia de un proceso a otro, debe:

```python
def switch_process(self, old_pid: int, new_pid: int) -> None:
    """
    Cambio de contexto entre procesos.
    Simula lo que hace el SO al cambiar procesos.
    """
    # Guardar estado del proceso antiguo
    # (ya está en self.page_tables[old_pid])

    # Cambiar a la tabla de páginas del nuevo proceso
    self.current_pid = new_pid

    # En hardware real: CPU carga CR3 con dirección de nueva tabla
    # Aquí simplemente cambiamos qué tabla usamos

    # La MMU ahora traduce usando page_tables[new_pid]
```

3. Métodos de acceso a memoria modificados:

```python
def read_byte(self, vaddr: int) -> int:
    # Obtener tabla del proceso actual
    page_table = self.page_tables[self.current_pid]

    # Traducción usando tabla del proceso actual
    page_no = vaddr // PAGE_SIZE
    offset = vaddr % PAGE_SIZE

    entry = page_table.get_entry(page_no)
    # ... resto igual ...
```

4. FIFO queue por proceso o global:

Dos opciones de diseño:

Opción A: FIFO local (por proceso)

```python
```

```
self.fifo_queues: Dict[int, List[int]] = {}
# Cada proceso tiene su propia cola FIFO
# Reemplazo solo entre páginas del mismo proceso
```

Opción B: FIFO global (entre todos)

python

```
self.global_fifo_queue: List[Tuple[int, int]] = []  # (pid, page_no)
# Reemplazo considera páginas de todos los procesos
# Más justo pero más complejo
```

5. Compartición de páginas (opcional):

Para eficiencia, algunos marcos pueden ser compartidos:

python

```
# Página de código compartida (ej: libc)
shared_frame = 10

# Proceso A y B mapean a mismo marco
self.page_tables[pid_A].get_entry(5).frame = shared_frame
self.page_tables[pid_B].get_entry(7).frame = shared_frame

# Ambos leen el mismo código, ahorrando RAM
```

6. Ejemplo completo de uso multi-proceso:

python

```
vm = VM()

# Crear proceso A
pid_a = vm.create_process()
vm.switch_process(None, pid_a)
vm.write_byte(100, 42)  # Escribe en espacio de A

# Crear proceso B
pid_b = vm.create_process()
vm.switch_process(pid_a, pid_b)
vm.write_byte(100, 99)  # Escribe en espacio de B (diferente marco)

# Volver a A
vm.switch_process(pid_b, pid_a)
value = vm.read_byte(100)  # Lee 42 (no 99!)
# ¡Aislamiento completo!
```

Desafíos adicionales en implementación real:

1. **Asignación de memoria física:**
   - ¿Cuántos marcos por proceso?
   - Políticas de fairness
2. **Copy-on-Write:**
   - fork() comparte páginas inicialmente
   - Se copian solo al escribir
3. **Páginas compartidas:**
   - Bibliotecas compartidas (libc, libssl)
   - Memoria compartida explícita (IPC)
4. **TLB flush:**
   - Al cambiar proceso, el TLB debe invalidarse
   - O usar etiquetas ASID para identificar proceso

Conclusión:

El aislamiento de procesos es posible porque:

- Cada proceso tiene su tabla de páginas privada
- La MMU usa la tabla del proceso actual para traducir

- Es imposible construir direcciones que accedan a otros procesos
- El hardware aplica protecciones en cada acceso

Nuestra VM simplificada demuestra los conceptos core, y extenderla a múltiples procesos sería principalmente:

- Replicar estructuras (una tabla por proceso)
- Implementar cambio de contexto
- Gestionar competencia por marcos físicos

El principio fundamental permanece: **separación de espacios de direcciones virtuales = aislamiento garantizado por hardware**.