Eduardo Prasniewski

1 Introdução

O presente trabalho busca simular o gerenciamento de memória utilizando TLB e os seguintes algoritmos de substituição de página: LRU e Segunda chance.

2 Desenvolvimento

Primeiramente foi necessário ler os arquivos e recuperar apenas os endereços, pois a entrada de escrita W e leitura R não são usadas. As primeiras linhas realizam exatamente isso. Logo após é instanciado um objeto da classe MMU com o algoritmo LRU para substituição de páginas, depois para cada endereço virtual obtido é feita a tradução para endereço físico. Por fim, as operações em disco e os acertos e erros na TLB são mostrados na tela.

Código 1: Código inicial

```
# Reading traces
def read_trace(filename):
    with open(filename, 'r') as file:
        txts = file.readlines()
    return np.array([x.split() for x in txts])

traces = read_trace('traces/bzip.trace').T
address = traces[0]

# Instance MMU object
mmu = MMU(method = LRU)

for adr in address:
    p = mmu.translate_addr(adr)

print(f'Opera  es em disco: {mmu.page_table.disk_op_counter}')
mmu.tlb.display_hits()
```

A classe MMU (Código 2) propõe a implementação do fluxograma apresentado no enunciado da prática. Possuindo apenas um método, para traduzir endereços virtuais (páginas) em endereços físicos (quadros/frames) utilizando um dos dois métodos de substituição de páginas.

Primeiramente separa os bits de página e deslocamento, depois procura na TLB caso esteja em cache (contabilizando um acerto na TLB), senão realiza uma busca na tabela de páginas, atualizando a TLB logo após (contabilizando um erro na TLB).

A classe da TLB (Código 3) realiza as operações necessárias para o cache entre as relações de páginas e quadros. A intenção foi criar uma classe independente do método de substituição de páginas, sendo assim é passado como paramêtro na inicialização da objeto, o mesmo ocorre com a classe da tabela de páginas (PageTable).

PageTable, ou a classe da tabela de páginas (Código 4) é similar a TLB, levando em consideração que possui uma função para gerar memórias físicas virtuais, a fim de simulação.

Código 2: MMU

```python
# MMU class
class MMU():
    def __init__(self, method) -> None:
        self.tlb = TLB(method=method)
        self.page_table = PageTable(method=method)

    def translate_addr(self, adr) -> list:
        p, d = adr[:5], adr[5:]

        # TLB ops
        frame = self.tlb.search_tlb(p)
        if frame == None:
            self.tlb.count_tlb_miss()
            frame = self.page_table.search_page_table(p)
            self.tlb.update(p, frame)
        else:
            self.tlb.count_tlb_hit()
        return frame, d
```

Código 3: TLB

```python
# TLB class
class TLB():
    TLB_SIZE = 16
    def __init__(self, method) -> None:
        self.tlb = {}
        self.hit = 0
        self.miss = 0
        self.alg_page_fault = method()

    def display_hits(self):
        print(f'TLB MISS: {self.miss}')
        print(f'TLB HIT: {self.hit}')

    def count_tlb_hit(self):
        self.hit+=1

    def count_tlb_miss(self):
        self.miss+=1

    def search_tlb(self, p) -> str:
        f = self.tlb.get(p)
        if f != None:
            self.alg_page_fault.update(p)
        return f

    def update(self, p, f):
        if len(self.tlb) == self.TLB_SIZE:
            removed_page, _ = self.alg_page_fault.page_fault(self.tlb)
            del self.tlb[removed_page]
        self.tlb[p] = f
        self.alg_page_fault.new(p)
```

Código 4: PageTable - Tabela de páginas

```python
class PageTable():
    PAGE_TABLE_SIZE = 64
    def __init__(self, method) -> None:
        self.page_table = {}
        self.disk_op_counter = 0
        self.alg_page_fault = method()
        self.free_mem = self.__gen_virtual_mem()

    def __gen_virtual_mem(self) -> list:
        page_size = 3 * 1024
        addrs = []
        max_addrs = self.PAGE_TABLE_SIZE
        for i in range(max_addrs):
            addr = i * page_size
            addrs.append(f'0x{addr:05X}')
        return addrs

    def search_page_table(self,p):
        x = self.page_table.get(p)
        # page not found at memory
        if x == None:
            self.disk_op_counter+=1
            # in case any frame is available
            if len(self.free_mem) > 0:
                x = self.free_mem.pop()
                self.page_table[p] = x
                self.alg_page_fault.new(p)
                return x
            else:
                removed_page, addr_available =
                    self.alg_page_fault.page_fault(self.page_table)
                self.alg_page_fault.new(p)
                del self.page_table[removed_page]
                self.page_table[p] = addr_available
                return addr_available
        else:
            self.alg_page_fault.update(p)
            return x
```

Como já foi comentado, os algoritmos de substituição são independentes das tabelas, sendo assim é criada uma classe para cada algoritmo, e eles seguem a mesma nomenclatura de métodos (posteriormente isto pode ser atualizado para uma classe abstrata, mas para via de praticidade inicial, esta estratégia foi adotada).

Os algoritmos são bem similares entre si, ambos possuem uma lista, a diferença é no manejo de cada: enquanto a LRU (Código 5) vai atualizando o vetor de uma dimensão a cada vez que uma página é utilizada, o Segunda chance (Código 6), agora de duas dimensões, possue um bit para tolerância de uma falha inicial, utilizando um próximo elemento da fila como página a ser substituída.

Código 5: LRU

```python
# LRU implementation class
class LRU():
    def __init__(self) -> None:
        self.queue = []

    def new(self, page, name = 'pt'):
        self.queue.insert(0, page)

    def page_fault(self, table):
        # verify addr that will be removed
        pages = list(table.keys())
        physical_addr = list(table.values())
        removed_page =  self.queue.pop()
        # get page that will be removed
        idx = pages.index(removed_page)
        addr_available = physical_addr[idx]
        return removed_page, addr_available

    def update(self, x) -> None:
        idx = self.queue.index(x)
        self.queue = [x] + self.queue[:idx] + self.queue[idx+1:]
```

3 Resultados

A seguir os resultados da implementação. É possível notar uma ligeira melhor performance do Segunda chance, possuindo menos operações em disco do que o LRU e também menos acessos a TLB. Vale ressaltar que tanto a TLB quando a tabela de páginas usam o mesmo algoritmo.

Arquivo	Operações em Disco	TLB miss	TLB hit
bzip.trace	1283	3344	996656
gcc.trace	60012	116604	883396
sixpack.trace	40599	108682	891318
swim.trace	21426	171961	828039

Tabela 1: Resultados usando LRU

Arquivo	Operações em Disco	TLB miss	TLB hit
bzip.trace	1275	3418	996582
gcc.trace	58368	115520	884480
sixpack.trace	39358	105942	894058
swim.trace	21295	160172	8398288

Tabela 2: Resultados usando Segunda chance

Código 6: Second Chance

```python
# Second Chance implementation class
class SecondChance():
    def __init__(self) -> None:
        self.queue = []

    def new(self, page):
        self.queue.insert(0, [page, 1])

    def get_removed_page(self):
        for value in self.queue:
            p, b = value
            if b == 0:
                self.queue.remove(value)
                return p
            # shift queue
            self.queue = self.queue[1:]
            self.queue.append([p, 0])
        # if there is no 0 bit, return the first
        p = self.queue[0][0]
        del self.queue[0]
        return p

    def page_fault(self, table):
        # verify page that will be removed
        pages = list(table.keys())
        physical_addr = list(table.values())
        removed_page = self.get_removed_page()

        # get index that will be removed
        idx = pages.index(removed_page)
        addr_available = physical_addr[idx]
        return removed_page, addr_available

    def update(self, x):
        for idx, value in enumerate(self.queue):
            p, b = value
            if p == x:
                self.queue[idx] = [p, 1]
```