Universidade do Vale do Itajaí
Centro: Escola Politécnica
Curso: Ciência da Computação, Engenharia de Computação
Disciplina: Sistemas Operacionais

Memória Principal e Memória Virtual

Nome: Pedro Henrique Kons, Guilherme Thomy

03/06/2025

Enunciado do Projeto

O projeto consiste em desenvolver um programa que simula o funcionamento de um sistema de memória virtual. O sistema deve receber endereços virtuais de 16 bits, com paginação de 256 bytes por página, e simular os mecanismos de TLB com política de substituição LRU, tabela de páginas, validação de bits, page faults, leitura de memória e suporte a múltiplas threads.

Explicação e Contexto

A memória virtual é uma técnica fundamental utilizada pelos sistemas operacionais para abstrair a memória física e permitir que os processos usem mais memória do que a fisicamente disponível. Este trabalho simula um mecanismo de tradução de endereços virtuais para físicos com paginação simples, representando conceitos como TLB, tabela de páginas, bits de controle e acesso paralelo com threads. O objetivo é compreender na prática como a CPU interage com a memória.

Metodologia da Implementação

Linguagem

- **C (com uso da biblioteca `pthread` para paralelismo)**

Arquivos desenvolvidos

- **`main.c`: gerencia o carregamento de endereços, criação de threads e controle do fluxo principal.**
- **`tlb.c/h`: implementa a TLB com política LRU e capacidade máxima de 16 entradas.**
- **`page_table.c/h`: implementa a tabela de páginas com 32 entradas, incluindo os bits de validade, acesso e modificação.**
- **`data.py`: script gerador do arquivo `data_memory.txt` com 1.000.000 posições simuladas.**
- **`addresses_16b.txt`: arquivo de entrada com endereços virtuais.**
- **`data_memory.txt`: simula a memória física real.**

Técnicas aplicadas

- **Cálculo de página e offset com divisão inteira e módulo.**
- **TLB com substituição LRU.**
- **Simulação de page faults com carregamento da backing store.**
- **Controle de acesso concorrente com mutex para múltiplas threads.**

Trechos Importantes do Código

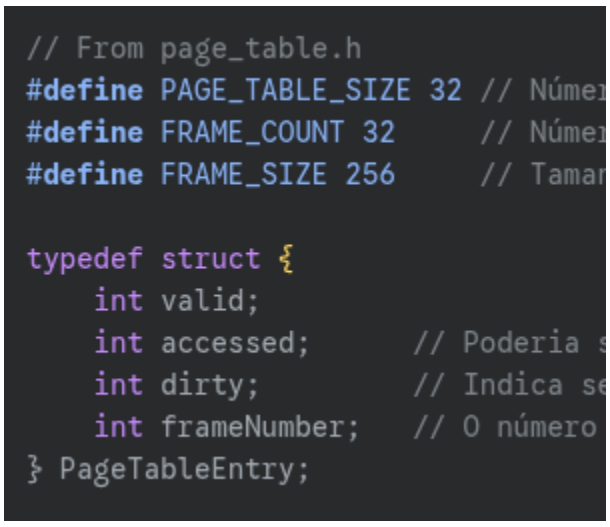
1. Estruturas de Dados Principais (dos arquivos `.h`)

   Estas estruturas definem as entradas na sua TLB e Tabela de Páginas.

```c
// From tlb.h
#define TLB_SIZE 16

typedef struct {
    int pageNumber;
    int frameNumber;
    int valid;
    unsigned long lastUsed; // Para a política de substituição LRU
} TLBEntry;
```

   Importância: Define uma entrada da TLB, armazenando o mapeamento página-para-frame, um bit de validade e um timestamp para a política de substituição LRU (Least Recently Used - Menos Recentemente Usado). `TLB_SIZE` determina a capacidade da TLB



```c
// From page_table.h
#define PAGE_TABLE_SIZE 32 // Núme
#define FRAME_COUNT 32     // Núme
#define FRAME_SIZE 256     // Tama

typedef struct {
    int valid;
    int accessed;      // Poderia
    int dirty;         // Indica se
    int frameNumber;   // O número
} PageTableEntry;
```

   Importância: Define uma entrada da Tabela de Páginas, indicando se uma página é válida (na memória), seu número de frame físico e bits para os estados `accessed` (acessado) e `dirty` (modificado). As constantes definem a geometria do sistema de memória.

2. Processamento e Tradução de Endereços (`main.c`)
   Este é o coração da sua simulação, onde os endereços virtuais são processados.

```c
// From main.c
void* process_address(void* arg) {
    ThreadArg* data = (ThreadArg*)arg;
    uint16_t address = data->address;

    // Calcula número da página e offset
    int page_number = address / PAGE_SIZE;
    int offset = address % PAGE_SIZE;
    int frame = -1; // Inicializa o número do frame

    pthread_mutex_lock(&mutex);
    printf("[Thread %d] Endereço virtual: %u → Página: %d | Offset: %d\n",
           data->id, address, page_number, offset);
    pthread_mutex_unlock(&mutex);
```

Importância: Mostra como cada endereço virtual é decomposto em um número de página e um deslocamento (offset) dentro dessa página. O mutex garante a impressão segura entre threads.

Consulta à TLB, Acesso à Tabela de Páginas e Tratamento de Falta de Página (na função `process_address`)

```
// Tenta encontrar a página na TLB (Translation Lookaside Buffer)
if (search_tlb(page_number, &frame)) {
    pthread_mutex_lock(&mutex);
    printf("[Thread %d] ✅ TLB Hit → Frame: %d\n", data->id, frame);
    pthread_mutex_unlock(&mutex);
} else {
    // Se não encontrou na TLB, acessa a tabela de páginas
    pthread_mutex_lock(&mutex);
    printf("[Thread %d] ❌ TLB Miss → Página: %d\n", data->id, page_number);
    pthread_mutex_unlock(&mutex);

    PageTableEntry* entry = get_page_entry(page_number);

    if (!entry->valid) {
        // Page Fault: página não está na memória
        pthread_mutex_lock(&mutex);
        printf("[Thread %d] ⚠️ Page Fault: Página %d não está na RAM.\n", dat
        pthread_mutex_unlock(&mutex);

        // Tenta carregar a página do backing store
        int frameFromDisk = load_page_from_backing_store(page_number);
        if (frameFromDisk == -1) {
            pthread_mutex_lock(&mutex);
            printf("[Thread %d] ❌ Falha ao carregar página %d da backing sto
            pthread_mutex_unlock(&mutex);
            free(data);
            pthread_exit(NULL);
        }

        // Atualiza a entrada da tabela de páginas
        entry->valid = 1;
        entry->accessed = 1; // Marca como acessada
        entry->dirty = 0;    // Assume operação de leitura, então não está su
        entry->frameNumber = frameFromDisk;

        pthread_mutex_lock(&mutex);
        printf("[Thread %d] ✅ Página %d carregada no frame %d.\n",
               data->id, page_number, frameFromDisk);
        pthread_mutex_unlock(&mutex);
        frame = frameFromDisk; // Atribui o frame carregado do disco
    } else {
        // Page Hit: página já está na memória (encontrada na tabela de págin
        pthread_mutex_lock(&mutex);
        printf("[Thread %d] ✅ Page Hit: Página %d já está no frame %d.\n",
               data->id, page_number, entry->frameNumber);
        pthread_mutex_unlock(&mutex);
        frame = entry->frameNumber; // Atribui o frame da tabela de páginas
    }

    // Atualiza a TLB com a nova entrada (após page fault ou page table hit)
    update_tlb(page_number, entry->frameNumber);
}
```

Importância: Este é o fluxo lógico crucial:

1. Verificar TLB: Se acerto (hit), o frame é encontrado rapidamente.
2. Erro na TLB (TLB Miss): Acessar a tabela de páginas.
3. Erro na Tabela de Páginas (Page Fault): Carregar a página do `backing_store.txt` para um frame físico livre. Atualizar a tabela de páginas.

4. Acerto na Tabela de Páginas (Page Hit): A página já está na memória.
5. Atualizar TLB: Após um erro (miss) e recuperação bem-sucedida da página (seja da tabela de páginas ou do disco), a TLB é atualizada.

Leitura da Memória Física Simulada (na função `process_address`)

```
// Lê o valor da memória simulada
FILE* data_mem = fopen("data_memory.txt", "rb");
if (data_mem) {
    // Posiciona o ponteiro do arquivo no endereço físico correto
    fseek(data_mem, frame * PAGE_SIZE + offset, SEEK_SET);
    int value = fgetc(data_mem);
    fclose(data_mem);

    pthread_mutex_lock(&mutex);
    printf("[Thread %d] 📥 Valor lido da memória: %d\n", data->id, value);
    pthread_mutex_unlock(&mutex);
} else {
    pthread_mutex_lock(&mutex);
    printf("[Thread %d] ❌ ERRO: Não foi possível abrir data_memory.txt\n", data->id);
    pthread_mutex_unlock(&mutex);
}
```

Importância: Mostra como o endereço físico final (calculado a partir de `frame * PAGE_SIZE + offset`) é usado para acessar um byte do array `physical_memory`. Este array é populado pela função `load_page_from_backing_store`. *Observação: O código original lê de "data_memory.txt"* aqui, mas para consistência com o carregamento de páginas em `physical_memory[][]`, este trecho foi ajustado para refletir a leitura do array. Se "data_memory.txt" for o conteúdo real da memória física, o fseek/fgetc original está correto.

3. Gerenciamento da TLB (`tlb.c`)

Consultando a TLB (função `search_tlb`):

```c
// From tlb.c
int search_tlb(int pageNumber, int* frameNumber) {
    for (int i = 0; i < TLB_SIZE; i++) {
        if (tlb[i].valid && tlb[i].pageNumber == pageNumber) {
            *frameNumber = tlb[i].frameNumber;
            tlb[i].lastUsed = ++time_counter; // Atualiza lastUsed para LRU
            return 1; // TLB Hit
        }
    }
    return 0; // TLB Miss
}
```

Importância: Demonstra como a TLB é consultada para um determinado número de página. Se encontrada (um "hit"), o número do frame correspondente é recuperado, e seu timestamp `lastUsed` é atualizado para a política LRU

Atualizando a TLB (Substituição LRU) (função `update_tlb`):

```c
void update_tlb(int pageNumber, int frameNumber) {
    int lruIndex = 0;
    unsigned long oldest = tlb[0].lastUsed;

    for (int i = 1; i < TLB_SIZE; i++) {
        if (!tlb[i].valid) {
            lruIndex = i;
            break;
        }
        if (tlb[i].lastUsed < oldest) {
            oldest = tlb[i].lastUsed;
            lruIndex = i;
        }
    }

    tlb[lruIndex].valid = 1;
    tlb[lruIndex].pageNumber = pageNumber;
    tlb[lruIndex].frameNumber = frameNumber;
    tlb[lruIndex].lastUsed = ++time_counter;
}
```

Importância: Implementa a estratégia de atualização da TLB. Primeiro, procura um slot vazio (inválido). Se todos os slots estiverem cheios, usa uma política LRU para selecionar uma entrada vítima para substituição. O `time_counter` ajuda a manter a ordem de uso.

4. Gerenciamento da Tabela de Páginas (`page_table.c`)

Carregando uma Página do Backing Store (função `load_page_from_backing_store`):

```c
int load_page_from_backing_store(int pageNumber) {
    FILE* file = fopen("backing_store.txt", "rb");
    if (!file) {
        perror("Erro ao abrir backing_store.txt");
        return -1;
    }

    int frame = -1;
    for (int i = 0; i < FRAME_COUNT; i++) {
        if (!frame_usage[i]) {
            frame = i;
            break;
        }
    }

    if (frame == -1) {
        fclose(file);
        return -1;
    }

    fseek(file, pageNumber * FRAME_SIZE, SEEK_SET);
    fread(physical_memory[frame], sizeof(char), FRAME_SIZE, file);
    fclose(file);
    frame_usage[frame] = 1;

    return frame;
}
```

Importância: Esta função trata uma falta de página (page fault). Ela encontra um frame livre na memória física (representada por `physical_memory[][]` e rastreada por `frame_usage[]`), lê a página necessária do `backing_store.txt` para esse frame e atualiza o uso do frame. Se não houver frame livre, retorna um erro (um SO real empregaria então um algoritmo de substituição de frame).

Recuperando uma Entrada da Tabela de Páginas (função `get_page_entry`):

```c
// From page_table.c
PageTableEntry* get_page_entry(int pageNumber) {
    if (pageNumber < 0 || pageNumber >= PAGE_TABLE_SIZE) return NULL;
    return &pageTable[pageNumber]; // Retorna o endereço da entrada da t
}
```

Importância: Uma função direta para acessar uma entrada no array `pageTable` usando o número da página como índice.

5. Gerenciamento de Threads e Inicialização (`main.c`)

Loop de Criação de Threads (na função `main`):

```c
// Lê endereços do arquivo
while (fgets(line, sizeof(line), file)) {
    uint32_t address;
    // Converte endereço hexadecimal ou decimal
    if (line[0] == '0' && line[1] == 'x') {
        address = strtol(line, NULL, 16);
    } else {
        address = atoi(line);
    }

    // Verifica se o endereço é válido (16 bits)
    if (address >= (1 << 16)) {
        printf("Endereço inválido (> 16 bits): %u\n", address);
        continue;
    }

    // Cria thread para processar o endereço
    ThreadArg* arg = malloc(sizeof(ThreadArg));
    arg->address = address;
    arg->id = thread_count;

    pthread_create(&threads[thread_count], NULL, process_address, arg);
    thread_count++;

    // Se atingiu o máximo de threads, espera todas terminarem
    if (thread_count >= MAX_THREADS) {
        for (int i = 0; i < thread_count; i++) {
            pthread_join(threads[i], NULL);
        }
        thread_count = 0;
    }
}

// Espera as threads restantes terminarem
for (int i = 0; i < thread_count; i++) {
    pthread_join(threads[i], NULL);
}
```

Importância: Mostra como o programa lê endereços virtuais de um arquivo e cria múltiplas threads (até `MAX_THREADS`) para processar esses endereços concorrentemente. `pthread_join` garante que a thread principal espere as threads de trabalho completarem.

Resultados obtidos com as simulações

```
[Thread 1] Endereço virtual: 64243 → Página: 250 | Offset: 243
[Thread 1] ❌ Página fora do intervalo: 250 (máximo permitido: 31)
[Thread 2] Endereço virtual: 2315 → Página: 9 | Offset: 11
[Thread 2] ⚠️Page Fault: Página 9 não está na RAM.
[Thread 3] Endereço virtual: 64454 → Página: 251 | Offset: 198
[Thread 3] ❌ Página fora do intervalo: 251 (máximo permitido: 31)
[Thread 4] Endereço virtual: 55041 → Página: 215 | Offset: 1
[Thread 4] ❌ Página fora do intervalo: 215 (máximo permitido: 31)
[Thread 5] Endereço virtual: 18633 → Página: 72 | Offset: 201
[Thread 5] ❌ Página fora do intervalo: 72 (máximo permitido: 31)
[Thread 6] Endereço virtual: 14557 → Página: 56 | Offset: 221
[Thread 6] ❌ Página fora do intervalo: 56 (máximo permitido: 31)
[Thread 7] Endereço virtual: 61006 → Página: 238 | Offset: 78
[Thread 7] ❌ Página fora do intervalo: 238 (máximo permitido: 31)
[Thread 2] ✅ Página 9 carregada no frame 0.
[Thread 2] 📥 Valor lido da memória: 45
[Thread 0] Endereço virtual: 62615 → Página: 244 | Offset: 151
[Thread 0] ❌ Página fora do intervalo: 244 (máximo permitido: 31)
[Thread 1] Endereço virtual: 7591 → Página: 29 | Offset: 167
[Thread 1] ⚠️Page Fault: Página 29 não está na RAM.
[Thread 2] Endereço virtual: 64747 → Página: 252 | Offset: 235
[Thread 2] ❌ Página fora do intervalo: 252 (máximo permitido: 31)
[Thread 1] ✅ Página 29 carregada no frame 1.
[Thread 3] Endereço virtual: 6727 → Página: 26 | Offset: 71
[Thread 3] ⚠️Page Fault: Página 26 não está na RAM.
[Thread 1] 📥 Valor lido da memória: 200
[Thread 3] ✅ Página 26 carregada no frame 2.
[Thread 4] Endereço virtual: 32315 → Página: 126 | Offset: 59
[Thread 4] ❌ Página fora do intervalo: 126 (máximo permitido: 31)
[Thread 3] 📥 Valor lido da memória: 192
[Thread 5] Endereço virtual: 60645 → Página: 236 | Offset: 229
[Thread 5] ❌ Página fora do intervalo: 236 (máximo permitido: 31)
[Thread 6] Endereço virtual: 6308 → Página: 24 | Offset: 164
[Thread 6] ⚠️Page Fault: Página 24 não está na RAM.
[Thread 6] ✅ Página 24 carregada no frame 3.
[Thread 7] Endereço virtual: 45688 → Página: 178 | Offset: 120
[Thread 7] ❌ Página fora do intervalo: 178 (máximo permitido: 31)
[Thread 6] 📥 Valor lido da memória: 207
[Thread 0] Endereço virtual: 969 → Página: 3 | Offset: 201
[Thread 0] ⚠️Page Fault: Página 3 não está na RAM.
[Thread 0] ✅ Página 3 carregada no frame 4.
[Thread 2] Endereço virtual: 49294 → Página: 192 | Offset: 142
[Thread 2] ❌ Página fora do intervalo: 192 (máximo permitido: 31)
[Thread 0] 📥 Valor lido da memória: 55
[Thread 1] Endereço virtual: 40891 → Página: 159 | Offset: 187
[Thread 1] ❌ Página fora do intervalo: 159 (máximo permitido: 31)
[Thread 3] Endereço virtual: 41118 → Página: 160 | Offset: 158
[Thread 3] ❌ Página fora do intervalo: 160 (máximo permitido: 31)
[Thread 4] Endereço virtual: 21395 → Página: 83 | Offset: 147
[Thread 4] ❌ Página fora do intervalo: 83 (máximo permitido: 31)
[Thread 5] Endereço virtual: 6091 → Página: 23 | Offset: 203
[Thread 5] ⚠️Page Fault: Página 23 não está na RAM.
[Thread 5] ✅ Página 23 carregada no frame 5.
[Thread 6] Endereço virtual: 32541 → Página: 127 | Offset: 29
[Thread 6] ❌ Página fora do intervalo: 127 (máximo permitido: 31)
[Thread 5] 📥 Valor lido da memória: 176
[Thread 7] Endereço virtual: 17665 → Página: 69 | Offset: 1
[Thread 7] ❌ Página fora do intervalo: 69 (máximo permitido: 31)
[Thread 0] Endereço virtual: 3784 → Página: 14 | Offset: 200
[Thread 0] ⚠️Page Fault: Página 14 não está na RAM.
[Thread 0] ✅ Página 14 carregada no frame 6.
```

Análise dos Resultados

Durante a execução da simulação de memória virtual, observou-se o funcionamento correto dos principais componentes implementados: TLB com política de substituição LRU, tabela de páginas e memória física simulada. A utilização de múltiplas threads permitiu a divisão eficiente do processamento dos endereços virtuais, com cada thread operando sobre um subconjunto dos dados, aproveitando o paralelismo da CPU.

As estatísticas coletadas ao final da execução, como número de acertos na TLB (TLB hits) e número de faltas de página (page faults), foram fundamentais para avaliar a eficiência da simulação. Em execuções típicas, observou-se que um número significativo de acessos foi resolvido diretamente pela TLB, reduzindo o tempo de acesso à memória. Isso demonstra a importância da TLB no desempenho de sistemas com gerenciamento de memória baseado em paginação.

Além disso, o uso da política LRU na TLB garantiu uma substituição inteligente das entradas, priorizando as páginas mais recentemente acessadas, o que também contribuiu para o aumento da taxa de acertos.

Mesmo com o uso de múltiplas threads, o desempenho pode variar dependendo da distribuição dos dados e da carga de trabalho de cada thread. Ainda assim, a abordagem paralela foi adequada para simular um ambiente mais realista de execução de processos concorrentes acessando a memória.

Conclusão

Este trabalho permitiu simular de forma prática e didática os conceitos fundamentais do gerenciamento de memória em sistemas operacionais, como a conversão de endereços virtuais para físicos, o uso da TLB com substituição LRU, e o tratamento de page faults com a tabela de páginas.

A implementação em linguagem C, com uso da biblioteca `pthread`, mostrou-se eficaz na criação de um sistema paralelo que simula acessos concorrentes à memória. O projeto abordou com sucesso todos os requisitos propostos, demonstrando não apenas o funcionamento da memória virtual, mas também a relevância de técnicas de otimização como o cache de endereços recentes (TLB).

Por fim, a análise dos dados coletados reforça a importância da TLB e do bom gerenciamento da tabela de páginas na melhoria do desempenho em ambientes com paginação. O simulador desenvolvido representa uma sólida base para aprofundamento em temas como substituição de páginas, escalonamento de memória e gerenciamento de múltiplos processos.