UNIVERSIDADE DO VALE DO ITAJAÍ
CURSO DE CIÊNCIA DA COMPUTAÇÃO

MEMÓRIA PRINCIPAL E MEMÓRIA VIRTUAL:
GERENCIADOR DE MEMÓRIA VIRTUAL

BERNARDO ESCOBAR
LAIS BLUM
LIVIA T. ZIMERMANN

ITAJAÍ
2025

SUMÁRIO

PROBLEMÁTICA

Suponha que um sistema tenha um endereço virtual de tamanho entre 16 bits à 32 bits com deslocamento na página de 256 B, 1 KB à 4 KB.

Escreva um programa que receba um endereço virtual (em decimal) na linha de comando ou leitura do arquivo addresses.txt faça com que ele produza o número da página e o deslocamento do endereço fornecido, sendo que essa posição indica qual a posição que será lido do arquivo data_memory.txt.

RESUMO

Este relatório descreve a implementação de um sistema de memória virtual que simula o processo de tradução de endereços virtuais para endereços físicos. O sistema implementa os principais componentes de um gerenciador de memória virtual, incluindo TLB (Translation Lookaside Buffer), tabela de páginas e gerenciamento de páginas.

O funcionamento se dá através de requisições de tradução de endereços, e utilizando uma estrutura hierárquica de memória, processa operações de tradução, carregamento de páginas e gerenciamento de cache.

A implementação demonstra o funcionamento eficiente do sistema de memória virtual, com suporte a tradução de endereços virtuais (32 bits) para físicos (24 bits), caching em dois níveis (TLB com 16 entradas e tabela de páginas com 32 entradas), gerenciamento de page faults e carregamento sob demanda, política de substituição LRU para o TLB, e backing store para armazenamento de páginas não residentes.

Palavras-chave: Memória virtual, tradução de endereços, endereços virtuais, endereços físicos, TLB (Translation Lookaside Buffer), tabela de páginas, gerenciamento de páginas, hierarquia de memória, cache, page fault, carregamento sob demanda, política de substituição LRU, backing store, simulação, sistema de memória, eficiência.

1.

2. INTRODUÇÃO

Sistemas de memória virtual são fundamentais para a gestão eficiente de recursos em sistemas operacionais modernos. Eles permitem que programas utilizem mais memória do que a fisicamente disponível, por meio da abstração de endereços virtuais. Essa técnica é amplamente adotada para isolar processos, melhorar a segurança e facilitar a alocação dinâmica de memória.

Neste contexto, este projeto tem como objetivo desenvolver um simulador que reproduz, de forma simplificada, o funcionamento de um sistema de memória virtual. O simulador é capaz de traduzir endereços virtuais (de 16 a 32 bits) em endereços físicos, considerando diferentes tamanhos de página (de 256 bytes a 4 KB). Além disso, a aplicação implementa mecanismos essenciais do gerenciamento de memória, como TLB (Translation Lookaside Buffer), tabela de páginas, detecção de faltas de página (page faults), uso de bits de controle (valid, accessed e dirty) e simulação de backing store para carregamento sob demanda.

3. ARQUITETURA

3.1. LÓGICA

O simulador de memória virtual implementa o processo de tradução de endereços virtuais para físicos de forma didática e funcional, simulando componentes reais de um sistema operacional. O fluxo de execução inicia-se com a conversão do endereço virtual, que pode ser informado diretamente em formato decimal ou hexadecimal, em sua representação binária através de bitset.

A tradução considera o tamanho total do endereço (16 ou 32 bits) e o tamanho da página (256 B, 1 KB, 2 KB ou 4 KB), influenciando diretamente a segmentação do endereço entre número da(s) página(s) e deslocamento. Para endereços de 32 bits com páginas de 4 KB, adota-se uma paginação em dois níveis, com separação dos bits em pag1 e pag2.

Após a segmentação, o sistema verifica a TLB (Translation Lookaside Buffer), que armazena as traduções mais recentes. Se houver um acerto (TLB hit), a tradução é imediata. Caso contrário (TLB miss), consulta-se a tabela de páginas. Em caso de ausência da página (Page Fault), é realizada a busca no backing store, que simula o armazenamento secundário.

O carregamento de páginas na tabela segue a política de substituição LRU (Least Recently Used), que também é aplicada à TLB. Todas as estruturas são atualizadas conforme o fluxo, e o endereço físico final é utilizado para acessar a simulação da memória principal via leitura do arquivo data_memory.txt.

3.2. ENTRADAS E SAÍDAS

O simulador de memória virtual desenvolvido aceita como entrada endereços virtuais representados tanto em formato decimal quanto hexadecimal, oferecendo flexibilidade ao usuário.

Além dos endereços propriamente ditos, o sistema requer a definição de dois parâmetros adicionais: o tamanho total do endereço virtual (16 ou 32 bits) e o tamanho do deslocamento (com opções entre 256 bytes e 4 KB). Essas configurações são fundamentais para determinar como os bits do endereço virtual serão particionados entre número da página e deslocamento, além de afetar diretamente a estrutura da tabela de páginas e a profundidade da hierarquia (quando aplicável).

A saída do programa é exibida diretamente no terminal. Para cada endereço processado, o sistema informa detalhadamente o fluxo de tradução, incluindo o endereço virtual original, o(s) número(s) da(s) página(s), o deslocamento correspondente e as decisões tomadas durante a simulação. Essas decisões envolvem mensagens como "TLB hit" ou "TLB miss", indicando se a tradução foi encontrada na TLB, e "Page fault" ou "Page hit", referentes à validade da página na tabela. Quando uma página inválida é acessada, a saída também indica que os dados foram "carregados da backing store".

Por fim, o valor correspondente ao endereço físico traduzido é lido do arquivo data_memory.txt e exibido como parte da resposta. Essa informação simula o dado real acessado na memória principal, encerrando o ciclo de tradução e acesso. Apesar de não haver tratamento de erro para endereços inválidos ou fora do intervalo esperado, a estrutura de entrada e saída é clara e consistente, facilitando a análise dos resultados.

3.3. ESTRUTURAS

3.3.1. TLB

A Translation Lookaside Buffer (TLB) é implementada como um vetor estático de 16 entradas para cada configuração (16 ou 32 bits). Cada entrada é definida por meio da estrutura:

```
typedef struct {
```

```
    int pg;               // Número da página virtual
    int value;            // Quadro de página física
    int temporizador = 0; // Marca de uso para LRU
} TLB;
```

Durante a execução, a TLB é consultada com a função findTLB, que retorna o quadro associado caso haja acerto (hit). Se não houver, realiza-se consulta à tabela de páginas e, posteriormente, à backing store se necessário.

A política de substituição implementada é LRU (Least Recently Used), baseada na comparação do campo temporizador. A função LRU_tlb_* (com variações para 16 e 32 bits) é responsável por inserir novas traduções na TLB, substituindo a entrada menos recentemente utilizada:

```
void LRU_tlb_16(int pag, int frame, TLB tlb16[]){
    // for(int i = 0; i % 16 < 16; i++){ //mod para fazer um for
circular, até achar access = 0
    //     if(tlb16[i].pg == -1 /*&& !tp16[i].dirty*/){ // 0, 0
    //         tlb16[i].value = frame;
    //         tlb16[i].temporizador = tlb16[i].temporizador++;
    //         return;
    //     }
    // }
    int posTempoMenor = 0; //menos utilizado
    for (int i = 1; i < 16; i++) {
        if (tlb16[i].temporizador <
tlb16[posTempoMenor].temporizador) {
            posTempoMenor = i;
        }
    }
    tlb16[posTempoMenor].pg = pag;
    tlb16[posTempoMenor].value = frame;
    tlb16[posTempoMenor].temporizador = tempo16++;
}
```

Esse modelo reflete o funcionamento real de TLBs em arquiteturas modernas, mantendo as traduções mais recentes ativas em cache.

3.3.2. TABELA DE PÁGINAS

A tabela de páginas é representada por um vetor estático de 32 entradas, implementado por estruturas em C++. Cada entrada armazena o número da página virtual, o valor do quadro de memória física associado, um bit de validade (indica se a entrada é válida), um bit de acesso (indica se a página foi acessada) e um bit de modificação.

A estrutura de TP usada é:

```
typedef struct {
     int value; // valor associado
     bool valid;
     bool access;
     bool dirty;
 } TP;
```

A TP é consultada por findTP16 ou findTP32, e caso não haja acerto (page fault), a função findBackingStore tenta localizar a página no arquivo backing_store.txt. Em caso de sucesso, a página é carregada na tabela de forma dinâmica por meio da substituição LRU (LRU_tp_*).

No caso de paginação em dois níveis, a estrutura usada é:

```
typedef struct {
    int value; // valor associado
    bool valid;
    bool access;
    bool dirty;
} NIVEL_2;

typedef struct {
    NIVEL_2* nivel2; // valor associado
    bool valid;
} NIVEL_1;
```

A alocação do segundo nível é feita sob demanda, e o controle também segue LRU para substituições. Essa arquitetura permite simular eficientemente cenários com grandes espaços de endereçamento.

3.3.3. BACKING STORE E SWAP

O simulador implementa o conceito de swap por meio da simulação de um backing store, representado pelo arquivo backing_store.txt. Esse arquivo atua como um armazenamento secundário, onde páginas que não estão presentes na memória principal (indicadas por valid = false na tabela de páginas) podem ser consultadas e carregadas sob demanda.

Durante a execução, quando ocorre uma falha na tabela de páginas (page fault), o sistema executa uma busca pelo número da página no backing store utilizando a função findBackingStore():

```
int findBackingStore(int searchPag) {
    ifstream arquivo("backing_store.txt");
    string linha;

    while (getline(arquivo, linha)) {
        stringstream ss(linha);
        string paginaStr, frameStr;

        if (getline(ss, paginaStr, ';') and getline(ss, frameStr))
{
            int pag = stoi(paginaStr);
            int frame = stoi(frameStr);

            if (pag == searchPag){
                return frame;
            }
        }
    }

    return -1;
}
```

Caso a página seja encontrada, ela é carregada para a memória principal e inserida na tabela de páginas utilizando a política de

substituição LRU (Least Recently Used), tanto para páginas quanto para entradas da TLB. O mesmo procedimento é seguido para a paginação em dois níveis, respeitando a hierarquia de pag1 e pag2.

Esse mecanismo reforça a simulação de um sistema de memória virtual com carregamento sob demanda, mantendo a coerência com sistemas reais que implementam swap entre RAM e disco para otimizar o uso de recursos limitados de memória física.

Exemplo de comportamento no código principal:

```
 if(!TPhit){
                    frame = findBackingStore(pag);
                    if(frame != -1){
                        pos = LRU_tp_16(frame, tp16);

                        LRU_tlb_16(pag, tp16[pos].value, tlb16);

                        frame = findTLB16(tlb16, pag, TLBhit);

                        sum = frame * tam + desloc;
                    }
                    else{
                        cout << "Erro ao tentar achar no backing
store" << endl;
                    }
                }
```

A implementação do swap torna o simulador mais robusto, fornecendo uma visão mais realista e funcional de como os sistemas operacionais modernos lidam com a falta de páginas e a gestão dinâmica da memória.

3.3.4. FORMATO DO ENDEREÇO VIRTUAL

O sistema suporta endereços de 16 e 32 bits, com diferentes divisões entre bits de página e deslocamento conforme o

tamanho da página especificado. A separação é feita por operações bitwise sobre bitset, como no exemplo:

```
pag = binaryInt32(num32, 10, 31);
desloc = binaryInt32(num32, 0, 9);
```

Endereços de 16 bits:

256B: 8 bits página, 8 bits deslocamento

1KB: 6 bits página, 10 bits deslocamento

2KB: 5 bits página, 11 bits deslocamento

Endereços de 32 bits:

256B: 24 bits página, 8 bits deslocamento

1KB: 22 bits página, 10 bits deslocamento

4KB: Paginação em dois níveis:

pag1: 10 bits (bits 22-31)

pag2: 10 bits (bits 12-21)

deslocamento: 12 bits (bits 0-11)

A paginação em dois níveis é gerenciada por estruturas NIVEL_1 e NIVEL_2, conforme o seguinte trecho de código:

```
int findTP32_2niveis(NIVEL_1 tp_nivel1[], int pag1, int pag2, bool &TPhit){
    // Primeiro: verifica se o nível 1 está válido
    if (tp_nivel1[pag1].valid) {
        NIVEL_2* tp_nivel2 = tp_nivel1[pag1].nivel2;

        // Se o ponteiro não foi alocado por algum motivo, aloca agora
        if (tp_nivel2 == nullptr) {
            tp_nivel2 = new NIVEL_2[32];
            for (int i = 0; i < 32; ++i) {
                tp_nivel2[i].valid = false;
                tp_nivel2[i].access = false;
                tp_nivel2[i].dirty = false;
```

```
            }
            tp_nivel1[pag1].nivel2 = tp_nivel2;
        }

        // Segundo: verifica se o nível 2 está válido
        if (tp_nivel2[pag2].valid) {
            TPhit = true;
            return tp_nivel2[pag2].value;
        }
    }
    else {
        // Se o nível 1 ainda não está válido, aloca o nível 2 e
marca como válido
        tp_nivel1[pag1].nivel2 = new NIVEL_2[32];
        for (int i = 0; i < 32; ++i) {
            tp_nivel1[pag1].nivel2[i].valid = false;
        }
        tp_nivel1[pag1].valid = true;
    }

    // Caso não haja acerto (miss)
    TPhit = false;
    return -1;
}
```

Esse modelo reproduz o comportamento de arquiteturas reais com segmentação de grandes espaços de endereçamento, mantendo o caráter didático da simulação.

4. CÓDIGOS

Esta seção apresenta trechos representativos da implementação do simulador, com ênfase na tradução de endereços virtuais, políticas de substituição e tratamento de faltas de página. Os códigos foram implementados em C++, com uso de bitset para manipulação binária e estruturas estáticas para simular a TLB e a tabela de páginas.

Conversão de Endereços e Separação de Bits

Esta seção apresenta trechos representativos da implementação do simulador, com ênfase na tradução de endereços virtuais, políticas de substituição e tratamento de

faltas de página. Os códigos foram implementados em C++, com uso de bitset para manipulação binária e estruturas estáticas para simular a TLB e a tabela de páginas.

```
//Seleciona parte do bitset (start, end) e converte de decimal para
binário
int binaryInt16(const bitset<16>& bits, int start, int end) {
    int result = 0; // Armazena o valor final em inteiro
    int bitPosition = 0; // Posição atual do bit no resultado final

    for (int i = start; i <= end; ++i) {
        if (bits[i]) { // Verifica se o bit na posição i é 1
            result |= (1 << bitPosition); // Define o bit correspondente
em result
        }

        ++bitPosition; // Avança para a próxima posição do bit no
resultado
    }

    return result;
}
```

Cálculo do Endereço Físico: Multiplicação pelo Tamanho da Página

O passo final da tradução de um endereço virtual consiste em calcular o **endereço físico**, combinando o número do quadro (frame) e o deslocamento. Isso é feito com a seguinte fórmula:

```
sum = frame * tam + desloc;
```

frame: é o índice do quadro de página física onde a página virtual está armazenada. Ele é obtido após consulta à TLB, tabela de páginas ou backing store.

tam: é o tamanho da página, definido pelo usuário na entrada (ex: 256 B, 1 KB, 4 KB).

desloc: representa o offset dentro da página, extraído dos bits menos significativos do endereço virtual.

A multiplicação frame * tam obtém o endereço base do quadro de página física. A adição do deslocamento completa o cálculo, apontando para a posição exata dentro da memória física onde o dado está.

Consulta à TLB e Política de Substituição

A busca por traduções recentes ocorre na TLB, que armazena até 16 entradas. A função findTLB16 verifica se a página está presente:

```
int findTLB16(TLB tlb16[], int pag, bool &TLBhit) {
    for(int i = 0; i < 16; i++) {
        if (tlb16[i].pg == pag) {
            TLBhit = true;
            tlb16[i].temporizador++;
            return tlb16[i].value;
        }
    }
    TLBhit = false;
    return -1;
}
```

Se não houver acerto, uma nova entrada é inserida pela função LRU_tlb_16, que seleciona a entrada menos utilizada com base no campo temporizador:

```
void LRU_tlb_16(int pag, int frame, TLB tlb16[]) {
    int posTempoMenor = 0;
    for (int i = 1; i < 16; i++) {
        if (tlb16[i].temporizador < tlb16[posTempoMenor].temporizador) {
            posTempoMenor = i;
        }
    }
    tlb16[posTempoMenor] = {pag, frame, tempo16++};
}
```

Tabela de Páginas e LRU

A tabela de páginas é consultada em seguida. Se a página estiver válida (valid == true), o quadro é retornado. Caso contrário, a falta é tratada com acesso ao backing store:

```
int findTP16(TP tp16[], int pag, bool &TPhit) {
    if (tp16[pag].valid) {
        TPhit = true;
        return tp16[pag].value;
    }
```

```
    TPhit = false;
    return -1;
}
```

A substituição segue a política LRU com reset de bits de acesso:

```
int LRU_tp_16(int frame, TP tp16[]) {
    for (int i = 0; i < 32; i++) {
        if (!tp16[i].access) {
            tp16[i].value = frame;
            tp16[i].valid = true;
            tp16[i].access = true;
            return i;
        } else {
            tp16[i].access = false;
        }
    }
    return -1;
}
```

Paginação em Dois Níveis

Para endereços de 32 bits com páginas de 4 KB, a estrutura utiliza dois níveis de paginação. O primeiro nível (NIVEL_1) contém ponteiros para tabelas do segundo nível (NIVEL_2). A consulta é feita com verificação da validade de ambos os níveis.

A substituição também segue LRU:

```
void LRU_tp_2niveis(int frame, NIVEL_1 tp_nivel1[], int &ret_pag1,
int &ret_pag2) {
    for (int p1 = 0; p1 < 32; ++p1) {

        if (!tp_nivel1[p1].valid) {
            // Alocar o nível 2 se não estiver alocado
            tp_nivel1[p1].nivel2 = new NIVEL_2[32];

            for (int i = 0; i < 32; ++i) {
                tp_nivel1[p1].nivel2[i].valid = false;
                tp_nivel1[p1].nivel2[i].access = false;
```

```
            }

            tp_nivel1[p1].valid = true;
        }

        NIVEL_2* nivel2 = tp_nivel1[p1].nivel2;

        for (int p2 = 0; p2 < 32; ++p2) {

            if (!nivel2[p2].access /* && !nivel2[pag2].dirty */) {
                // Substituição LRU aqui
                nivel2[p2].value = frame;
                nivel2[p2].valid = true;
                nivel2[p2].access = true;

                ret_pag1 = p1;
                ret_pag2 = p2;

                return;
            }
            else if (nivel2[p2].access) {
                nivel2[p2].access = false;
            }
        }
    }
}
```

5. SIMULAÇÕES

CASO 1:

16 BITS

ENTRADA DECIMAL

DESLOCAMENTO DE 1KB

```
Qual tamanho de endereco virtual? 1 - 16 bits; 2 - 32 bits
1
Vai informar o endereço em que formato? 1 - Decimal; 2 - Hexadecimal
1
Qual é o endereço virtual? (Entre 0 e 65535)
59757
Qual tamanho do deslocamento? 1 - 256 B; 2 - 1 KB; 3 - 2 KB
2
```

RESULTADO

```
Endereco virtual: 1110100101101101
TLBhit: true
Carregado da TLB
Pagina: 58
Deslocamento: 365
Soma: 101741
Valor lido: 29
```

6. CONCLUSÃO

O desenvolvimento deste simulador de memória virtual proporcionou uma compreensão aprofundada dos mecanismos utilizados por sistemas operacionais reais para gerenciar o espaço de endereçamento de processos. Através da implementação de estruturas como TLB, tabela de páginas, backing store e suporte a diferentes tamanhos de página, foi possível reproduzir de forma didática e funcional os principais conceitos envolvidos na tradução de endereços virtuais para físicos.

Dentre os aprimoramentos realizados, destaca-se o suporte à entrada de endereços no formato hexadecimal, alinhando-se às práticas convencionais em arquitetura de computadores. Além disso, a adoção de uma política de substituição baseada em LRU, tanto para a TLB quanto para a tabela de páginas, torna a simulação mais realista e próxima das estratégias adotadas em ambientes de produção.

A introdução de um sistema de swap com leitura dinâmica do backing store, bem como a implementação de paginação em dois níveis para endereços de 32 bits, ampliaram a complexidade do simulador, permitindo representar cenários com hierarquia de memória e escassez de quadros físicos.