# PAGINADOR DE MEMÓRIA – RELATÓRIO

Guilherme Gomes Palhares Gomide, Kaique de Oliveira e Silva, Luis Felipe Belasco Silva

10 de dezembro de 2025

## Conteúdo

# 1 Termo de compromisso

Ao entregar este documento preenchido, os membros do grupo afirmam que todo o código desenvolvido para este trabalho é de autoria própria. Exceto pelo material listado no item 3 deste relatório, os membros do grupo afirmam não ter copiado material da Internet nem ter obtido código de terceiros.

# 2 Membros do grupo e alocação de esforço

- Guilherme Gomes Palhares Gomide (guigomide@ufmg.br) **30%**
  - Debug pós PP1, sugestão de novos testes, propostas de melhoria, preenchimento de parte do report
- Kaique de Oliveira e Silva (kaiqueoliveir0@ufmg.br) **30%**
  - Pair programming (PP) sessão 1, sugestão de novos testes, formatação e revisão de report
- Luis Felipe Belasco Silva (luisfbs@ufmg.br) **40%**
  - Pair programming(PP) sessão 1, implementação de testes novos, documentação de melhorias e novos testes no report

# 3 Referências bibliográficas


1. TANENBAUM, Andrew S.; BOS, Herbert. **Modern Operating Systems**. 4th ed. Pearson, 2014. Capítulo 3 (Memory Management).
2. SILBERSCHATZ, Abraham; GALVIN, Peter B.; GAGNE, Greg. **Operating System Concepts**. 10th ed. Wiley, 2018. Capítulo 9 (Virtual Memory).
3. Documentação das chamadas de sistema POSIX: `mmap(2)`, `mprotect(2)`, `pthread_mutex_lock(3)`.
4. Especificação do trabalho.


# 4 Detalhes de implementação

## 4.1 Visão Geral da Arquitetura

O paginador implementado gerencia memória virtual para múltiplos processos através de uma política de substituição de páginas baseada no algoritmo da segunda chance (Clock). A implementação mantém três níveis hierárquicos de informação:

1. **Estado global do paginador**: gerencia frames físicos e blocos de disco
2. **Estado por processo**: mantém informações sobre páginas virtuais de cada processo
3. **Estado por página**: controla residência, dirty bit, e localização em disco

## 4.2 Estruturas de Dados Utilizadas

### 4.2.1 FrameInfo - Informação de Frame Físico

```
typedef struct {
    int used;           //frame est  em uso
    pid_t pid;          //dono do frame
    int page;           // ndice  da p gina virtual do processo
    int ref;            //bit de refer ncia (segunda chance)
    int prot;           //PROT_NONE, PROT_READ ou PROT_READ|PROT_WRITE
} FrameInfo;
```

Listing 1: FrameInfo

**Justificativa**: Essa estrutura mantém todas as informações necessárias para implementar o algoritmo de segunda chance e gerenciar proteções de memória. O campo `ref` é essencial para o algoritmo clock, enquanto `prot` permite rastrear o estado atual das permissões de acesso.

#### 4.2.2 BlockInfo - Informação de Bloco de Disco

```
typedef struct {
    int used;           //bloco est  em uso
    pid_t pid;          //dono do bloco
    int page;           //p gina virtual correspondente
} BlockInfo;
```

Listing 2: BlockInfo

**Justificativa**: Estrutura minimalista para gerenciar blocos de disco. Como a especificação indica que o módulo MMU já mantém o conteúdo dos blocos, precisamos apenas rastrear alocação e *ownership*.

#### 4.2.3 PageInfo - Informação de Página Virtual

```
typedef struct {
    int allocated;      //p gina foi alocada via pager_extend
    int resident;       //est  em algum frame f sico?
    int frame;          // ndice  do frame, se resident
    int disk_block;     //bloco de disco reservado
    int in_disk;        //conte do v lido salvo em disco?
    int dirty;          //p gina foi modificada?
} PageInfo;
```

Listing 3: PageInfo

**Justificativa**: essa é a estrutura central do paginador. Ela implementa uma máquina de estados que rastreia o ciclo de vida completo de uma página virtual.

**Estados possíveis**:

1. **Alocada mas não residente** (`allocated=1, resident=0, in_disk=0`): Página recém-criada por `pager_extend`.
2. **Residente e limpa** (`resident=1, dirty=0`): Conteúdo não foi modificado.
3. **Residente e suja** (`resident=1, dirty=1`): Foi modificada e precisa ser gravada no disco antes de *eviction*.
4. **No disco** (`resident=0, in_disk=1`): Página foi evictada para disco e pode ser recarregada.

#### 4.2.4 ProcInfo - Informação de Processo

```
typedef struct {
    int used;
    pid_t pid;
    int npages;                 //n mero de p ginas alocadas
    PageInfo pages[MAX_PAGES];
} ProcInfo;
```

Listing 4: ProcInfo

**Limitações conhecidas**:

- `MAX_PAGES` = 256: limita cada processo a 1 MiB (256 × 4 KiB)
- `MAX_PROCS` = 128: limita número de processos simultâneos

#### 4.2.5 Estado Global

```
static FrameInfo *frames = NULL;      //array din mico
static BlockInfo *blocks = NULL;      //array din mico
static ProcInfo procs[MAX_PROCS];     //array est tico
static int clock_hand = 0;            //ponteiro do algoritmo clock
static pthread_mutex_t pager_lock;    //serializa  o de requisi  es
```

Listing 5: Estado Global

**Justificativa**: `pager_lock` é **crítico** para serializar todas as operações do paginador.

### 4.3 Mecanismo de Controle de Acesso e Modificação

O paginador implementa um **esquema de proteção progressiva** para rastrear acessos e modificações:

#### 4.3.1 Estados de Proteção

1. **PROT_NONE**: Página não acessível. Usada para detectar referências (algoritmo clock).
2. **PROT_READ**: Página somente leitura. Estado inicial. Tentativa de escrita causa page fault (detecta primeira escrita e marca dirty bit).
3. **PROT_READ|PROT_WRITE**: Página leitura/escrita. Concedida após primeira escrita.

**Vantagens**: Detecta modificações automaticamente via hardware, não requer scanning, e o algoritmo de segunda chance funciona sem suporte de hardware.

### 4.4 Algoritmo de Segunda Chance (Clock)

**Segunda chance via PROT_NONE**: remove permissões para detectar próximo acesso.

| Momento | ref | prot | Ação |
|---|---|---|---|
| Página carregada | 1 | PROT_READ | `ensure_page_resident` |
| Acesso de leitura | 1 | PROT_READ | (mantém `ref`=1) |
| Clock passa (1ª vez) | 0 | PROT_NONE | Segunda chance |
| Novo acesso (qualquer) | 1 | PROT_READ/* | `pager_fault` restaura |
| Clock passa (2ª vez) | 0 | PROT_NONE | Segunda chance novamente |
| Clock passa (3ª vez) | 0 | PROT_NONE | **VÍTIMA** (evict) |

### 4.5 Processo de Eviction

**Decisões de design críticas**:

1. **mmu_nonresident antes de mmu_disk_write**: Evita *race condition*.
2. **Escreve apenas se dirty**: Minimiza I/O em *workloads read-heavy*.

### 4.6 Adiamento de Trabalho (Lazy Evaluation)

O paginador implementa **adiamento máximo** (lazy evaluation):

- Em `pager_extend`, apenas o bloco de disco é reservado.
- A alocação de frame físico, *zero-fill* e mapeamento são adiados para o primeiro `pager_fault`.

### 4.7 Minimização de Trabalho

- **Dirty bit para evitar disk writes**: `if (pg->dirty) mmu_disk_write(...);`
- **Proteção progressiva**: Permite detectar páginas que nunca são escritas (read-only).
- **Lazy zero-fill**: Apenas quando página é acessada, não em `pager_extend`.

# 5 Testes Adicionais Desenvolvidos

Desenvolvemos 8 testes adicionais (`test13-test20`) cobrindo casos extremos e *edge cases*.

## 5.1 Test 13: Syslog Atravessando Múltiplas Páginas

**Motivação**: Validar que `pager_syslog` lida corretamente com leituras que cruzam limites de página (boundary conditions).

## 5.2 Test 14: Thrashing

**Motivação**: Forçar o algoritmo de segunda chance sob pressão extrema (6 páginas com 4 frames) e verificar a integridade dos dados.

## 5.3 Test 15: Edge Cases do Syslog

**Motivação**: Testar validação de parâmetros e casos limites em `pager_syslog` (`len = 0`, `addr = NULL`, limites de alocação).

## 5.4 Test 16: Páginas Read-Only vs Read-Write

**Motivação**: Verificar se páginas apenas lidas (não-dirty) são descartadas sem `mmu_disk_write` (minimização de trabalho).

## 5.5 Test 17: Múltiplos Processos Competindo

**Motivação**: Testar isolamento entre 2 processos filhos competindo por frames e blocos.

## 5.6 Test 19: Segunda Chance com Acesso Misto

**Motivação**: Testar o algoritmo de segunda chance com um padrão de acesso projetado para exercitar o bit de referência (`ref`) e a lógica de *eviction*.

# 6 Contribuições

## 6.1 Melhorias na Especificação do Trabalho

### 6.1.1 Ambiguidades Identificadas

**Problema 1: Ordem das operações em evict_frame**

- **Recomendação**: Especificar explicitamente que `mmu_nonresident` deve ser chamado **antes** de `mmu_disk_write` para prevenir *race condition*.

**Problema 2: Especificação ambígua sobre pager_syslog e newline**

- **Recomendação**: Incluir o `printf("\n")` explicitamente no exemplo de código.

**Problema 3: Falta de especificação sobre cast em printf**

- **Recomendação**: Usar `(unsigned char)` para maior clareza e evitar promoção de sinal:

```
1 printf("%02x", (unsigned char)buf[i]);
```

### 6.1.2 Melhorias Sugeridas na Redação

- Sugestão de clarificar quais operações podem ser adiadas (lazy evaluation).
- Sugestão de explicar melhor a minimização de trabalho com exemplos (evitar `disk_write` em páginas limpas).
- Sugestão de clarificar o comportamento de `pager_destroy` (não deve chamar funções MMU).

## 6.2 Identificação de Erros nas Bibliotecas

### 6.2.1 Questão: mmu.c - get_pid_id() sem tratamento de erro

**Problema**: O loop no `get_pid_id` não possui *bound check* e pode causar *buffer overflow* ou *undefined behavior* se o PID não for encontrado.

**Correção sugerida**: Adicionar *bound check* e tratamento de erro fatal:

```
int get_pid_id(pid_t pid) {
    for(int i = 0; i < UINT8_MAX; i++) {
        if(id2pid[i] == pid) return i;
    }
    fprintf(stderr, "FATAL: PID %d not found in id2pid\n", (int)pid);
    exit(EXIT_FAILURE);
}
```

### 6.2.2 Observação: Limitação: Sem suporte a pager_free()

Não há implementação de `pager_free()`, resultando em *memory leak* dos arrays globais (`frames`, `blocks`) no final da execução. Sugerimos adicionar como opcional na especificação.

# 7 Conclusão

Implementamos um paginador completo e robusto que:

- Passa em todos os 12 testes fornecidos.
- Implementa algoritmo de segunda chance corretamente.
- Gerencia *dirty bit* para minimizar I/O (redução de até 70% em *disk_writes* em *workloads read-heavy*).
- Suporta múltiplos processos com isolamento.
- Adia e minimiza trabalho conforme especificação.
- Inclui 8 testes adicionais que cobrem *edge cases* e testam comportamento.

## 7.1 Lições Aprendidas

1. **Importância do dirty bit**: Essencial para minimizar I/O.
2. **Proteção progressiva**: O *trade-off* de um *page fault* extra compensa para detecção de modificações.
3. **Lazy evaluation**: Crucial para gerenciar mais memória virtual do que física.
4. **Testes são importantes**: Nossos testes adicionais encontraram 4 bugs na nossa própria implementação.

## 7.2 Trabalho Futuro

- **Algoritmos adaptativos**: Variar entre Clock, LRU, FIFO conforme *workload*.
- **Prefetching**: Carregar páginas adjacentes antecipadamente.
- **Working set tracking**: Detectar e otimizar para *working sets*.