# Simulador de Memória Virtual

Camila Santana Braz - 2019027423

João Pedro Menezes - 2019027679

Junho de 2023

## 1 Introdução

A memória virtual desempenha um papel crucial no funcionamento eficiente dos sistemas computacionais atuais. Ela permite que os programas acessem uma quantidade maior de memória do que está fisicamente disponível no sistema, utilizando uma combinação de memória principal (RAM) e memória secundária (disco rígido ou SSD). Para gerenciar esse processo complexo, são utilizados simuladores de memória virtual, que fornecem uma visão abstrata da memória ao sistema operacional e aos programas em execução.

Em resumo, um simulador de memória virtual é uma ferramenta que reproduz o funcionamento do gerenciamento de memória virtual em um sistema real, permitindo que os desenvolvedores e pesquisadores estudem e testem diferentes cenários e algoritmos de gerenciamento de memória.

O funcionamento básico de um simulador de memória virtual envolve os seguintes passos:

- Configuração: O simulador precisa ser configurado com parâmetros que definem o tamanho total da memória física disponível, o tamanho das páginas de memória e o algoritmo de substituição de páginas a ser utilizado.
- Divisão em páginas: A memória física e a memória virtual são divididas em páginas de tamanho fixo. As páginas são blocos de memória contíguos que servem como unidades básicas de alocação.
- Mapeamento: O simulador mantém tabelas de mapeamento que associam as páginas de memória virtual às páginas de memória física correspondentes.
- Substituição de páginas: Quando um programa tenta acessar uma página de memória virtual que não está na memória física, ocorre um evento de falta de página *(page fault)*. Dessa forma, o simulador simula o carregamento da página em disco na memória atualizando a tabela de páginas. Caso a memória esteja

super alocada ocorre a substituição de páginas, em que é selecionada uma página para ser removida da memória física e substituída pela nova página. O simulador precisa implementar um algoritmo de substituição de páginas para decidir qual página será removida da memória física quando ocorrer uma falta de página. Algoritmos comuns incluem o algoritmo de substituição de página FIFO, o LRU, o Segunda Chance e o Random.

- Estatísticas e monitoramento: O simulador pode coletar estatísticas sobre o desempenho da memória virtual, como o número de faltas de página, o número de páginas escritas e o tempo de execução. Essas estatísticas podem ser usadas para avaliar a eficácia dos algoritmos de substituição de páginas e fazer ajustes no sistema.

Neste trabalho, será abordado o desenvolvimento de um simulador de memória virtual que opere de acordo com os princípios fundamentais dessa tecnologia e que atenda aos requisitos específicos fornecidos pelo professor e disponíveis na tabela 1.

| Requisito | Valor |
|---|---|
| Algoritmos | FIFO, LRU, RANDOM, 2A |
| Tamanho da página | 2KB a 64KB |
| Tamanho da memória | 128KB a 16384KB |
| Desempenho | Ordem de segundos |

Tabela 1: Especificações do trabalho

# 2 Estrutura geral

De forma geral, o código implementa um simulador de gerenciamento de memória em linguagem C.

Ao ser executado, o programa recebe uma série de parâmetros de entrada através da linha de comando. Esses parâmetros incluem o algoritmo de substituição de páginas a ser utilizado (FIFO, LRU, RANDOM ou Segunda Chance), o arquivo de entrada que contém os acessos à memória (compilador.log, compressor.log, matriz.log, simulador.log), o tamanho da página e o tamanho total da memória. Existe também um quinto parâmetro de entrada opcional (quinto argumento da chamada deve ser igual a 'debug') que, caso incluído, executa o programa em modo debug, ou seja, imprimindo no terminal uma depuração detalhada do funcionamento do simulador.

Após a leitura dos parâmetros, o programa realiza uma verificação para garantir que os valores fornecidos são válidos e estão dentro dos limites esperados. Essas verificações são feitas em relação ao algoritmo de substituição de páginas, ao arquivo.log

de entrada e se os tamanhos de página e memória total estão dentro dos limite razoáveis e possíveis setados na tabela 1.

Com os parâmetros validados, o programa aloca memória para a tabela de páginas e para a memória física, de acordo com o tamanho total da memória e o tamanho da página especificados. Em seguida, inicializa essas estruturas de dados, garantindo que elas estejam prontas para uso. Isso inclui a configuração de valores iniciais como zero para todos os atributos de ambas estruturas, marcando como livres e zerados todos atributos dos quadros de memória e das páginas.

Após a inicialização, o programa abre o arquivo de entrada que contém os acessos à memória, lê cada linha e processa o acesso correspondente.. Cada linha do arquivo representa um acesso à memória e contém um endereço de memória e uma operação (leitura ou escrita).

Durante o processamento de um acesso à memória, o programa primeiro busca por quadros não ocupados na memória física. Caso encontre, a página é alocada a esse quadro. Caso contrário, se a memória estiver cheia o simulador utiliza o algoritmo de substituição de páginas definido na chamada da execução para decidir qual página deve ser substituída na memória física. Isso envolve a consulta da tabela de páginas para verificar se a página desejada já está presente na memória ou se precisa ser alocada.

Durante a execução do simulador, são coletadas estatísticas para avaliar o desempenho do sistema de memória, que incluem o número total de acessos à memória, o número de (page_faults) e o número de páginas escritas. Essas estatísticas são atualizadas conforme os acessos são processados e serão utilizadas para analisar o desempenho de cada técnica de substituição em conjunto com o tempo de execução na seção 5. O tempo de execução é calculado em um arquivo de extensão .sh utilizado para realizar os testes definidos pelo professor. Ao final da simulação, o programa imprime as estatísticas coletadas, fornecendo um relatório completo do desempenho do sistema com base nos parâmetros de entrada fornecidos.

Dessa forma, o código implementa um simulador que permite experimentar diferentes algoritmos de substituição de páginas e analisar seu impacto no desempenho do sistema de gerenciamento de memória.

# 3 Estruturas de Dados

## 3.1 Variáveis globais

```
1 // Variaveis globais
2 // proximo fifo é utilizado no algorirmo de substituicao segunda chance
3 int proximo_fifo = 0;
```

Listing 1: Varíaveis globais

O programa possui uma variável global 1 utilizada no algoritmo de substituição Segunda Chance para armazenar o próximo quadro da fila a partir da ordenação feita pelo algoritmo FIFO (First-in First-out). Ela é inicializada em zero e vai seguindo uma lista circular à medida que as substituições vão ocorrendo.

## 3.2 Estruturas

```
// Estrutura para representar um quadro de memoria na memória física
typedef struct {
    int indice;           // Número da página que ocupa o quadro
    int ultimo_acesso;    // Tempo do último acesso ao quadro
    int ocupado;          // Flag para indicar se o quadro está ocupado
} Quadro;
```

Listing 2: Estruturas de dados principais: Quadro.

Quadro é uma estrutura que é projetada para representar um quadro na memória física. Ele possui três atributos: "índice", que indica o número da página que ocupa o quadro, "ultimo_acesso", que indica quando o quadro foi acessado pela última vez e por fim "ocupado", que denota se o quadro está ocupado no momento ou não.

```
// Estrutura para representar uma entrada de página na tabela de páginas
typedef struct {
    int referencia;       // Flag para indicar se a página está na memória
    física
    int ultimo_acesso;    // Tempo do último acesso a página
    int suja;             // Flag que indica se a página está limpa ou
   suja
    int bit_ref; // flag q indica se a pagina ja foi rerenciada ou nao
} Pagina;
```

Listing 3: Estruturas de dados principais: Página.

Pagina é uma estrutura de dados que representa uma página da tabela de páginas, ela mapeia os endereços virtuais para os quadros de memória física correspondentes. Essa tabela é utilizada pelo algoritmo de substituição de páginas para tomar decisões sobre quais páginas devem ser mantidas na memória e quais devem ser substituídas, levando em conta qual página está disponível ou não. Para isso, ela possui quatro atributos: uma *flag* "referência", para indicar se a páginas está na memória física, "ultimo_acesso", que indica quando o quadro foi acessado pela última vez, "suja"que indica se a página está limpa (ocupada) ou não e um "bit ref"que é usado no algoritmo de substituição e indica se a página está apta a ter uma segunda chance ou não.

## 3.3 Funções de verificação e de imprimir estatísticas

```
void check_algoritmo_substituicao(char *algoritmo_check) {
```

```
2     // O programa utiliza 4 algoritmos de substituição: 2a, fifo, lru ou
      random
3     // Essa funcao confere se a entrada feita por linha de comando do nome
       do algoritmo é valida
4     if(strcmp(algoritmo_check, "2a") == 0 || strcmp(algoritmo_check, "fifo
      ") == 0 || strcmp(algoritmo_check, "random") == 0 || strcmp(
      algoritmo_check, "lru") == 0) {
5         return;
6     }
7     else {
8         printf("Algoritmo de substituição inválido!\n Favor executar
      novamente escolhendo '2A', 'FIFO', 'LRU' ou 'RANDOM' como algoritmo.\n
      ");
9         exit(1);
10    };
11 }
```

Listing 4: Função para verificar o algoritmo de substituição

A função 4 recebe o nome de um algoritmo de substituição como entrada e verifica se ele está dentro dos algoritmos disponíveis válidos. Os algoritmos disponíveis são: "2a", "fifo", "random"e "lru". Se o nome do algoritmo fornecido for igual a um dos algoritmos válidos, a função retorna. Caso contrário, a função imprime uma mensagem de erro indicando que o algoritmo de substituição é inválido e, em seguida, encerra a execução com um código de saída 1.

```
1  void check_arquivo_entrada(char *arquivo_check) {
2     // O programa deve ler um arquivo de entrada com extensão .log
3     // Os arquivos disponíveis são: compilador.log, compressor.log, matriz
      .log e simulador.log
4     // Essa função confere se essa entrada feita por linha de comando do
      nome do arquivo
5     // é valida
6     if(strcmp(arquivo_check, "compilador.log") == 0 || strcmp(
      arquivo_check, "compressor.log") == 0 ||
7         strcmp(arquivo_check, "matriz.log") == 0 || strcmp(arquivo_check,
      "simulador.log") == 0) {
8         return;
9     }
10    else {
11        printf("Arquivo de entrada de memória inválido!\n Favor executar
      novamente escolhendo 'compilador.log', 'compressor.log', 'matriz.log'
      ou 'simulador.log' como arquivo de entrada.\n");
12        exit(1);
13    };
14 }
```

Listing 5: Função para verificar o arquivo de entrada

A função 5 recebe o nome de um arquivo como entrada e verifica se ele está dentro

dos arquivos disponíveis válidos. Os arquivos disponíveis são: "compilador.log", "compressor.log", "matriz.log"e "simulador.log". Se o nome do arquivo fornecido for igual a um dos arquivos válidos, a função retorna. Caso contrário, a função imprime uma mensagem de erro indicando que o arquivo de entrada é inválido e, em seguida, encerra a execução com um código de saída 1.

```
void check_tamanho_quadro_memoria(int check_tamanho_quadro) {
    // O tamanho do quadro da memória deve ser um valor positivo maior que zero e múltiplo de 2
    // Além disso, um tamanho razoável para o quadro de memória está na faixa de 2KB a 64KB
    // Esta função valida se essa entrada está dentro destas duas faixas
    if(!(check_tamanho_quadro % 2) && check_tamanho_quadro >= 2 && check_tamanho_quadro <= 64){
        return;
    }
    else {
        printf("Tamanho de quadro de memória inválido!\n");
        if(check_tamanho_quadro < 2 || check_tamanho_quadro > 64){
            printf("Valor fora da faixa razoável! Favor entrar com um valor maior ou igual a 2KB e menor ou igual a 64KB.\n");
        }
        if(check_tamanho_quadro % 2){
            printf("Valor não múltiplo de 2. Favor entrar com um valor que seja divisível por 2.\n");
        }
        if(check_tamanho_quadro < 0){
            printf("Valor negativo. Favor entrar com um valor que seja maior que zero.\n");
        }
        exit(1);
    };
    }
```

Listing 6: Função para verificar o tamanho do quadro da memória

A função 6 recebe um tamanho de quadro de memória como entrada e verifica se ele está dentro dos critérios válidos. Os critérios são: o tamanho do quadro deve ser um valor positivo maior que zero, múltiplo de 2 e estar na faixa razoável de 2KB a 64KB. Se o tamanho não atender a esses critérios, a função imprime mensagens de erro específicas para indicar qual critério não foi cumprido e, em seguida, encerra a execução com um código de saída 1.

```
void check_tamanho_memoria_total(int check_tamanho_memoria) {
    // O tamanho da memória total deve ser um valor positivo maior que zero e múltiplo de 2
    // Além disso, um tamanho razoável paraa memória total está na faixa de 128KB a 16384KB
    // Esta função valida se essa entrada está dentro destas duas faixas
```

```
5     if(!(check_tamanho_memoria % 2) && check_tamanho_memoria >= 128 &&
      check_tamanho_memoria <= 16384){
6         return;
7     }
8     else {
9         printf("Tamanho de memória total inválido!\n");
10        if(check_tamanho_memoria < 128 || check_tamanho_memoria> 16384){
11            printf("Valor fora da faixa razoável! Favor entrar com um
      valor maior ou igual a 2KB e menor ou igual a 64KB.\n");
12        }
13        if(check_tamanho_memoria % 2){
14            printf("Valor não múltiplo de 2. Favor entrar com um valor que
       seja divisível por 2.\n");
15        }
16        if(check_tamanho_memoria < 0){
17            printf("Valor negativo. Favor entrar com um valor que seja
      maior que zero.\n");
18        }
19        exit(1);
20    };
21 }
```

Listing 7: Função para verificar o tamanho da memória

A função 7 recebe um tamanho de memória como entrada e verifica se ele está dentro dos critérios válidos. Os critérios são: o tamanho deve ser um valor positivo maior que zero, múltiplo de 2 e estar na faixa razoável de 128KB a 16384KB. Se o tamanho não atender a esses critérios, a função imprime mensagens de erro específicas para indicar qual critério não foi cumprido e, em seguida, encerra a execução com um código de saída 1.

```
1  void verificar_linhas_arquivo(FILE *arquivo_entrada) {
2      // Lê cada linha do arquivo
3      // Variável para armazenar a linha lida do arquivo
4      char linha[100];
5      // Variável para armazenar o endereço em formato hexadecimal de 32
      bits
6      uint32_t endereco;
7      // Variável para armazenar a operação ('R' ou 'W')
8      char operacao;
9
10     while (fgets(linha, sizeof(linha), arquivo_entrada) != NULL) {
11         // Verificar se a linha é nula
12         if (endereco == 0 && operacao == '\0') {
13             // Pular a linha nula
14             continue;
15         }
16
17         // Verificar o formato da linha
18         if (scanf(linha, "%x %c", &endereco, &operacao) != 2) {
```

```
19              // A linha não está no formato correto
20              printf("Uma linha fora do formato esperado foi encontrada.
    Favor verificar seu arquivo de entrada!\n");
21              printf("O programa será encerrado agora!\n");
22              exit(1);
23          }
24
25          // Verificar se o endereço está dentro do intervalo esperado
26          if (!(endereco >= 0x00000000 && endereco <= 0xFFFFFFFF)) {
27              // O endereço não está no formato correto
28              printf("Um endereço %x fora do formato esperado foi encontrado
    . Favor verificar seu arquivo de entrada!\n", endereco);
29              printf("O programa será encerrado agora!\n");
30              exit(1);
31          }
32
33          // Verificar se a operação é válida ('R' ou 'W')
34          if (operacao != 'R' && operacao != 'W') {
35              // A operação não é válida
36              printf("Uma operação inválida foi encontrada. Favor verificar
    seu arquivo de entrada!\n");
37              printf("O programa será encerrado agora!\n");
38              exit(1);
39          }
40      }
41      // Voltar o ponteiro para o início do arquivo
42      fseek(arquivo_entrada, 0, SEEK_SET);
43      return;
44  }
```

Listing 8: Função imprimir o relatório com estatíticas

A função 8 organiza e imprime as informações relevantes do simulador de memória, incluindo os parâmetros de entrada e as estatísticas coletadas durante a execução, como o número de page faults e o número de páginas escritas.

## 3.4 Função de determinação do índice da página

```
1  int determinar_pagina(int tamanho_quadro_memoria, unsigned addr) {
2      unsigned s, tmp;
3
4      // Converter tamanho do quadro de memória de kilobytes para bytes
5      tmp = tamanho_quadro_memoria * 1024;
6      // Variável para armazenar o número de bits necessários para
    representar o tamanho do quadro de memória
7      s = 0;
8
9      while (tmp > 1) {
10         // Deslocar o valor de 'tmp' uma posição para a direita (
    equivalente a dividir por 2)
```

```
11          tmp = tmp >> 1;
12          // Incrementar 's' em 1 a cada iteração, contando o número de bits
        necessários para representar o tamanho do quadro
13          s++;
14      }
15
16      // Calcular o número da página a partir do endereço de 32 bits
17      int pagina = addr >> s;  // Desloca o endereço 'addr' para a direita
        em 's' posições (remove os bits menos significativos)
18
19      // Retornar o número da página correspondente ao endereço
20      return pagina;
21  }
```

Listing 9: Varíaveis globais

A função 9 determina o número da página a partir de seu endereço. Ela recebe um tamanho de quadro de memória e um endereço de memória como entrada e calcula o número de bits necessários para representar o tamanho do quadro de memória. Em seguida, utiliza esse número de bits para deslocar o endereço de memória e obter o número da página correspondente. Por fim, retorna o número da página calculado.

## 3.5 Funções de algoritmos substituição de páginas

```
1  int substituicao_fifo(Pagina *tabela_de_paginas, Quadro *memoria_fisica,
       int indice_pagina, int numero_quadros) {
2      // Implementação do algoritmo FIFO (First-In, First-Out)
3      // Armazenar o índice da página que será substituída (a primeira pá
       gina inserida)
4      int pagina_reposta = memoria_fisica[0].indice;
5
6      // Deslocar os índices das páginas uma posição para a esquerda
7      for (int i = 1; i < numero_quadros; i++) {
8          memoria_fisica[i - 1].indice = memoria_fisica[i].indice;
9      }
10
11     // Substituir o último quadro pelo novo
12     memoria_fisica[numero_quadros - 1].indice = indice_pagina;
13
14     // Retornar o índice da página que foi substituída
15     return pagina_reposta;
16 }
```

Listing 10: Implementação lgoritmo de Substituição FIFO

A função 10 implementa o algoritmo FIFO (First-In, First-Out) para substituição de páginas em uma memória virtual. Ela recebe como entrada a tabela de páginas, a memória física, o índice da página a ser inserida e o número de quadros disponíveis na memória física.A função realiza o deslocamento dos índices das páginas

na memória física, inserindo o índice da nova página no último quadro. Em seguida, a função retorna o índice da página que foi substituída.

```
1
2  int substituicao_lru(Pagina *tabela_de_paginas, Quadro *memoria_fisica,
       int indice_pagina, int numero_quadros, int tempo_atual) {
3      // Implementação do algoritmo LRU (Least Recently Used)
4      // Variável para armazenar o índice do quadro com a página menos
       recentemente utilizada
5      int min_index = 0;
6      // Armazenar o tempo de acesso da primeira página na memória física
7      int min_access_time = memoria_fisica[0].ultimo_acesso;
8
9      // Encontrar o quadro com o menor tempo de acesso (página menos
       recentemente utilizada)
10     for (int i = 1; i < numero_quadros; i++) {
11         if (memoria_fisica[i].ultimo_acesso < min_access_time) {
12             // Atualiza o menor tempo de acesso
13             min_access_time = memoria_fisica[i].ultimo_acesso;
14             // Armazena o índice do quadro com o menor tempo de acesso
15             min_index = i;
16         }
17     }
18
19     // Armazenar o índice da página que será substituída
20     int pagina_reposta = memoria_fisica[min_index].indice;
21
22     // Substituir o quadro encontrado pelo novo
23     memoria_fisica[min_index].indice = indice_pagina;
24     // Atualizar o tempo de acesso do quadro para o tempo atual
25     memoria_fisica[min_index].ultimo_acesso = tempo_atual;
26
27     // Retornar o índice da página que foi substituída pelo algoritmo LRU
28     return pagina_reposta;
29 }
```

Listing 11: Implementação lgoritmo de Substituição LRU

A função 11 implementa o algoritmo LRU (Least Recently Used) para substituição de páginas em uma memória virtual. Ela encontra o quadro com o menor tempo de acesso (página menos recentemente utilizada) na memória física. A função armazena o índice dessa página como "pagina_reposta". Em seguida, substitui o quadro encontrado pelo novo, atualizando o índice da nova página e o tempo de acesso para o tempo atual. Por fim, a função retorna o índice da página substituída.

```
1  int substituicao_random(Pagina *tabela_de_paginas, Quadro *memoria_fisica,
       int indice_pagina, int numero_quadros) {
2      // Implementação do algoritmo aleatório
3      // Gerar uma posição aleatória dentro do número de quadros
4      int random_position = rand() % numero_quadros;
```

```
    // Armazenar o índice da página que será substituída
    int pagina_reposta = memoria_fisica[random_position].indice;

    // Substituir um quadro aleatório pelo novo índice
    memoria_fisica[random_position].indice = indice_pagina;

    // Retornar o índice da página que foi substituída aleatoriamente
    return pagina_reposta;
}
```

Listing 12: Implementação do algoritmo de Substituição Random

A função 12 implementa o algoritmo de substituição aleatório para substituição de páginas em uma memória virtual. Ela gera uma posição aleatória dentro do número de quadros disponíveis na memória física. Em seguida, armazena o índice da página que será substituída e substitui o quadro selecionado aleatoriamente pelo índice da nova página. Por fim, retorna o índice da página substituída.

```
int substituicao_segunda_chance(Pagina *tabela_de_paginas, Quadro *
    memoria_fisica, int indice_pagina, int numero_quadros) {
    // Implementação do algoritmo Segunda Chance (Second Chance)
    // Variável para indicar se ocorreu a substituição
    int subs = 0;
    // Variável para armazenar o índice do quadro atual
    int current_frame = -1;

    while (subs == 0) {
        // Encontrar a página do próximo FIFO
        current_frame = memoria_fisica[proximo_fifo].indice;
        if (tabela_de_paginas[current_frame].bit_ref == 1) {
            // Página ganha uma segunda chance
            // Zera o bit de referência da página
            tabela_de_paginas[current_frame].bit_ref = 0;
            // Circular para o próximo quadro
            proximo_fifo = (proximo_fifo + 1) % numero_quadros;
        } else {
            // Ocorreu a substituição
            subs = 1;
            // Substituir o quadro selecionado pelo novo
            memoria_fisica[proximo_fifo].indice = indice_pagina;
        }
    }
    // Retorna o índice do quadro substituído
    return current_frame;
}
```

Listing 13: Implementação do algoritmo de Substituição Segunda Chance

A função 13 implementa o algoritmo de substituição Segunda Chance (Second Chance) para substituição de páginas em uma memória virtual. Ela itera sobre os quadros de

memória de acordo com a ordem FIFO e verifica se o bit de referência da página é igual a 1. Se for, a página recebe uma segunda chance e o próximo quadro é verificado. Caso contrário, ocorre a substituição de página, armazenando o índice da nova página no quadro selecionado. A função retorna o índice da página substituída.

## 3.6 Leitura de linhas e processamento de acessos à memória

A função main() atribui as entradas às variáveis correspondentes, chama as funções de verificação e validação e a de imprimir as estatísticas para o relatório, aloca memória e a libera para estruturas Quadro e Página, inicializa essas estruturas com zero em seus atributos, abre e fecha o arquivo de entrada, cria as variáveis de controle e de coleta de estatísticas e processa os acessos à memoria. Esta última parte esta em 14.

```
// Loop principal para processar os acessos ã memória
    unsigned addr;
    char rw;
    int linhas_lidas;

    // Lê o endereço e a operação (R ou W)
    while ((linhas_lidas = fscanf(fptr, "%x %c", &addr, &rw)) != EOF) {

    // Verificações de linha, descritas anteriormente estão aqui,
    // mas foram omitidas por questão de espaço

        acessos_totais++;

        // Incrementar o contador de tempo
        clock++;

        // Determinar o número da página a partir do endereço
        int indice_pagina = determinar_pagina(tamanho_quadro_memoria, addr
    );

        // Setar o bit de referência da página para 1
        tabela_de_paginas[indice_pagina].bit_ref = 1;

        // Sempre que uma operação de escrita é realizada em uma página na
     memória,
        // ela fica suja
        if (rw == 'W' || rw == 'w' ) {
            acessos_escrita++;
            tabela_de_paginas[indice_pagina].suja = 1;
        }
        if (rw == 'R'|| rw == 'r') {
            acessos_leitura++;
        }

```

```
33          if (debug){
34              printf("Linhas lidas: %d\n", acessos_totais);
35              printf("Acessando página: %d\n", indice_pagina);
36          }
37
38          // Verifica se a página está na memória física
39          if (tabela_de_paginas[indice_pagina].referencia != 0){
40              // Página já está na memória física
41              // Procurar o quadro ao qual ela está alocada
42              int aux = -1;
43              for(int i = 0; i <numero_quadros; i++){
44                  if(memoria_fisica[i].indice == indice_pagina){
45                      aux = i;
46                  }
47              }
48              if (debug){
49                  printf("Página já está na memória física (Sem page fault).
     Está alocada no quadro: %d\n", aux);
50              }
51
52              // Atualizar último acesso da página e do quadro
53              tabela_de_paginas[indice_pagina].ultimo_acesso = clock;
54              memoria_fisica[aux].ultimo_acesso = clock;
55          } else {
56              // Página não está na memória física, ocorre um page fault
57              if (debug){
58                  printf("Página não está na memória física! Page fault\n");
59              }
60
61              // Incrementar o contador de page faults
62              num_page_faults++;
63
64              // Verificar se há um quadro vazio na memória física
    procurando algum quadro desocupado
65              int quadro_vazio = -1;
66              for (int i = 0; i < numero_quadros; i++) {
67                  if (memoria_fisica[i].ocupado == 0) {
68                      if (debug){
69                          printf("Quadro vazio encontrado! Memória alocada
    ao quadro: %d\n", i);
70                      }
71                      // Se achou um quadro livre, atualiza a memória física
     colocando o quadro como ocupado,
72                      // o índice como o índice da página e o último acesso
    com o clock atual
73                      memoria_fisica[i].ocupado = 1;
74                      memoria_fisica[i].indice = indice_pagina;
75                      memoria_fisica[i].ultimo_acesso = clock;
76                      quadro_vazio = 1;
77                      break;
```

```
78                  }
79              }
80              // Caso não exista um quadro vazio, utiliza-se o algoritmo de substituição selecionado para
81              // substituir a página
82              int pagina_reposta = -1;
83              if (quadro_vazio == -1) {
84                  if (debug){
85                      printf("Sem quadro vazio! Vamos usar uma técnica de reposição\n");
86                  }
87                  if (strcmp(algoritmo_substituicao, "fifo") == 0) {
88                      pagina_reposta = substituicao_fifo(tabela_de_paginas, memoria_fisica, indice_pagina, numero_quadros);
89                  } else if (strcmp(algoritmo_substituicao, "lru") == 0) {
90                      pagina_reposta = substituicao_lru(tabela_de_paginas, memoria_fisica, indice_pagina, numero_quadros, clock);
91                  } else if (strcmp(algoritmo_substituicao, "random") == 0) {
92                      pagina_reposta = substituicao_random(tabela_de_paginas, memoria_fisica, indice_pagina, numero_quadros);
93                  } else if (strcmp(algoritmo_substituicao, "2a") == 0) {
94                      pagina_reposta = substituicao_segunda_chance(tabela_de_paginas, memoria_fisica, indice_pagina, numero_quadros);
95                  }
96
97                  if(tabela_de_paginas[pagina_reposta].suja == 1){
98                      // Página teve que ser escrita de volta no disco
99                      num_dirty_pages++;
100                 }
101
102                 // Atualizar a tabela de páginas com o processo que não está presente
103                 // Setando seus atributos como zero
104                 tabela_de_paginas[pagina_reposta].referencia = 0;
105                 tabela_de_paginas[pagina_reposta].ultimo_acesso = 0;
106                 tabela_de_paginas[pagina_reposta].suja = 0;
107                 tabela_de_paginas[pagina_reposta].bit_ref = 0;
108
109                 if (debug){
110                     printf("Página %d substituída\n", pagina_reposta);
111                 }
112             }
113
114             // Atualizar a tabela de páginas com os valores para o novo processo
115             tabela_de_paginas[indice_pagina].referencia = 1;
116             tabela_de_paginas[indice_pagina].ultimo_acesso = clock;
117         }
```

```
118     }
```

Listing 14: Implementação do processamento de acessos à memória

O código possui um loop principal que processa os acessos à memória, lendo endereços e operações (leitura ou escrita) de um arquivo. A cada iteração do loop, o contador de tempo é incrementado e o código lê o endereço e a operação correspondente. Em seguida, ele determina o número da página associada ao endereço, usando essa informação para indexar a tabela de páginas. Após isso, o bit de referência da página é definido como 1, para que ela tenha uma segunda chance no algoritmo de substituição Segunda Chance. Se a operação for uma escrita, o contador de acessos de escrita é incrementado e a página é marcada como "suja". No caso de uma leitura, o contador de acessos de leitura é incrementado. Se o modo de depuração estiver ativado, o código imprime informações relevantes para auxiliar na depuração. Em seguida, é verificado se a página está presente na memória física:

- Se estiver presente, o código localiza o quadro de memória correspondente à página e atualiza o último acesso tanto da página quanto do quadro.
- Caso a página não esteja presente, ocorre um page fault, indicando que a página precisa ser buscada na memória física. Nesse caso, o contador de page faults é incrementado.

Caso ocorrra um page fault, é verificado se há um quadro vazio disponível na memória física:

- Se houver um quadro vazio, o código aloca esse quadro para a página atual, atualizando as informações do quadro e marcando-o como ocupado.
- Se não houver um quadro vazio, o código utiliza um algoritmo de substituição (como FIFO, LRU, Random ou Segunda chance) para selecionar uma página a ser substituída.
  - Se a página substituída estiver marcada como suja, o contador de páginas sujas que precisam ser escritas de volta ao disco é incrementado.
  - Além disso, a tabela de páginas é atualizada, retirando-se a página substituída, tendo seus atributos (refereência, último acesso, suja e bit_ref) zerados

A tabela de páginas é atualizada com as informações da nova página, definindo a referência como 1 e o último acesso como o tempo atual e o loop principal continua até que todos os acessos à memória tenham sido processados.

# 4 Decisões de projeto

Foram encontradas duas ambiguidades na especificação do trabalho.

A primeira está relacionada com a entrada executada no terminal: caso ela não

seja feita de acordo com o esperado não foi indicado o que o programa deve fazer. Assim, foi decidido que o código realizará diversas checagens quanto às entradas inseridas nele e o programa é encerrado caso algo não esteja conforme o esperado, retornando-se uma mensagem de erro especificando seu motivo.

A segunda está relacionada ao arquivo de entrada: caso alguma linha não esteja no formato esperado (endereço hexadecimal de 32 bits seguido por 'R' ou 'W'), não ficou claro como o programa deve se comportar. Dessa forma, decidiu-se implementar, dentro do loop de leitura do arquivo, verificações sobre as linhas do programa. Desse modo, 15 verifica se linha está no formato endereço hexadecimal de 32 bits seguido por 'R' ou 'W'. Caso algo esteja fora dos parões esperados, o motivo do erro é impresso e encerra-se a execução do programa. Tentou-se implementar uma função para realizar esse procedimento mas o desempenho do código em termos de tempo de execução piorou bastante, de modo que a implementação da forma como foi feita teve um impacto menor nessa métrica.

```
1  if (linesRead != 2) {
2              // A linha não está no formato correto
3              printf("Uma linha fora do formato esperado foi encontrada.
       Favor verificar seu arquivo de entrada!\n");
4              printf("O programa será encerrado agora!\n");
5              exit(1);
6          }
7
8          uint32_t endereco = addr;
9          // Verificar se o endereço está dentro do intervalo esperado
10         if (!(endereco >= 0x00000000 && endereco <= 0xFFFFFFFF)) {
11             // O endereço não está no formato correto
12             printf("Um endereço %x fora do formato esperado foi encontrado
       . Favor verificar seu arquivo de entrada!\n", endereco);
13             printf("O programa será encerrado agora!\n");
14             exit(1);
15         }
16
17         // Verificar se a operação é válida ('R' ou 'W')
18         if (rw != 'R' && rw != 'W') {
19             // A operação não é válida
20             printf("Uma operação inválida foi encontrada. Favor verificar
       seu arquivo de entrada!\n");
21             printf("O programa será encerrado agora!\n");
22             exit(1);
23         }
```

Listing 15: Verificação do arquivo de entrada

# 5 Análise do desempenho dos algoritmos

Nesta seção, será realizada uma análise do desempenho dos quatro algoritmos de substituição de páginas considerados: FIFO, LRU, Random e Segunda chance. O comportamento desses algoritmos será avaliado em diferentes configurações de tamanho de memória e tamanho de página.

## 5.1 Algoritmos e expectativas gerais

- FIFO (First-In, First-Out):

  O algoritmo FIFO substitui a página que está na memória há mais tempo e mantém uma fila das páginas na ordem em que foram carregadas na memória. Quando uma página precisa ser substituída, o algoritmo seleciona a página que está no início da fila, ou seja, a primeira que foi carregada. O desempenho do FIFO é simples e ele é fácil de implementar, mas não leva em consideração o uso atual das páginas. Isso pode elevar o número de page faults, especialmente em casos em que páginas frequentemente referenciadas são substituídas.

  **Desempenho esperado:** O desempenho do FIFO pode ser bom em casos em que os padrões de acesso às páginas sejam uniformes e as páginas referenciadas recentemente não sejam necessárias no futuro próximo. No entanto, em cenários com padrões de acesso complexos ou páginas frequentemente referenciadas, o FIFO pode ter um desempenho ruim.

  - **Page Fault:** O FIFO pode ter um alto número de page faults em cenários onde páginas frequentemente referenciadas são substituídas por páginas mais antigas.
  - **Páginas escritas:** O algoritmo FIFO não considera a sujeira das páginas. Portanto, não tem impacto direto no tratamento de páginas sujas.
  - **Tempo de Execução:** O FIFO é relativamente eficiente em termos de tempo de execução, pois requer apenas a manipulação de uma fila simples. No entanto, a alta ocorrência de page faults pode aumentar o tempo total de execução do sistema.

- LRU (Least Recently Used):

  O algoritmo LRU substitui a página que não foi referenciada por mais tempo. Ele se baseia na premissa de que as páginas que não foram usadas há mais tempo provavelmente serão as menos usadas no futuro próximo. Para implementar o LRU, é necessário manter um registro do último acesso às páginas.

  **Desempenho esperado:** O LRU geralmente tem um bom desempenho em termos de número de page faults, pois tenta manter as páginas mais relevantes na memória. No entanto, a implementação do LRU pode ser complexa e requerer um alto custo computacional para manter o registro das páginas mais recentes. Em

sistemas com muitas páginas e acesso intensivo à memória, o LRU pode se tornar custoso.

  - **Page Fault:** O LRU tende a ter um número de page faults menor do que o FIFO, pois leva em consideração o uso atual das páginas.
  - **Páginas escritas:** O LRU não considera a sujeira das páginas no processo de substituição.
  - **Tempo de Execução:** O LRU é mais complexo do que o FIFO, pois requer o rastreamento do histórico de uso das páginas. Isso pode aumentar o tempo de execução em comparação com o FIFO, mas geralmente é considerado uma troca aceitável dada a redução nas page faults.

- Random

  O algoritmo Random seleciona aleatoriamente uma página para substituição. Ele não leva em consideração o comportamento de uso das páginas nem a ordem de chegada. O Random é geralmente usado para fins de comparação e é útil quando não há informações disponíveis sobre o uso das páginas.

  **Desempenho esperado:** O desempenho do algoritmo aleatório pode variar amplamente. Em alguns casos, pode ser surpreendentemente eficiente se a seleção aleatória coincidir com as páginas menos usadas. No entanto, devido à natureza aleatória, pode ter um baixo desempenho.

  - **Page Fault:** Random pode ter um número de page faults variável, dependendo da aleatoriedade da seleção das páginas.
  - **Páginas escritas:** Assim como os outros algoritmos mencionados anteriormente, o Random não considera a sujeira das páginas durante a substituição.
  - **Tempo de Execução:** O Random é simples de implementar e tem um tempo de execução eficiente, já que não requer rastreamento do histórico de uso das páginas.

- Segunda chance

  O algoritmo de segunda chance é uma variação do FIFO que adiciona um bit adicional a cada página para rastrear se ela foi recentemente referenciada. Quando uma página precisa ser substituída, o algoritmo verifica esse bit. Se o bit estiver definido como 1, a página recebe uma 'segunda chance' e é movida para o final da fila. Se o bit estiver definido como 0, a página é substituída normalmente.

  **Desempenho esperado:** O desempenho do algoritmo de segunda chance fica entre o FIFO e o LRU. Ele melhora o desempenho em comparação com o FIFO ao evitar a remoção imediata de páginas recentemente referenciadas. No entanto, pode não ser tão eficiente quanto o LRU em cenários com padrões de acesso mais complexos.

  - **Page Fault:** O Segunda Chance tem um numero de page faults inter-

mediário entre o FIFO e o LRU. Ele oferece um desempenho melhor do que o FIFO em termos dessa métrica, mas não tão bom quanto o LRU.

- **Páginas escritas:** O Segunda Chance não leva em consideração a sujeira das páginas durante a substituição.
- **Tempo de Execução:** O Segunda Chance requer uma lógica adicional para verificar se as páginas foram referenciadas recentemente, mas geralmente tem um tempo de execução semelhante ao FIFO.

Em resumo, em termos de page faults, o LRU geralmente apresenta o melhor desempenho, seguido pelo Segunda Chance e FIFO, enquanto o Random é imprevisível. Quanto às páginas escritas, nenhum desses algoritmos considera diretamente esse aspecto durante a substituição. Em relação ao tempo de execução, o FIFO, o Segunda Chance e o Random são mais eficientes do que o LRU, que requer o rastreamento do histórico de uso das páginas. No entanto, é importante notar que o desempenho dos algoritmos pode variar dependendo do cenário de uso específico e das características do sistema.

## 5.2 Metodologia de Avaliação

Para avaliar o desempenho dos algoritmos de substituição de páginas serão executados programas projetados para simular o comportamento dos quatro algoritmos definidos para diferentes padrões de acesso à memória. Esses cenários estão descritos a seguir:

- Tamanho da memória crescente com páginas de 4 KB:

  Nessa configuração, o tamanho da memória utilizada foi variado dentro da faixa de valores razoáveis e o tamanho das páginas foi mantido constante em 4 KB. Foi avaliado o desempenho dos algoritmos com os seguintes tamanhos de memória de 128KB, 256KB, 512KB, 1024KB, 2048KB, 4096KB, 8192KB e 16384KB

- Tamanho da memória total de 256KB com variação do tamanho das páginas:

  Nessa configuração, o tamanho da memória ficou constante em 256KB e tamanho das páginas foi variado variado dentro da faixa de valores razoáveis. Foi avaliado do desempenho dos algoritmos com os seguintes tamanhos de página 2KB, 4KB, 8KB, 16KB, 32KB e 64KB.

Durante a execução dos programas de teste nas diferentes configurações mencionadas, foram coletadas as seguintes métricas:

- **Número de page faults:** O número de page faults indica quantas vezes uma página precisou ser buscada na memória secundária. Um menor número de page faults geralmente indica um desempenho melhor do algoritmo, pois significa que menos páginas precisaram ser substituídas. Portanto, quanto menor o número de page faults, melhor o desempenho do algoritmo.
- **Número de páginas escritas:** O número de páginas escritas indica quantas

páginas foram modificadas e precisaram ser escritas de volta na memória secundária. Um menor número de páginas escritas é desejável, pois indica uma menor carga no sistema de armazenamento secundário. Isso pode ser relevante em situações em que a escrita de páginas é um processo custoso, como em dispositivos de armazenamento em disco. Portanto, algoritmos que resultam em um menor número de páginas escritas podem ser considerados mais eficientes nesse aspecto.

- **Tempo de execução:** O tempo de execução mede o tempo total necessário para executar o simulador com uma configuração pre-definida. Comparar o tempo de execução dos algoritmos pode ajudar a identificar qual deles é mais eficiente em termos de desempenho geral.

Para garantir que os testes fossem realizados sob as mesmas condições, um arquivo shell foi escrito de forma a automatizar os dois testes com as configurações pré-definidas.

## 5.3 Resultados

O resultado dos testes foi salvo em um arquivo de extensão csv que foi utilizado para plotar os gráfico desta seção. Os gráficos de uma mesma métrica para cada cenário possuem as mesmas escalas nos eixos x e y, de modo que é possível compará-los visualmente. Além disso, o eixo x representa o tamanho da memória (teste um) ou da página (teste dois) e o eixo y a métrica que se deseja medir.

De forma geral, à medida que o tamanho da memória aumenta em um cenário de tamanhos de páginas constante, é esperado que o número de page faults diminua para todos os algoritmos, pois é possível manter uma maior quantidade de páginas ativas na memória, o que reduz a necessidade de buscar páginas no disco e carregá-las na memória física. Da forma contrária, ao aumentar o tamanho das páginas em um ambiente de memória física constante, espera-se um aumento no número de page faults, pois menos páginas de serão mantidas em memória de uma vez. Consequentemente, à medida que o programa faz referência a páginas que não estão presentes na memória física, mais page faults ocorrerão, pois essas páginas precisarão ser buscadas no disco e carregadas na memória.

Em relação ao número de paginas escritas a expectativa é que, mantendo-se o tamanho da página constante, com um tamanho total de memória maior, mais páginas podem ser armazenadas na memória física, reduzindo a taxa de substituição de páginas e, consequentemente, potencialmente reduzindo o número de páginas sujas. Na configuração de memória total constante, à medida que o tamanho da página aumenta, o número de páginas sujas pode diminuir, pois, com um tamanho de página maior, mais dados são modificados em uma única operação de escrita e, consequentemente, menos páginas individuais precisam ser atualizadas e escritas de volta no disco, resultando em menos páginas sujas. Entretanto, o número de páginas escritas também é influenciado

pelo arquivo de entrada. Programas que realizam muitas operações de escrita tendem a ter um número maior de páginas sujas e programas com padrões de acesso aleatórios ou fragmentados podem levar a mais modificações de páginas e, portanto, a um maior número de páginas sujas.

Além disso, em termos de tempo de execução, algoritmos mais complexos, como LRU e Segunda Chance, exigem mais processamento para manter o registro do histórico de acesso às páginas, o que pode levar a um tempo de execução maior em comparação com FIFO e Random. Por outro lado, eles oferecem maior eficiência na substituição e podem resultar em um melhor desempenho em termos de tempo de execução, pois tentam manter na memória as páginas mais frequentemente acessadas.

### 5.3.1 Primeiro cenário

1. Page faults

   Para o primeiro cenário de teste em relação às page faults, os resultados estão apresentados na figura 1. Nela, podemos perceber que independente do tipo de algoritmo de substituição de página a quantidade de page faults diminui com o aumento do tamanho da memória simulada (eixo X), pois assim o SO possui mais espaço para carregar mais páginas. Isso está condizente com o esperado.

   Dentre os algortimos, o LRU apresenta um melhor resultado para essa métrica.

2. Paginas escritas

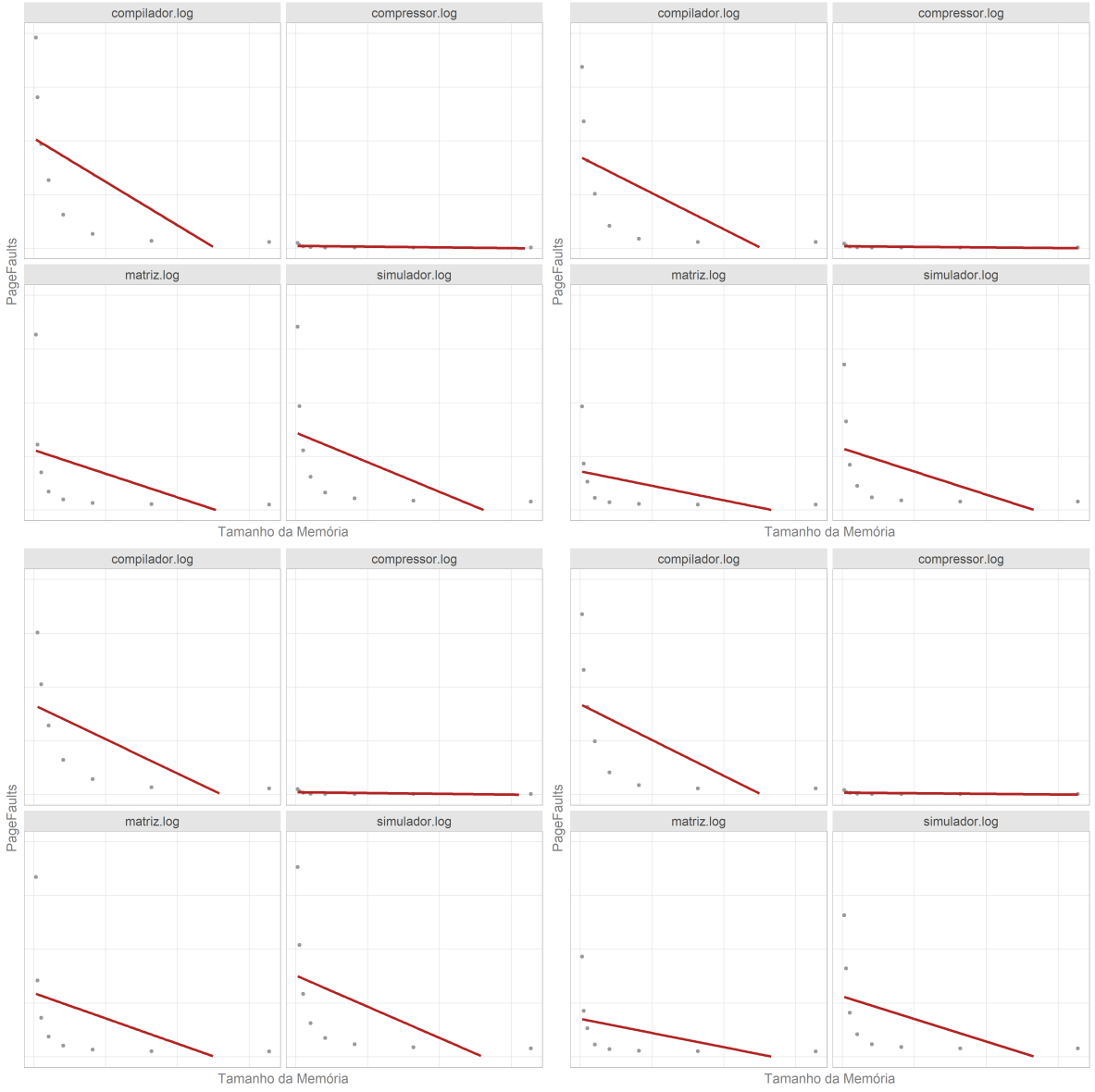
   Para o primeiro cenário de teste em relação às páginas escritas, os resultados estão apresentados na figura 2. Podemos perceber por essa análise que a quantidade de páginas escritas (eixo Y) diminui a medida que a memória aumenta (eixo X). Isso acontece pelo fato de que, quando a memória é maior, a necessidade de escrever uma nova página fica atrelada à ocorrência de page faults, que são diminuídos com o aumento da memória, como mostrado na análise acima. Esse comportamento é o esperado.

   Dentre os algortimos, o LRU e o Segunda Chance apresentam os melhores resultado para essa métrica.

3. Tempo de execução

   Para o primeiro cenário de teste em relação ao tempo de execução, os resultados estão apresentados na figura 3. Nesta figura, pode-se perceber que, a medida que a memória aumenta (eixo X), o tempo de execução aumenta proporcionalmente (eixo Y). Como a memória é aumentada e o tamanho de página é mantido o mesmo, a quantidade de páginas aumenta. Dessa forma, a tabela de página a ser conferida é muito grande, tornando a sua consulta demorada. Em geral, aumentar o tamanho da memória total disponível tende a reduzir a frequência de falhas de página, o que pode resultar em um melhor desempenho em termos de tempo de execução. Portanto, esse resultado está conforme esperado. No entanto, o

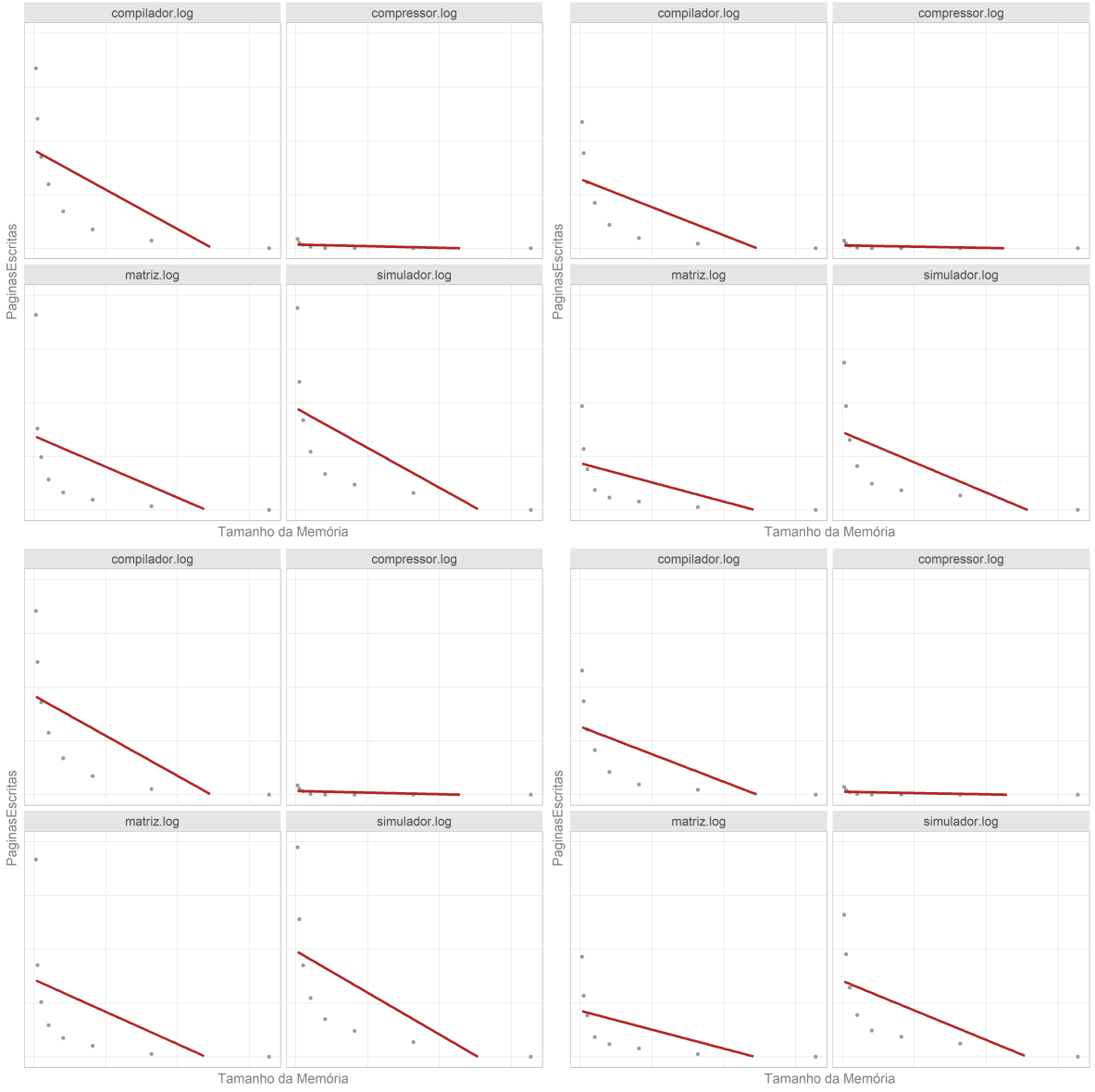
(a) FIFO (b) LRU




(c) Random (d) Segunda Chance

Figura 1: Resultados do primeiro teste para page fault

(a) FIFO (b) LRU




(c) Random (d) Segunda Chance

Figura 2: Resultados do primeiro teste para páginas escritas

tempo de acesso à memória física também pode ser afetado de forma negativa pelo aumento do tamanho da memória total disponível. Se a memória física for muito grande em relação ao tamanho da página, pode haver um aumento no tempo de acesso à memória, pois o simulador precisa percorrer uma quantidade maior de memória física para localizar uma página específica.

Os algoritmos apresentaram desempenho similar nesta métrica.

### 5.3.2 Segundo cenário

1. Page faults

   Para o segundo cenário de teste em relação às page faults, apresentado na figura 4, fica evidente que com o aumento do tamanho da página (eixo X), e mantendo o tamanho de memória, o número de page faults aumenta independente do algoritmo de substituição. Isso ocorre pois o SO fica com menos quadros disponíveis para armazenar os dados e consequentemente aumentando a chance da página desejada não estar carregada na memória, conforme esperado.

   Dentre os algortimos, o LRU e Segunda Chance apresentaram melhores resultados para essa métrica.

2. Páginas escritas

   Para o segundo cenário de teste em relação às páginas escritas, os resultados estão apresentados na figura 5. À medida que o tamanho da página aumenta (eixo X), a quantidade de páginas escritas (eixo Y) aumenta. Esse aumento, embora seja o oposto do comportamento esperado, pode ser explicado por arquivos de entrada com muitas operações de escrita ou com padrões de acesso aleatórios ou fragmentados podem levar a mais modificações de páginas.

   Dentre os algortimos, o LRU e Segunda Chance apresentaram melhores resultados para essa métrica.

3. Tempo de execução

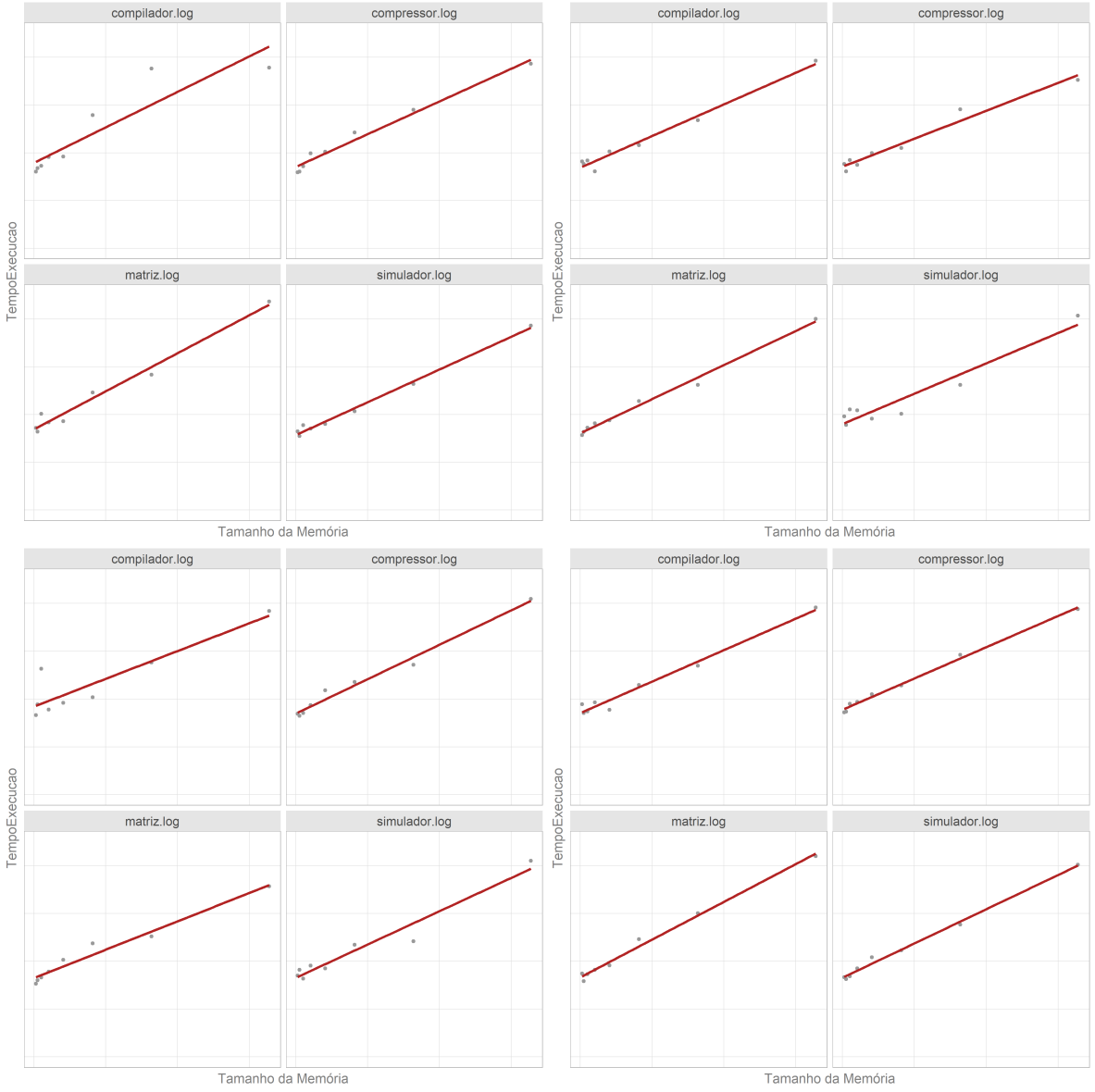
   Para o segundo cenário de teste em relação ao tempo de execução, os resultados estão apresentados na figura 6. Percebe-se por essa figura que o tempo de execução (eixo Y) sem grandes variações.

   Os algoritmos apresentaram desempenho similar nesta métrica.

# 6 Conclusão

Neste trabalho, foi realizada o desenvolvimento de um simulador de memória virtual na linguagem C em conjunto com uma análise do desempenho dos algoritmos de substituição de página FIFO, LRU, Random e Segunda Chance nesse ambiente em dois cenários de teste.

(a) FIFO (b) LRU




(c) Random (d) Segunda Chance

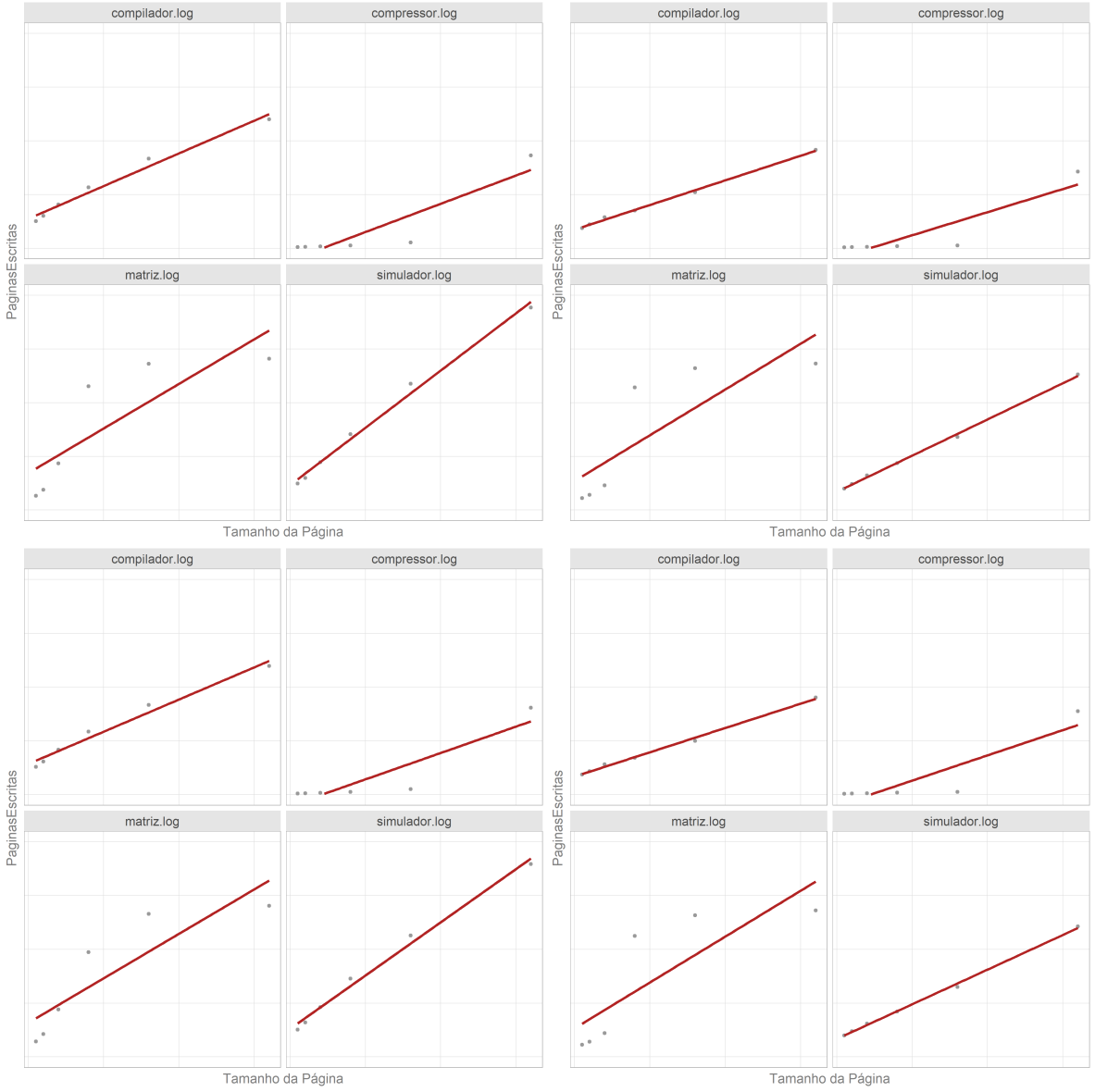
Figura 3: Resultados do primeiro teste para tempo de execução

(a) FIFO (b) LRU


(c) Random (d) Segunda Chance

Figura 4: Resultados do segundo teste para page fault

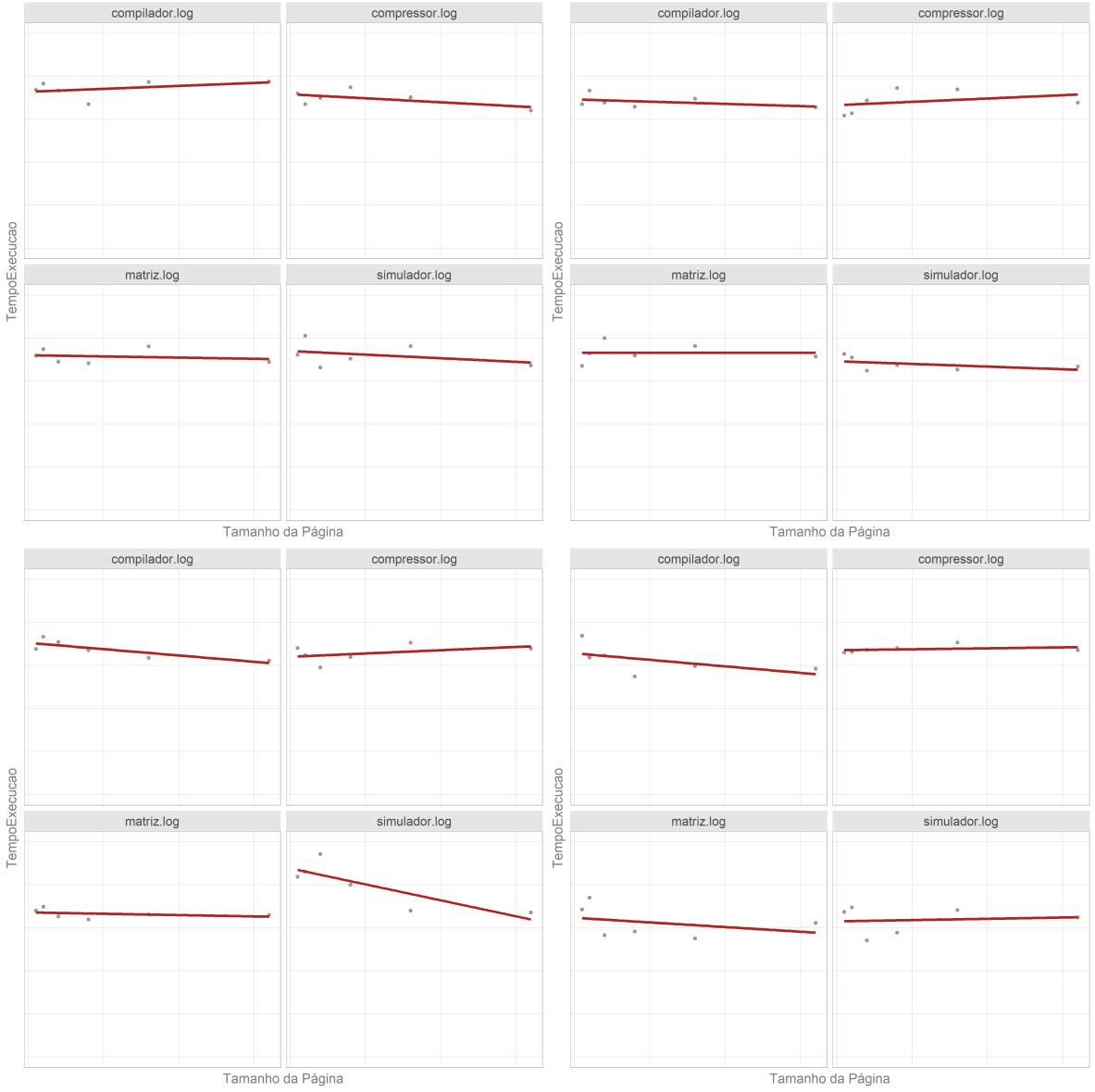
(a) FIFO (b) LRU




(c) Random (d) Segunda Chance

Figura 5: Resultados do segundo teste para páginas escritas

(a) FIFO (b) LRU




(c) Random (d) Segunda Chance

Figura 6: Resultados do segundo teste para tempo de execução

Os códigos C, H, Shell e R o csv com as saídas estão disponíveis no github no repositório `https://github.com/camilasbraz/virtual-memory-simulator`

Ao comparar os algoritmos entre si, notamos que cada um tem seus pontos fortes e fracos. O FIFO é simples de implementar, mas não considera padrões de acesso às páginas. O LRU leva em conta o histórico de acesso e geralmente apresenta um bom desempenho, mas requer uma implementação mais complexa. O Random é simples, mas pode ter um desempenho inconsistente devido à seleção aleatória das páginas. O Segunda Chance é uma variação do FIFO que leva em conta o bit de referência, sendo capaz de lidar melhor com padrões de acesso.

Em conclusão, cada algoritmo apresenta características distintas e desempenho variado dependendo do contexto em que é aplicado e a escolha de qual utilizar depende das características específicas do sistema e das demandas de desempenho.