Gerenciamento de Memória – Analise de algoritmos de substituição de pagina

1st Pablo Alessandro Santos Hugen
Centro de Ciencias Exatas e da Terra – CCET
Universidade Estadual do Oeste do Parana – UNIOESTE
Cascavel-PR, Brasil
PabloASHugen@protonmail.com

1st Felipi Lima Matozinho
Centro de Ciencias Exatas e da Terra – CCET
Universidade Estadual do Oeste do Parana – UNIOESTE
Cascavel-PR, Brasil
matozinhof.academico@gmail.com

1st Daniel Carlos Chaves Boll
Centro de Ciencias Exatas e da Terra – CCET
Universidade Estadual do Oeste do Parana – UNIOESTE
Cascavel-PR, Brasil
danielboll.academico@gmail.com

I. Introdução

A gestão de memória é um dos principais desafios enfrentados pelos sistemas operacionais modernos. A quantidade de memória disponível em um sistema não é ilimitada, e é fundamental que o sistema operacional gerencie de forma eficiente o acesso a essa memória para garantir o melhor desempenho possível.

Dentre as várias técnicas de gerenciamento de memória utilizadas pelos sistemas operacionais, a paginação é uma das mais comuns e eficientes. A técnica de paginação é utilizada para permitir que um processo acesse mais memória do que a quantidade de memória física disponível no sistema. Em outras palavras, ela permite que o sistema operacional simule um espaço de endereçamento maior do que a quantidade de memória RAM presente no sistema.

De acordo com [1], a técnica de paginação funciona dividindo a memória virtual em pequenos blocos chamados de páginas. Cada página tem um tamanho fixo, que é determinado pelo sistema operacional. O tamanho típico de uma página varia de alguns kilobytes a alguns megabytes, dependendo do sistema operacional e do hardware em questão.

Cada processo tem uma tabela responsável por mapear todas as páginas daquele processo e seu respectivo endereço na memória física. O processo acredita que está acessando uma grande quantidade de memória virtual contígua, enquanto, na verdade, ele está acessando várias páginas não contíguas na memória física.

Quando um processo tenta acessar uma página que ainda não está presente na tabela de páginas, ocorre uma falha de página (*page fault*). O sistema operacional interrompe o processo e busca a página solicitada na memória secundária (geralmente um disco rígido). Quando a página é encontrada, ela é carregada na memória física e a instrução é reiniciada. A Figura 1 mostra esse fenômeno.

A técnica de paginação traz vários benefícios para o gerenciamento de memória em sistemas operacionais. Ela permite que os processos acessem mais memória do que a quantidade de memória RAM presente no sistema, o que aumenta o desempenho e a capacidade do sistema. Além disso, a técnica de paginação permite que o sistema operacional faça um melhor uso da memória física disponível, permitindo que as páginas que não são usadas com frequência sejam armazenadas na memória secundária e carregadas apenas quando necessário.

A técnica de paginação requer algoritmos eficientes de substituição de páginas. Quando a memória física está cheia, o sistema operacional precisa escolher qual página substituir para dar lugar a uma nova página. Existem vários algoritmos de substituição de páginas disponíveis, cada um com suas vantagens e desvantagens.

O algoritmo de substituição de páginas LRU (Least Recently Used) é um dos algoritmos mais utilizados. Ele substitui a página que não foi usada há mais tempo. O algoritmo LRU é baseado no princípio de que as páginas que não foram usadas recentemente são menos prováveis de serem usadas novamente no futuro próximo. O algoritmo LRU é eficiente em sistemas com grande quantidade de memória disponível, mas pode ser menos eficiente em sistemas com pouca memória.

Outro algoritmo popular de substituição de páginas é o Counter. Ele associa um contador a cada página e escolhe a página com o menor contador para substituição. Quando uma página é referenciada, seu contador é incrementado. O algoritmo Counter é eficiente em sistemas com pouca memória, pois escolhe páginas que não foram usadas recentemente e não têm muitas referências.

O algoritmo Second Chance é uma variação do algoritmo FIFO (First In First Out). Ele dá uma segunda chance para as páginas que ainda são usadas pelo processo. O algoritmo Second Chance mantém uma lista circular das páginas na memória, onde cada uma delas tem um bit de acesso. Quando uma página é referenciada, seu bit de acesso é setado para 1. Quando o

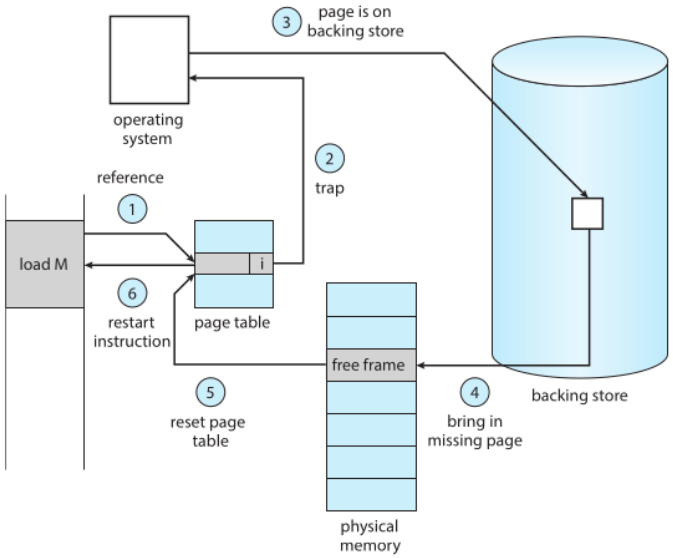


Fig. 1. Passos realizados pelo SO em uma falha de páginas [2]

sistema precisa substituir uma página, ele começa pelo início da lista e escolhe a primeira página que tem seu bit de acesso setado em 0 e caso seja 1, retorna-o para 0.

De acordo com [2], a escolha do algoritmo de substituição de páginas depende do sistema e do ambiente em que ele é executado. Cada algoritmo tem suas próprias vantagens e desvantagens e é importante escolher o algoritmo certo para cada sistema e situação

II. Materiais e métodos

Dessarte, o presente trabalho foi implementado utilizando a linguagem de programação Rust [3]. O código fonte completo pode ser visto em [4].

De forma a mimetizar o comportamento da *Memory Management Unit [MMU]*, que pode ser vista na Figura 1, a seguinte modelagem do simulador foi empregada:

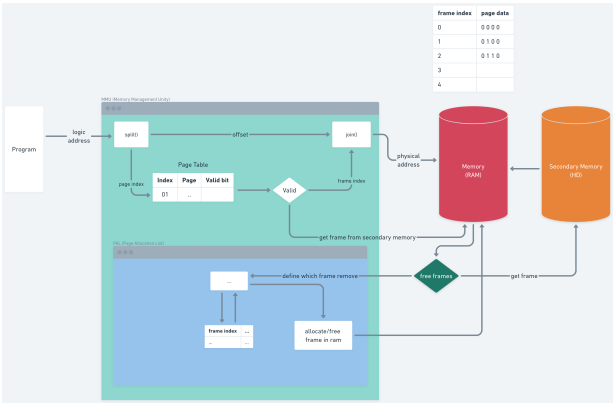


Fig. 2. Modelo simples de funcionamento de uma MMU [2]

- **MMU**: Módulo responsável pela tradução de *endereços lógicos para endereços físicos*. Basicamente é composto de um tabela de páginas e um método de tradução de endereço.

```
#[derive(Debug, Clone)]
pub struct MMU {
```

```
  pub page_table  PageTable,
  pub page_size  usize,
}
impl MMU {
  pub fn new(page_size  usize)    Self {
    Self {
      page_table  PageTable {
        entries  (0..(1 << (32 - page_size.ilog2())))
          .map(|_| PageTableEntry  default())
          .collect(),
      },
      page_size,
    }
  }
  pub fn translate(&mut self, address  &LogicalAddress)    TranslationResult {
  ...
  }
}
```

- **Memory**: Já a memória, foi modelada como um vetor de Frames na memória. Sua struct possui métodos de obtenção e alocação de frames.

```
pub struct PrimaryMemory {
  pub frames  Vec<Frame>,
  pub guard  Mutex<()>,
}

impl PrimaryMemory {
  pub fn new(size  usize)    Self {
    Self {
      frames  (0..size).map(|_| Frame  default()).collect(),
      guard  Mutex  new(()),
    }
  }

  pub fn get_frame(&self, index  usize)    bool {
    let _guard = self.guard.lock().expect("Failed to get guard");

    self.frames[index].data
  }

  pub fn alloc_frame(&mut self)    Option<usize> {
    let _guard = self.guard.lock().expect("Failed to get guard");

    for (i, frame) in self.frames.iter_mut().enumerate() {
      if !frame.data {
        frame.data = true;
        return Some(i);
      }
    }

    None
  }
```

- **PAL**: É o módulo responsável pelas implementações dos algoritmos de substituição. Esse módulo possui uma interface comum:

```
pub trait PALTable {
  fn find_frame_to_deallocate(&mut self)    usize;
```

```
  fn update_access(&mut self, frame  usize);
  fn insert(&mut self, frame  usize)    Option<usize>;
  fn clone_dyn(&self)    Box<dyn PALTable>;
  fn print(&self);
}
```

Dessa forma, cada algoritmo em específico implementa essa interface para atingir os objetivos esperados:

```
pub enum PALAlgorithm {
  LRU,
  Counter,
  SecondChance,
}
```

Tendo implementado o projeto da maneira descrita, a simulação de traduções de endereços foi conduzida. As entradas das simulações (traces) são arquivos contendo um ou mais endereços hexadecimais de memória, sendo os utilizados aqueles oferecidos pelo professor além de um contendo endereços totalmente aleatórios, utilizado como efeito de comparação dos demais traces.

Portanto, para cada algoritmo implementado e para cada trace, uma simulação foi realizada com todos os valores de frames livres na memória no intervalo $[2, 2^{16}]$.

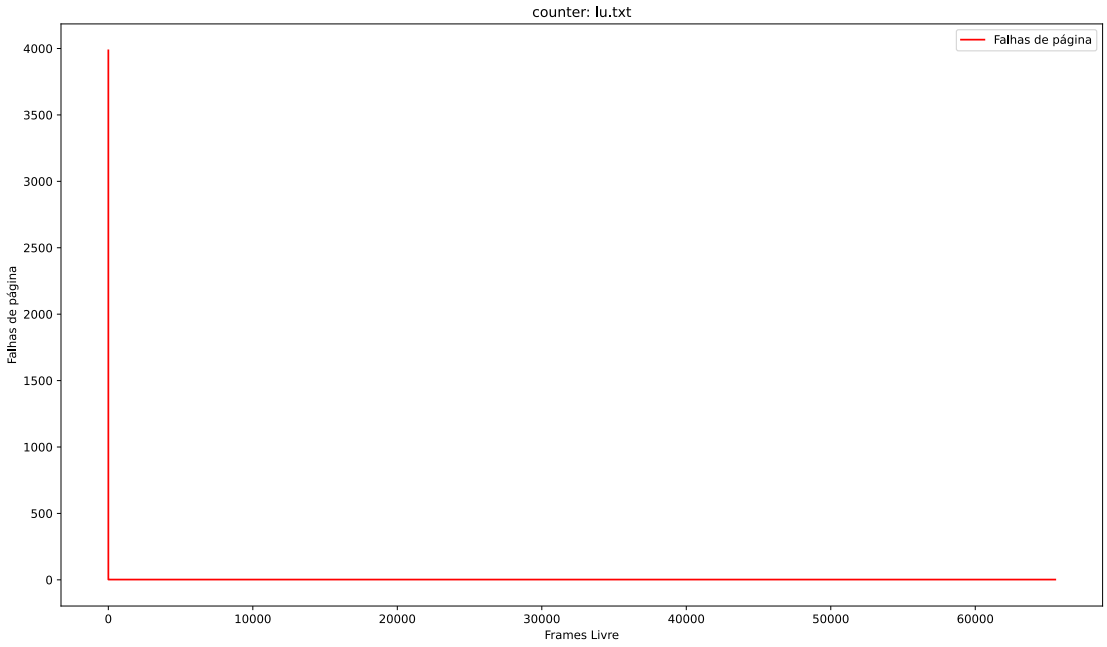
III. Objetivos

O objetivo do presente experimento foi comparar o comportamento de diferentes algoritmos de substituição de páginas frente a diferentes valores de frames livres.

IV. Resultados

Sendo assim, prosseguindo com as execuções das simulações, para cada algoritmo e para cada arquivo de trace, os resultados foram obtidos:

A. *Counter*

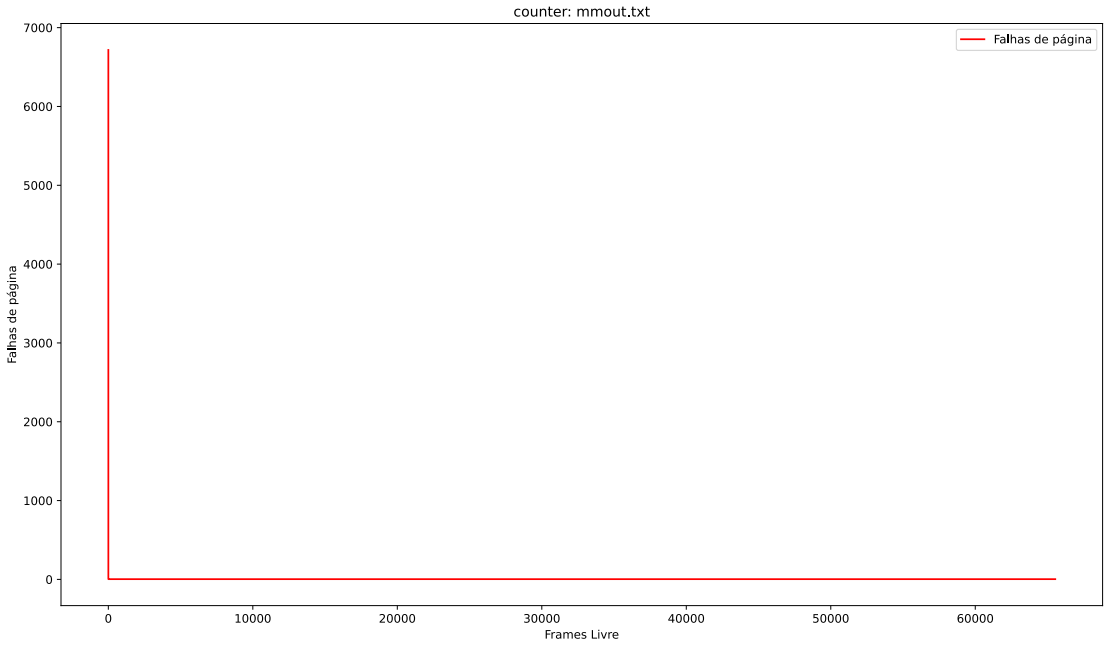


1) *lu:*

counter: lu.txt

frames livres	falhas de página
1	3986
2	2
4	2
8	2
16	2
32	2
64	2
128	2
256	2
512	2
1024	2
2048	2
4096	2
8192	2
16384	2
32768	2
65536	2

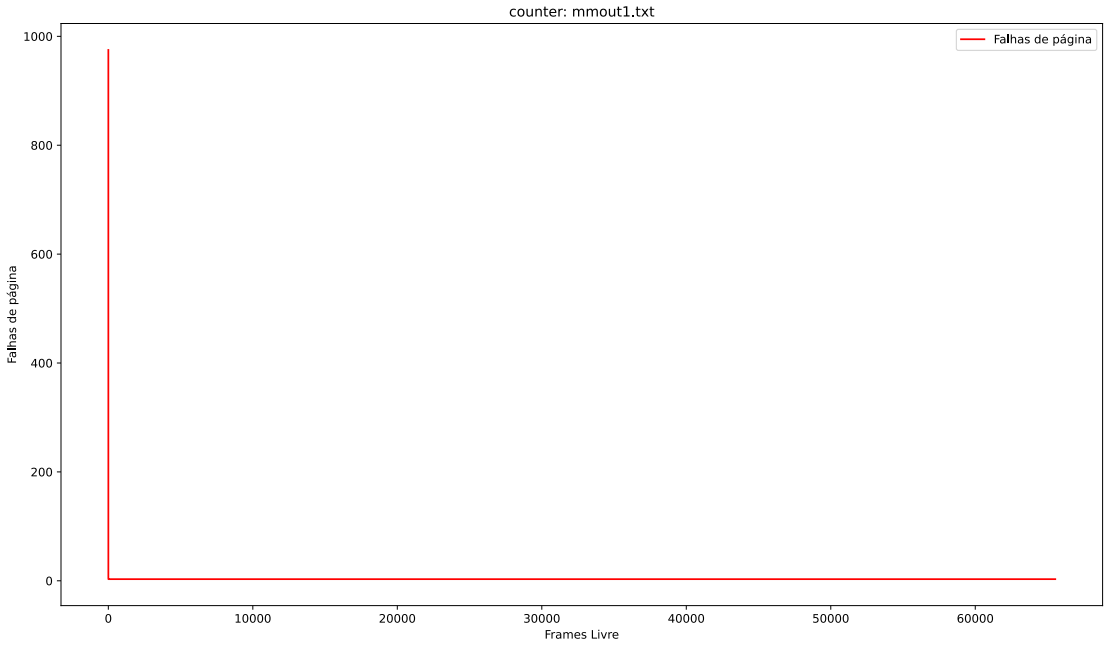
2) mmout:



counter: mmout.txt

frames livres	falhas de página
1	6717
2	6
4	3
8	3
16	3
32	3
64	3
128	3
256	3
512	3
1024	3
2048	3
4096	3
8192	3
16384	3
32768	3
65536	3

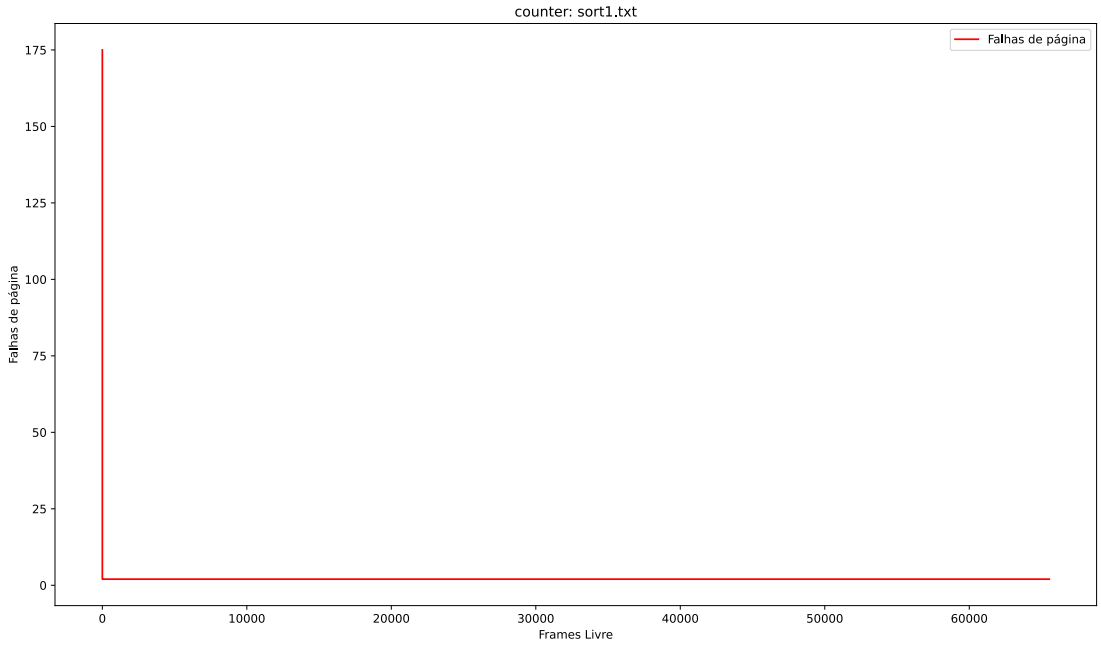
3) mmout1:



counter: mmout1.txt

frames livres	falhas de página
1	975
2	6
4	3
8	3
16	3
32	3
64	3
128	3
256	3
512	3
1024	3
2048	3
4096	3
8192	3
16384	3
32768	3
65536	3

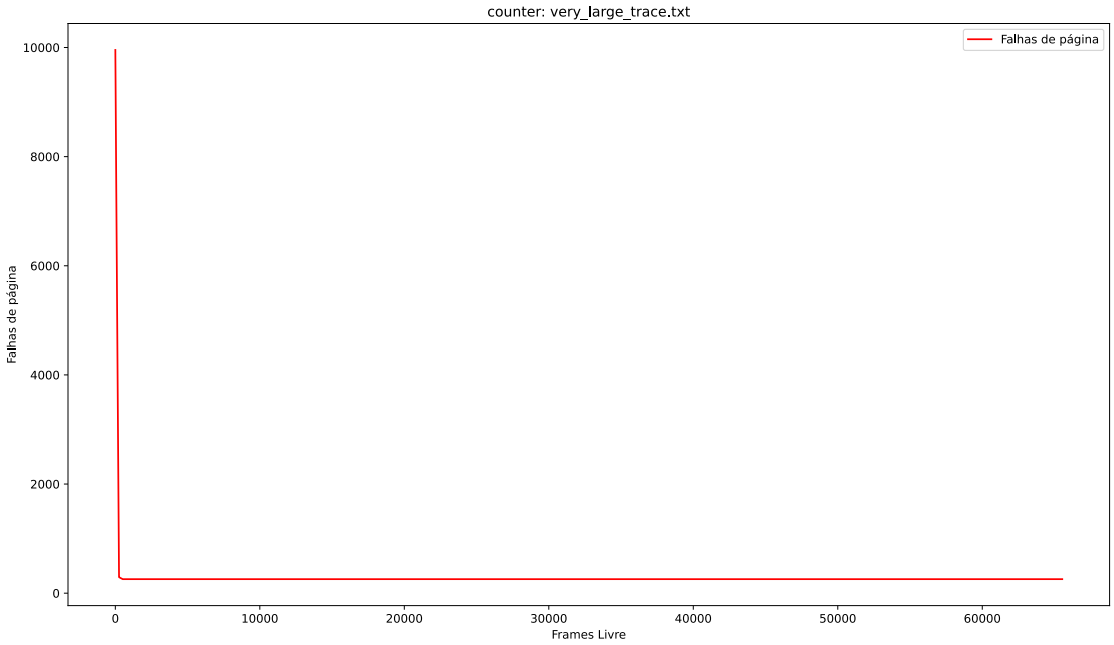
4) sort1:



counter: sort1.txt

frames livres	falhas de página
1	175
2	2
4	2
8	2
16	2
32	2
64	2
128	2
256	2
512	2
1024	2
2048	2
4096	2
8192	2
16384	2
32768	2
65536	2

5) very_large_trace:

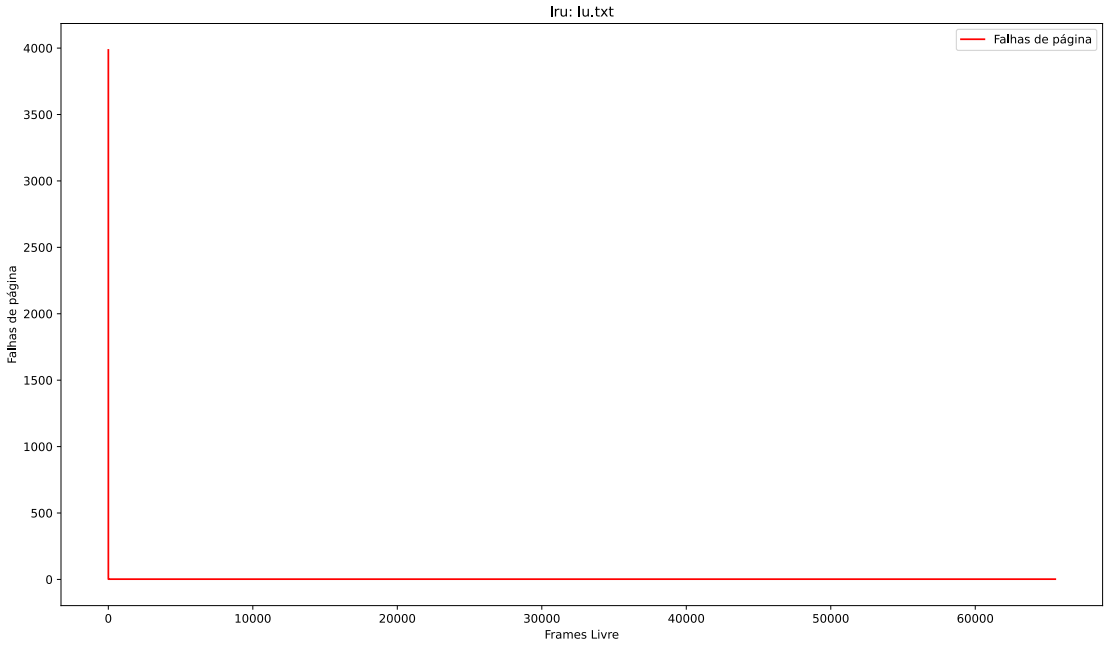


counter: very_large_trace.txt

frames livres	falhas de página
1	9955
2	9919
4	9832
8	9688
16	9399
32	8761
64	7462
128	5036
256	293
512	257
1024	257
2048	257
4096	257
8192	257
16384	257
32768	257
65536	257

B. Least Recently Used (LRU)

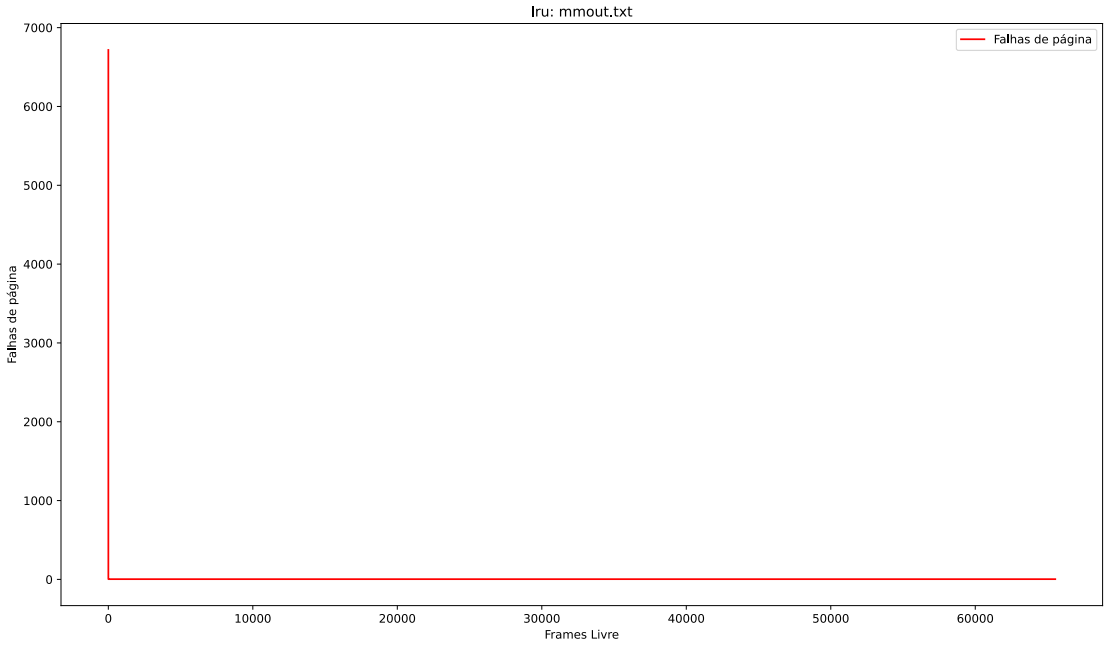
1) lu:



lru: lu.txt

frames livres	falhas de página
1	3986
2	2
4	2
8	2
16	2
32	2
64	2
128	2
256	2
512	2
1024	2
2048	2
4096	2
8192	2
16384	2
32768	2
65536	2

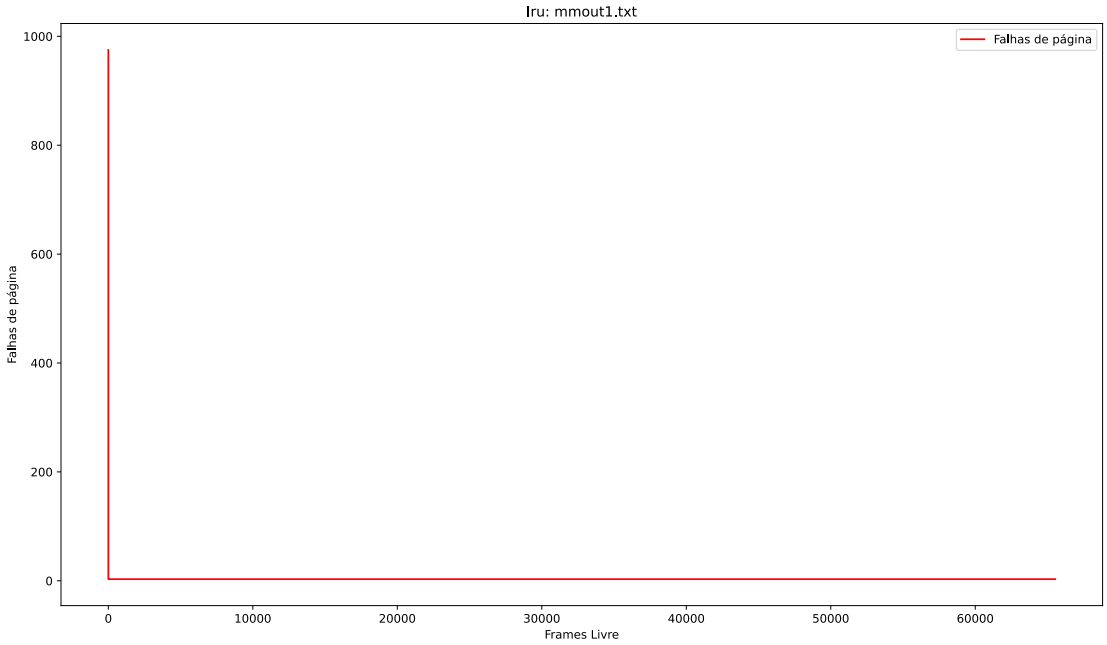
2) mmout:



lru: mmout.txt

frames livres	falhas de página
1	6717
2	5
4	3
8	3
16	3
32	3
64	3
128	3
256	3
512	3
1024	3
2048	3
4096	3
8192	3
16384	3
32768	3
65536	3

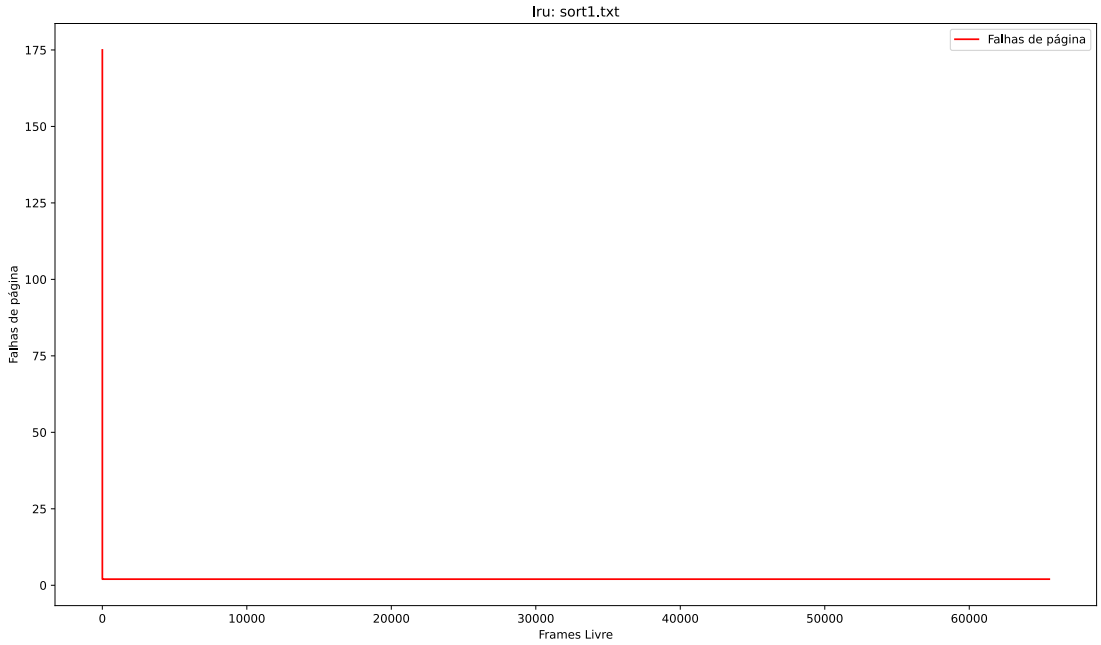
3) mmout1:



lru: mmout1.txt

frames livres	falhas de página
1	975
2	5
4	3
8	3
16	3
32	3
64	3
128	3
256	3
512	3
1024	3
2048	3
4096	3
8192	3
16384	3
32768	3
65536	3

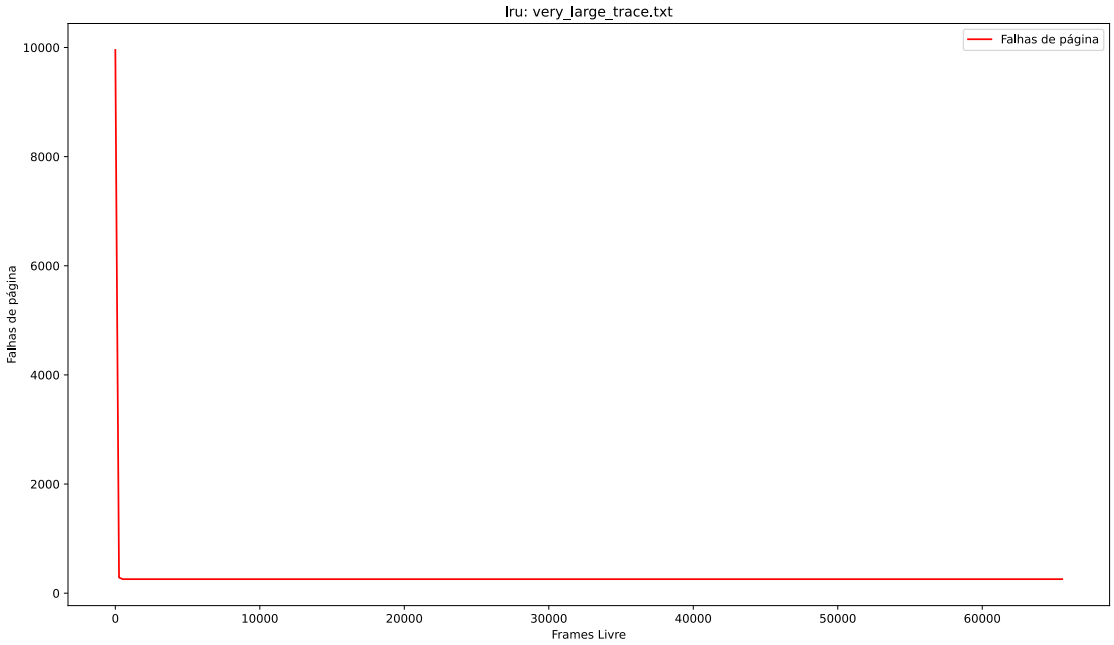
4) sort1:



lru: sort1.txt

frames livres	falhas de página
1	175
2	2
4	2
8	2
16	2
32	2
64	2
128	2
256	2
512	2
1024	2
2048	2
4096	2
8192	2
16384	2
32768	2
65536	2

5) very_large_trace:

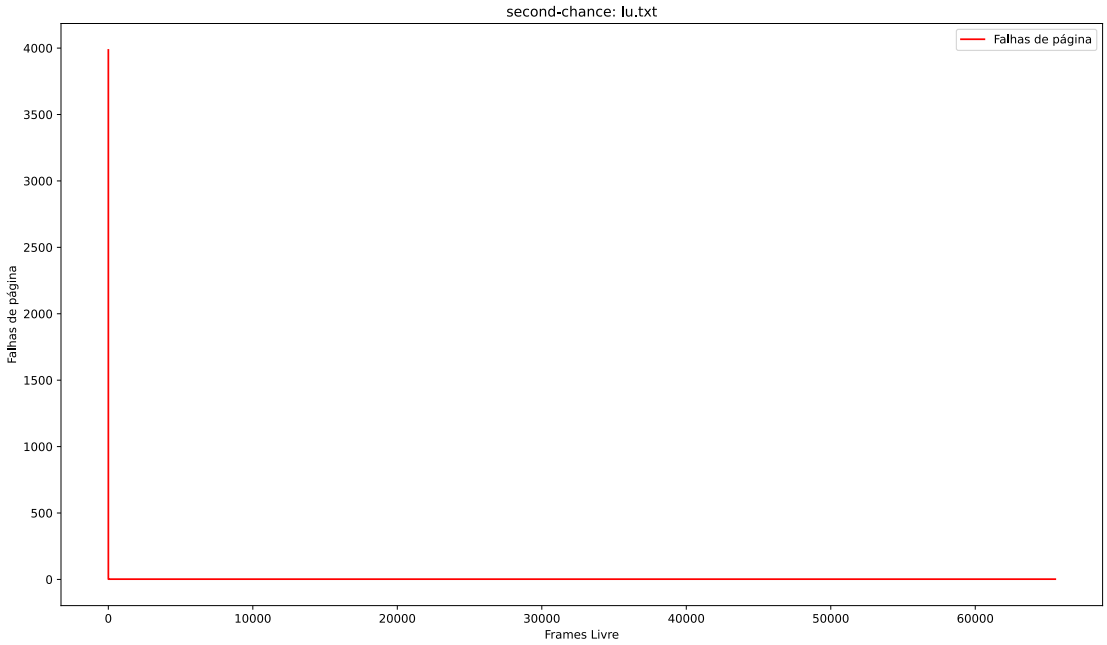


lru: very_large_trace.txt

frames livres	falhas de página
1	9955
2	9905
4	9829
8	9673
16	9342
32	8716
64	7444
128	5007
256	287
512	257
1024	257
2048	257
4096	257
8192	257
16384	257
32768	257
65536	257

C. Second chance

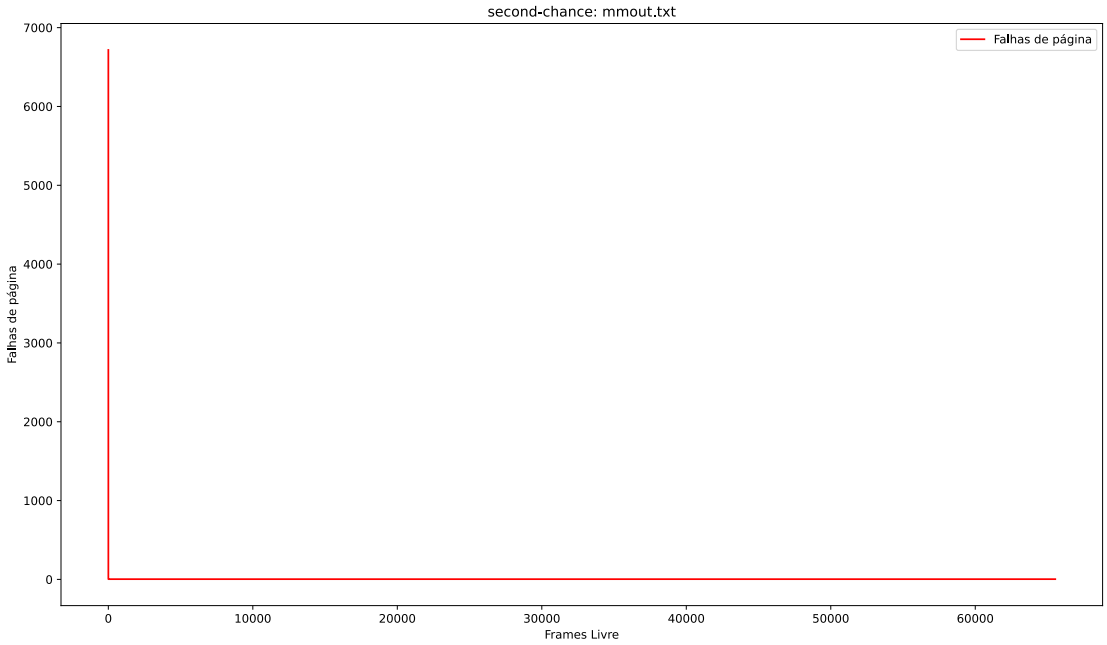
1) lu:



second-chance: lu.txt

frames livres	falhas de página
1	3986
2	2
4	2
8	2
16	2
32	2
64	2
128	2
256	2
512	2
1024	2
2048	2
4096	2
8192	2
16384	2
32768	2
65536	2

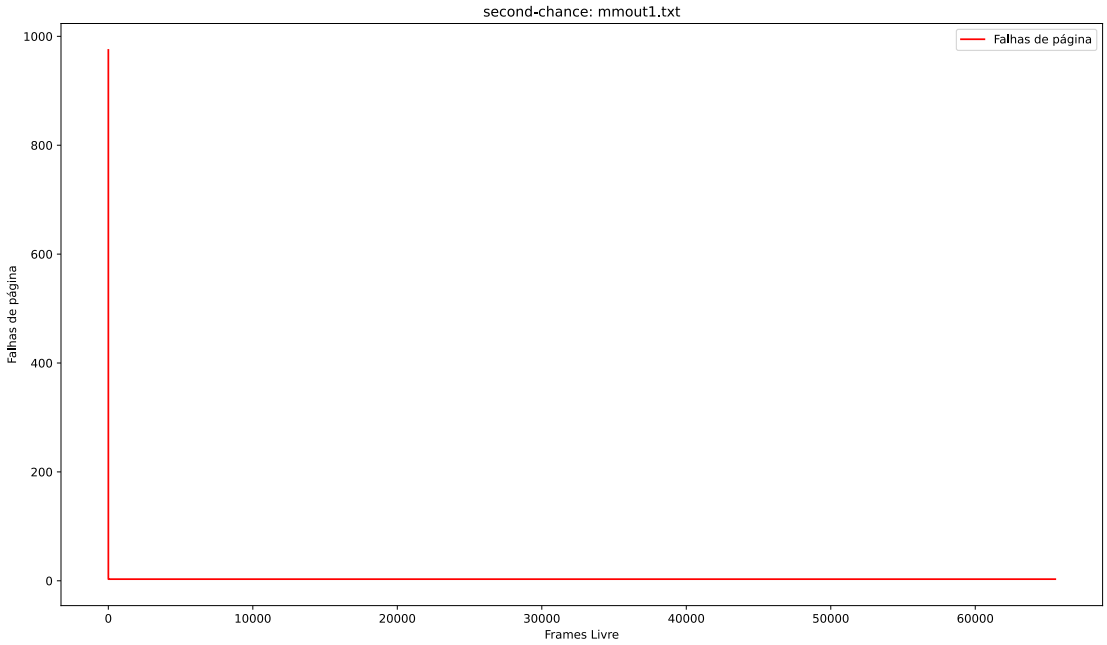
2) mmout:



second-chance: mmout.txt

frames livres	falhas de página
1	6717
2	5
4	3
8	3
16	3
32	3
64	3
128	3
256	3
512	3
1024	3
2048	3
4096	3
8192	3
16384	3
32768	3
65536	3

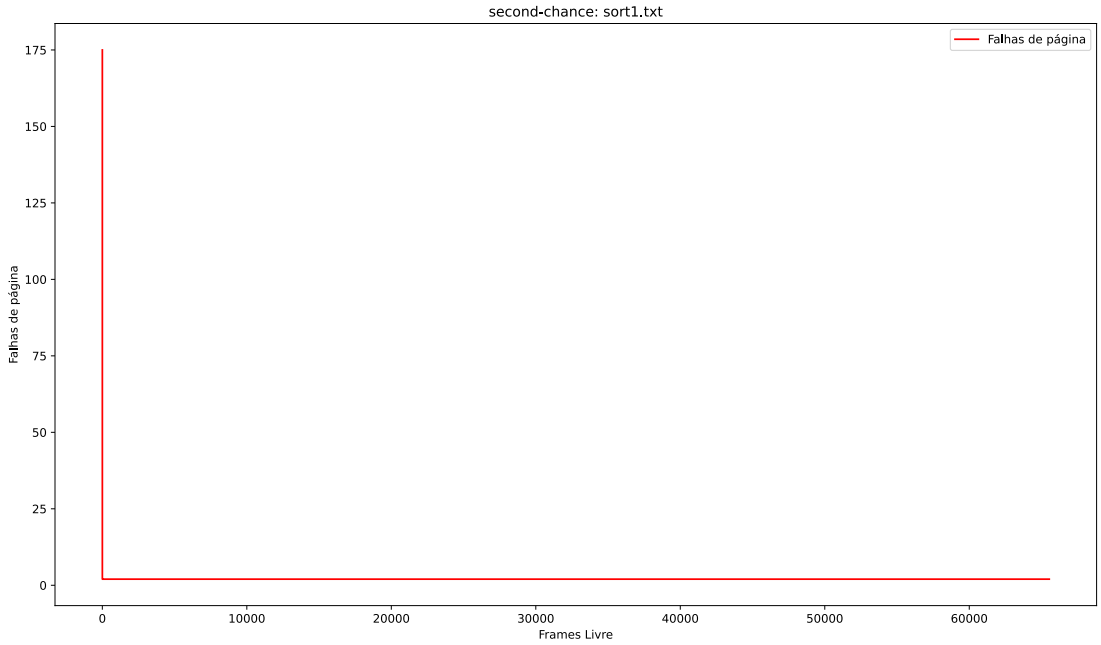
3) mmout1:



second-chance: mmout1.txt

frames livres	falhas de página
1	975
2	5
4	3
8	3
16	3
32	3
64	3
128	3
256	3
512	3
1024	3
2048	3
4096	3
8192	3
16384	3
32768	3
65536	3

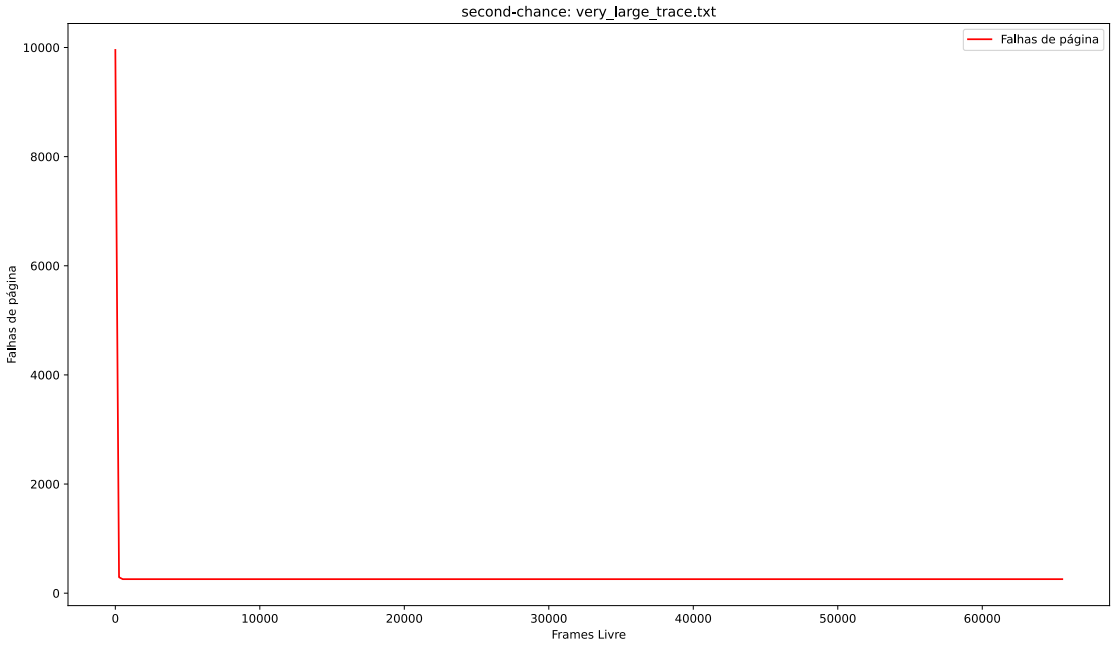
4) sort1:



second-chance: sort1.txt

frames livres	falhas de página
1	175
2	2
4	2
8	2
16	2
32	2
64	2
128	2
256	2
512	2
1024	2
2048	2
4096	2
8192	2
16384	2
32768	2
65536	2

5) very_large_trace:



second-chance: very_large_trace.txt

frames livres	falhas de página
1	9955
2	9905
4	9830
8	9678
16	9347
32	8717
64	7467
128	4972
256	292
512	257
1024	257
2048	257
4096	257
8192	257
16384	257
32768	257
65536	257

V. Discussão

Com base na observação dos arquivos de trace, nota-se que o valor de erros de página diminui drasticamente logo nos primeiros valores, se tornando um valor muito pequeno e constante no decorrer dos próximos valores de páginas livres. Isso pode ser explicado pelo fato de que muitos arquivos de trace possuem poucas páginas diferentes acessadas. Essa constatação é coerente com o princípio da localidade espacial em sistemas operacionais, que diz que quando um processo acessa uma página, é provável que ele acesse novamente uma página próxima a ela em um futuro próximo. Além disso, a maioria dos processos de um sistema operacional possui poucas páginas, o que pode contribuir para o baixo número de erros de página observado nos arquivos de trace com poucas páginas diferentes acessadas.

A técnica de paginação em sistemas operacionais é baseada em dois princípios fundamentais: localidade espacial e temporal. A localidade espacial diz respeito ao fato de que um processo tende a acessar uma página e suas páginas adjacentes em um curto intervalo de tempo. Já a localidade temporal se refere à tendência de um processo de acessar a mesma página várias vezes em um curto período de tempo. Esses princípios são utilizados pelos algoritmos de substituição de páginas, que tentam

manter as páginas mais frequentemente acessadas na memória física para reduzir o número de erros de página.

VI. Conclusões

Com base nas observações das saídas das simulações, não foram identificadas variações significativas no número de falhas de página nos arquivos de trace padrão, uma vez que a maioria dos arquivos apresentou pouca variação de páginas utilizadas, muitas vezes apenas duas. Já no trace gerado aleatoriamente, o algoritmo LRU se mostrou ligeiramente superior aos demais algoritmos testados, embora a margem de diferença tenha sido pequena.

References

[1] A. S. Tanenbaum, *Modern Operating Systems*, 4th ed. Pearson, 2015.
[2] A. Silberdchatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed. Wiley, 2018.
[3] M. Foundation, "The rust programming language," https://www.rust-lang.org, 2021.
[4] B. Team, "A page substitution algorithms simulator," https://github.com/Daniel-Boll/PAL_rs, 2021, accessed: March 27, 2023.