Gerenciamento de memória

1st Rodrigo Brickmann Rocha
UNIOESTE
Cascavel, Paraná
rodrigorocha1032@gmail.com

2nd Gabriel Alves Mazzuco
UNIOESTE
Cascavel, Paraná
gabrielalvesmazzuco@hotmail.com

3rd Gabriel Norato Claro
UNIOESTE
Cascavel, Paraná
gabrielnoratoclaro@gmail.com

***Index Terms*—Paginação, Falha de página, Substituição de página, Algoritmos de substituição, Gerenciamento de memória, Sistemas Operacionais**

I. Introdução

O *Gerenciamento de Memória* é uma parte fundamental dos Sistemas Operacionais modernos, a memória é um recurso limitado e valioso e precisamos garantir que as aplicações possam acessá-la de forma eficiente e segura.

Dentro do *Gerenciamento de Memória* temos o conceito de páginação, onde a memória é dividida em blocos fixos de tamanho chamados de páginas. Quando o processo é chamado, dentro desses blocos, ele recebe um espaço de endereçamento virtual que é mapeado para páginas físicas na memória principal.

Mas nem sempre há memória suficiente disponível para mantermos todas as páginas dos processos em execução na memória principal. Quando um processo precisa de uma página que não está na memória, ocorre uma falha de página, isso significa que o SO precisa buscar essa página no disco rígido e carrega-lo na memória principal *(RAM)*, sendo uma operação cara quando se fala de tempo e uso de recursos.

Por este intuito o SO utiliza algumas técnicas de gerenciamento para minimizar as falhas de páginas e otimizar o uso dos seus recursos. Dentro destas técnicas temos algoritmos de subsituição de páginas que nada mais são que formas do SO determinar qual página dever ser substituída quando a memória está cheia, e o pré-carregamento de páginas, que tenta antecipar quais páginas serão necessárias num futuro próximo.

II. Descrição do problema

A. Objetivo do trabalho

O Gerenciamento de memória é uma das funções mais importantes dentro de um SO, pois é responsável por alocar e liberar recursos de memória para nosso processos. Levando em conta este problema, o objetivo deste trabalho é determinar o número de falha de páginas em dois algoritmos sendo eles o LRU e o LRU aproximado.

B. Testes no algoritmo

Para que seja possível realizar testes dentro da implementação, são necessários arquivos em formato hexadecimal, sendo divididos em número da página e deslocamento, conforme o tamanho da página. É necessário contruir uma string de referência para que se possa determinar a página acessada e assim avaliar o comportamento do algoritmo de substituição considerando diferentes números de frames livres e páginas de tamanho 4096 bytes.

C. Os algoritmos que serão simulados

Os algoritmos de substituição de páginas que serão simulados nesta análise são o LRU (Least Recently Used) e o LRU aproximado, cada um com suas próprias técnicas de gerenciamento de memória. A escolha do algoritmo mais adequado dependerá das necessidades e requisitos específicos do sistema em questão, tendo em vista que eles apresentam variações significativas em suas abordagens. A compreensão detalhada de cada algoritmo, seus benefícios e limitações, é crucial para se obter resultados precisos e relevantes nesta simulação.

1) O LRU puro: O *LRU puro* é um algoritmo que trabalha com a ideia de que as páginas que foram menos recentemente acessadas, devem ser as primeiras a serem substituidas. O LRU puro é implementado mantendo uma lista ordenada das páginas na memória, em que a página menos recentemente usada é mantida no início da lista. Sempre que uma página é acessada, ela é movida para o final da lista e quando uma nova página precisa ser inserida na memória e não há espaço, a página no ínicio da lista (ou a menos recentemente usada) é substituida.

2) O LRU aproximado: O *LRU aproximado* é uma variação do algoritmo LRU que utiliza uma abordagem menos custosa em termos de recursos. Em vez de manter uma lista ordenada de todas as páginas na memória, o LRU aproximado utiliza uma estrutura de dados mais simples como uma pilha ou uma fila, para manter um conjunto de referências recentes. Aqui podemos por exemplo utilizar *bit referência, bits adicionais, segunda chance, segunda chance melhorado* e entre outros.

III. Descrição das Implementações

A. Arquivos separados que são utilizados no LRU e no LRU aproximado

No início das duas implementações, serão importados dois arquivos: *leitura.py* e *funcoes.py*. Esses arquivos contêm funções vitais para o funcionamento dos dois algoritmos. Como os dois algoritmos são semelhantes, suas primeiras funções são idênticas. Tambem é importado a biblioteca time para sabermos o tempo de execução em certas partes do

código, como a leitura do arquivo e a implementação de fato da LRU/LRU aproximada.

```
import leitura as L
import funcoes as F
import time
```

1) leitura.py: O arquivo *leitura.py* contém duas funções. A primeira sendo chamada de *defineleitura()*, permitindo que o usuário selecione um arquivo de .txt ou .trace que será lido posteriormente no algoritmo. Já na segunda função, chamada de *HexBin()*, há a conversão de caracteres hexadecimais em sequências binárias de 4 bits. Está função lê cada linha do arquivo e itera sobre cada caractere fazendo assim a representação binária.

```
def define_leitura():
def Hex_Bin(dados, arquivo):
```

2) funcoes.py: O arquivo *funcoes.py* contém duas funções também, a primeira sendo chamada de *clearterminal()* sendo responsável por limpar o terminal, apenas utilizada por bens estéticos e de organização. Já a segunda sendo chamada de *B2Dpagina()* recebe uma lista de valores binários que representam as páginas da memória. Está função converte cada valor binário para seu equivalente decimal utilizando a fórmula 2^n. O resultado será o *somatório* que será adicionado a lista *paginador*.

```
def clear_terminal():
def B2D_pagina(dados):
```

B. Leitura e determinação dos frames

Dentro dos dois algoritmos, o usuário é solicitado para informar qual será o arquivo que será escolhido para a leitura, logo após ele é aberto em modo de leitura. Dependendo do arquivo, poderá demorar um pouco mais de tempo para a execução. Tambem é convocado novamento o user para determinar a quantidade de frames que vão estar disponíveis para o algoritmo utilizar, caso o valor informado seja zero ou menor, o programa se encerra.

```
arquivo = open("traces/" + texto, "r")
qtd_frames = int(input("..."))
```

C. Criação da lista "paginador"

A partir da lista *dados*, é criada uma lista *paginador* que representa as páginas que serão gereciadas pelo algoritmo. A política LRU é implementada em um loop que percorre cada página *i* na lista *paginador*.

```
paginador = F.B2D_pagina(dados)
```

D. Diferenças da implementação da LRU pura e da LRU aproximada

1) LRU pura: Dentro da iteração do paginador, primeiro é testado se o conjunto *set-aux* ainda não está cheio (ou seja, tiver menos elementos que a quantidade de frames), o algoritmo verifica se a página *i* já está no conjunto *set-aux*, caso sim ele é movida para o topo da pilha stack e contabiliza um acerto, caso não, a página *i* é adicionada ao *set-aux* e ao topo da pilha, assim contabilizado uma falha de página.

```
if len(set_aux) < qtd_frames:
```

Caso o conjunto *set-aux* estiver cheio, o algoritmo verificará se a página *i* está presente no conjunto, se sim a página é movida para o topo da pilha contabilizando um acerto, mas caso isso seja um não, a página menos recente é removida tanto do conjunto *set-aux* quanto da pilha *stack* e a página *i* é adicionada ao conjnuto e ao topo da pilha e assim contabilizando uma falha.

2) LRU aproximada: Da mesma forma que a LRU pura, o paginador se inicia testando a quantidade de frames disponíveis é menor do que o número de páginas que precisam ser acessadas. Se isso for verdadeiro, o algoritmo adiciona as páginas à lista de frames até que ela esteja cheia gerando falhas e acertos.

Caso a quantidade de frames diponíveis for maior ou igual ao número de páginas, o algoritmo verifica se a página já está presente na lista de frames disponíveis, se a página estiver presente o bit de referência é atualizado e a página movida para o topo da pilha, mas caso ela não esteja presente, o algoritmo remove a página na base da pilha (que foi usada há mais tempo) e adiciona a nova página ao topo da pilha atualizando a lista de frames disponíveis.

```
if bitRef[indice] == False:
    bitRef[indice] = True
```

No caso em que a lista de frames está cheia, o algoritmo move a página da base da pilha enquanto o bit de referência estiver definido como verdadeiro. Quando encontrar uma página definida como falsa, essa página é removida e a nova página é adicionada ao topo da pilha e à lista de frames sendo novamente atualizada.

Caso todas as páginas estejam com bit de referência definidos como verdadeiro, a página na base da pilha é removida, o bit de referência é definido como falso e a nova página é adicionada ao topo da pilha.

E. Estatísticas da execução

No final do loop paginador, os dois programas imprimem as estastísticas da sua execução com o arquivo que foi lido, a quantidade de frames que foi escolhido, o tempo de execução da leitura do arquivo, o tempo de execução do paginador, os acerto e as falhas de páginas.

1) Cálculo de tempo: Em dois momentos no algoritmo é utilizado uma flag que conta o tempo de execução, uma sendo na leitura do arquivo e outra na execução do paginador. Os segundos são determinados pela fórmula $fim_{tempo} - inicio_{tempo} = tempo_{segundos}$ sendo arredondados no print para duas casas decimais.

F. Resultado do Algoritmo

Usando a implementação do algoritmo de substituição de página LRU Pura e LRU Aproximada, conseguimos alguns resultados interessantes. O algoritmo de substituição de página LRU é utilizado para otimizar a memória em sistemas operacionais. Ele funciona de forma a remover da memória as páginas que foram menos utilizadas recentemente, dando prioridade para aquelas que foram mais utilizadas.

Para testar esse algoritmo, foram selecionados diferentes arquivos de tamanhos e formatos variados. Os arquivos foram construídos para testar as políticas de substituição de página LRU Puro e LRU Aproximado.

TABLE I
TABELA DE ACERTO (LRU_{puro})

	2	4	8	16	32
bigone.trace	2487776	3033865	3336252	3599409	3797467
bzip.trace	845571	907230	969309	996656	997867
gcc.trace	596045	756191	828814	883396	915599
sixpack.trace	516839	717380	823504	891318	932253
swim.trace	529321	653064	714625	828039	951746
lu.txt	41830	41830	41830	41830	41830
mmout.txt	67231	67232	67232	67232	67232
mmout1.txt	9071	9072	9072	9072	9072
sort1.txt	522	522	522	522	522

TABLE II
TABELA DE ACERTOS ($\text{LRU}_{aproximado}$)

	2	4	8	16	32
bigone.trace	2463803	3023890	3343905	3612778	3798069
bzip.trace	825984	900924	969187	996555	997861
gcc.trace	597544	754333	830368	883409	915382
sixpack.trace	513089	719542	825411	891955	932329
swim.trace	527186	649092	718939	840856	952496
lu.txt	41830	41830	41830	41830	41830
mmout.txt	67229	67232	67232	67232	67232
mmout1.txt	9069	9072	9072	9072	9072
sort1.txt	522	522	522	522	522

Observa-se que o número de acertos aumentou à medida que o tamanho da memória cache aumentou. Isso indica que, quanto maior a memória, maior é a probabilidade de as páginas serem armazenadas na memória e, consequentemente, maior é o número de acertos.

TABLE III
TABELA DE FALHAS (LRU_{puro})

	2	4	8	16	32
bigone.trace	1512224	966135	663748	400591	202533
bzip.trace	154429	92770	30691	3344	2133
gcc.trace	403955	243809	171186	116604	84401
sixpack.trace	483161	282620	176496	108682	67747
swim.trace	470679	346936	285375	171961	48254
lu.txt	2	2	2	2	2
mmout.txt	4	3	3	3	3
mmout1.txt	4	3	3	3	3
sort1.txt	2	2	2	2	2

TABLE IV
TABELA DE FALHAS ($\text{LRU}_{aproximado}$)

	2	4	8	16	32
bigone.trace	1536197	976110	656095	387222	201931
bzip.trace	174016	99076	30813	3445	2139
gcc.trace	402456	245667	169632	116591	84618
sixpack.trace	486911	280458	174589	108045	67671
swim.trace	472814	350908	281061	159144	47504
lu.txt	2	2	2	2	2
mmout.txt	6	3	3	3	3
mmout1.txt	6	3	3	3	3
sort1.txt	2	2	2	2	2

O número de falhas, apresentado nas tabelas, foi relativamente baixo. Isso indica que a política de substituição de página LRU Pura e LRU Aproximada são eficientes em minimizar o número de falhas.

A política de substituição de página LRU Pura e LRU Aproximada foram eficazes para a maioria dos arquivos testados, já que os resultados das tabelas de acertos foram relativamente altos em ambas as políticas. Em geral, os resultados obtidos para ambas as políticas de substituição de página foram muito semelhantes. No entanto, em alguns casos, como para o arquivo bigone.trace, a política LRU Aproximada apresentou um resultado ligeiramente melhor do que a LRU Pura.

Por fim, para os arquivos de tamanho menor, como lu.txt, mmout.txt, mmout1.txt e sort1.txt, o número de acertos e falhas foi o mesmo em todas as configurações de tamanho da memória cache testadas. Isso pode ser explicado pelo fato de que, para arquivos menores, a probabilidade de acessar as mesmas páginas várias vezes é maior.

REFERENCES

[1] Roman, N. T.; Morandini, M.; Ueyama, J. (s.d.). Gerência de Memória. Acesso em: 19 de março de 2023, de http://wiki.icmc.usp.br/images/d/dc/Aula12.pdf.

[2] XLMS2065. Aproximação LRU: Algoritmo de Segunda Chance. Disponível em: https://acervolima.com/aproximacao-lru-algoritmo-de-segunda-chance/. Acesso em: 18 de março de 2023.

[3] Gerenciamento de Memória. In: Fundamentos de Sistemas Computacionais. Departamento de Computação e Estatística, Universidade Estadual Paulista, São José do Rio Preto, SP. Disponível em: https://www.dcce.ibilce.unesp.br/ aleardo/cursos/fsc/cap12.php. Acesso em: 19 de março de 2023.