UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE CIÊNCIA DA COMPUTAÇÃO

LIVIA BARBOSA FONSECA - 118039721
LUIZ RODRIGO LACÉ RODRIGUES - 118049873

SIMULAÇÃO DE GERENCIADOR DE MEMÓRIA VIRTUAL

RIO DE JANEIRO
2022

INTRODUÇÃO

O Sistema Operacional utiliza o gerenciador de memória virtual para coordenar e gerenciar as memórias. Esse gerenciador é responsável por alocar espaço em memória aos processos que serão executados e liberar as posições de memória ocupadas quando os processos são finalizados. Além disso, ele também controla quais partes da memória estão sendo utilizadas e quais não estão.

Uma das técnicas utilizadas na organização lógica da memória é a paginação. Nesse tipo de método, a memória física é dividida em frames e uma tarefa é dividida em páginas ou segmentos. Os frames da memória física, correspondem a páginas de memória virtual e cada página é mapeada num frame de memória através da paginação. Já para a substituição de páginas, existem diversas políticas. Uma delas é a substituição de páginas Least Recently Used (LRU), que, na necessidade de retirar uma página de disco da memória, dá prioridade àquelas que não foram acessadas por um maior período de tempo.

Para esse trabalho, desenvolvemos, na linguagem C, um simulador gerenciador de memória virtual através da substituição de páginas LRU.

PREMISSAS DEFINIDAS PARA O DESENVOLVIMENTO DO SIMULADOR

As premissas definidas para o desenvolvimento do simulador foram:

- **Tamanho da memória primária**
  A memória primária é um vetor de 64 posições, esse vetor será do tipo “Pagina”, estrutura mostrada na Figura 1.

- **Memória secundária**
  A memória secundária foi tratada como um estado em que a página se encontrava quando saia da memória principal.

- **Alocando páginas na memória primária quando ela se encontra vazia**
  Quando a memória principal ainda está vazia e determinado processo ainda não estourou seu working set limit, as páginas são encaixadas em sequência na memória principal, a página atual entra na posição “anterior+1” da memória.

- **PID de cada processo**
  O valor do PID atual será o valor do processo anterior +1.

- **Quantidade de processos**
  20 processos criados. (0 à 19)

- **Criação dos processos**
  Um processo é criado a cada unidade de tempo. Uma unidade de tempo possui 3s.

- **Quantidade de páginas**
  Cada processo possui 50 páginas (0 à 49).

- **Informações sobre páginas**
  No algoritmo, cada página é instanciada com o auxílio de uma estrutura que possui os seguintes atributos (Figura 1):
    - ID do processo;
    - ID da página;
    - Contador de tempo que a página está na memória primária.

```
// Estrutura de paginas para nas memórias
typedef struct{
    int idProcesso;
    int idPagina;
    int tempoNaMemoria;
}Pagina;
```

Figura 1 - estrutura das páginas

- **Tabela de páginas**
  A tabela de páginas do algoritmo criado possui os seguintes atributos (Figura 2):
    - Posição na memória, caso o processo esteja na memória principal;
    - Inteiro, que vai ser utilizado como booleano, indicando se o processo está na memória primária ou não;
    - Inteiro, que vai ser utilizado como booleano, indicando se o processo está na memória secundária ou não.

  Obs: os atributos não foram escolhidos como booleano visto que, para tratar erros do código, instanciamos esses atributos com -1

```
// Estrutura para a tabela de processos/paginas
typedef struct{
    int posMP;
    int isInMP;
    int isInMS;
}Tabela;
```

Figura 2 - estrutura da tabela de páginas

- **Solicitação de alocação na memória primária**
  Cada processo criado solicita a alocação de uma página aleatória na memória a cada unidade de tempo. Uma unidade de tempo possui 3s.

- **Working set limit**
  No algoritmo, o working set é um vetor que guarda a quantidade de processos que se encontram na memória principal. O índice do vetor é o PID do processo. Cada processo só pode possuir até 4 páginas na memória principal.

- **Quando é realizado a substituição das páginas do mesmo processo**
  Quando um processo pede a alocação de uma página na memória principal, porém, esse processo já possui 4 páginas na memória principal.

- **Política adotada quando a memória primária está cheia**
  Quando a memória primária está cheia e uma página solicita alocação, podem ocorrer duas situações:
  - Quando o processo já completou seu working set limit:
    A página alocada só poderá entrar no espaço de uma das páginas do seu processo. Ela entrará no lugar da página mais antiga do processo.

  - Quando ainda não temos páginas do processo na memória ou quando ainda não completou seu working set limit:
    A página alocada poderá entrar em qualquer espaço da memória, poderá inclusive entrar no lugar de uma página do seu processo. Ela entrará no lugar da página mais antiga em memória.

ALGORITMO DESENVOLVIDO

No algoritmo desenvolvido, um processo é criado a cada unidade de tempo e todos já criados até o momento solicitam uma página aleatória no intervalo do ID da página (0-49) na memória. Dessa forma, a primeira coisa que o algoritmo realiza é a criação e a alocação de uma página aleatória do processo na memória primária, a criação desta página fica a cargo da função **criaPaginaAleatoria** (figura 3), que passado um processo como argumento, retorna uma página aleatoriamente criada (0 à 49).

Em seguida, a função **verificaPaginaNaMP** (figura 4), analisa se essa página já se encontra na memória primária, caso positivo, vamos apenas atualizar o seu tempo nela para 0 e passamos para a alocação de uma página de outro processo. Caso negativo, o próximo passo é descobrir se o working set limit do processo foi extrapolado ou não. Caso o working set do processo que está pedindo para alocar uma página no momento seja menor que 4 e a memória ainda tenha espaços não alocados, então vamos alocar a página randomizada e incrementamos o working set

do processo. Também vamos fazer o tratamento da tabela de processo/página, sinalizando que tal processo possui tal página na memória principal. Caso o working set do processo seja menor que 4, mas a memória já esteja cheia, então vamos varrer a memória primária com a função **indiceDaPaginaMaisAntigaTodas** (figura 5), que vai retornar a posição da página mais antiga da memória principal, independente do processo, e então vamos substituí-la pela página que foi recentemente randomizada. Há então um tratamento, onde a página "antiga" é sinalizada na tabela processo/pagina que agora está na memória secundária a página atual tomou o seu lugar na memória primária. Caso a página que foi alocada estava anteriormente na memória secundária, há um tratamento para sua retirada. Tratamos também de decrementar o working set do processo que teve sua pagina removida, caso a página substituída seja do processo em questão o working set não se altera.

Caso nosso working set seja igual a 4, então vamos utilizar a função **indiceDaPaginaMaisAntiga** (figura 6), para retornar o índice da memória principal onde se encontra a página mais antiga do processo em questão, assim vamos alocar a página nova em seu lugar. Vamos fazer todo o tratamento quanto à questão da memória secundária e a substituição das páginas na tabela processo/pagina. Entretanto, não precisamos nos preocupar com a alteração do working set, visto que estamos substituindo a página por outra do mesmo processo.

A cada instante de tempo, vamos então chamar a função **adicionaTempo** (figura 7). Nela vamos varrer a memória primária e adicionar uma unidade de tempo a todas as páginas que nela se encontram. É dessa forma que vamos avaliar quais paginas serão substituídas. Depois vamos printar a matriz com a função **printaMatriz** (figura 8), nela vamos printar uma matriz onde as linhas são os nossos processos(de 0 até 19) e as colunas são as páginas de cada processo (0 até 49). As páginas que são sinalizadas em verde, são as que se encontram na memória principal, e nelas podemos ver um número, esse número representa as unidades de tempo em que elas passaram na memória primária. Também imprimimos o conteúdo da memória principal para um melhor entendimento da tabela em questão.

Durante toda a execução do algoritmo há logs para acompanhar o que está ocorrendo.

O algoritmo roda até que o usuário interrompa com o comando "ctrl + c" no terminal.

```
Pagina criaPaginaAleatoria(int processo){
    Pagina pagina;
    pagina.idProcesso = processo;
    pagina.idPagina = rand() % QUANT_PAGINAS;
    pagina.tempoNaMemoria = 0;

    return pagina;
}
```

Figura 3 - Função que cria uma página aleatória

```
//metodo para verificar se uma dada pagina já se encontra na MP
bool verificaPaginaNaMP(Pagina pagina,Tabela matriz[QUANT_THREADS][QUANT_PAGINAS]){
    if(matriz[pagina.idProcesso][pagina.idPagina].isInMP == 1){
        return true;
    }
    return false;
}
```

Figura 4 - Função que verifica se o processo já está na memória primária

```
int indiceDaPaginaMaisAntigaTodas(Pagina MP[TAMANHO_MEMORIA_PRIMARIA]){
    int indiceNaMp;
    int tempo = 0;
    for(int i =0; i<TAMANHO_MEMORIA_PRIMARIA ;i++){
        //verificando qual delas possui o maior tempo
        if (MP[i].tempoNaMemoria > tempo){
            indiceNaMp = i;
            tempo = MP[i].tempoNaMemoria;
        }
    }
    return indiceNaMp;
}
```

Figura 5 - função que retorna o índice na memória primária da página mais antiga

```c
int indiceDaPaginaMaisAntiga(int processo, Tabela tabela[QUANT_THREADS][QUANT_PAGINAS], Pagina MP[TAMANHO_MEMORIA_PRIMARIA]){
    int indiceNaMp;
    int tempo = 0;
    for(int page=0; page<50; page++){
        //caso a pagina esteja na MP
        if(tabela[processo][page].isInMP){
            //verificando qual delas possui o maior tempo
            if (MP[tabela[processo][page].posMP].tempoNaMemoria > tempo){
                indiceNaMp = tabela[processo][page].posMP;
                tempo = MP[tabela[processo][page].posMP].tempoNaMemoria;
            }
        }
    }
    return indiceNaMp;
}
```

Figura 6 - Função que retorna o índice da página mais antiga de determinado processo

```c
// varrendo o vetor da MP e adicionando uma unidade de tempo em todas as paginas
void adicionaTempo(int pos, Pagina *MP){
    for (int i = 0; i<pos;i++ ){
        MP[i].tempoNaMemoria += 1;
    }
}
```

Figura 7 - Função que realiza o acréscimo de uma ut para as páginas na memória primária

```c
// imprimindo o conteúdo da matriz (tempo na MP de cada pagina)
void printaMatriz (Tabela matriz[QUANT_THREADS][QUANT_PAGINAS],Pagina MP[TAMANHO_MEMORIA_PRIMARIA]){
    for (int i = 0; i < QUANT_THREADS; i++) { //para as linhas
        for (int j = 0; j < QUANT_PAGINAS; j++) { //para as colunas
            if(matriz[i][j].isInMP){
                printf("\033[37m\033[42m%d\033[0m ", MP[matriz[i][j].posMP].tempoNaMemoria);
            }
            else{
                printf("%d ",matriz[i][j].isInMP);
            }
        }
        printf("\n");
    }
}
```

Figura 8 - Função que realiza o acréscimo de uma ut para as páginas na memória primária

```c
//metodo para verificar se uma dada pagina já se encontra na MS
bool verificaPaginaNaMS(Pagina pagina,Tabela matriz[QUANT_THREADS][QUANT_PAGINAS]){
    if(matriz[pagina.idProcesso][pagina.idPagina].isInMS == 1){
        return true;
    }
    return false;
}
```

Figura 9 - Função que retorna se um processo se encontra na memória secundária

EXECUÇÃO DO CÓDIGO

Primeiro precisamos compilar o arquivo com o código da linguagem em C.
Para isso usamos o comando no terminal para compilar o código com gcc:

gcc memoria.c -o memoria - Wall

Com o código compilado, agora temos o executavel “memoria”, podemos executar o código com o comando

./memoria

SAÍDA

A saída do simulador é constituída de basicamente 3 partes. A primeira trata-se de um “trace” do que aconteceu em determinada unidade de tempo. Nela podemos acompanhar quando um processo é criado e o que acontece na memória primária quando um determinado processo pede para alocar uma página. Temos também avisos de quando a memória primária está cheia e quando uma página sai da memória secundária e volta para a memória principal e vice-versa.

No segundo trecho da saída, temos o conteúdo do vetor que representa a nossa memória primária, onde temos o número da página, seu processo e há quantas unidades de tempo ela se encontra nela. Esse trecho da saída serve como uma legenda da tabela que vem a seguir.

O último trecho do código é um print da tabela de processos/páginas. Nela temos as linhas que representam os processos (0 até 19) e as colunas representam as páginas de determinado processo (0 até 49). Quando uma página está sinalizada em verde, quer dizer que ela se encontra na memória primária e o número que se altera nela representa há quantas unidades de tempo ela se encontra nela. As páginas que estão em verde mas com o número 0 são as que acabaram de entrar na memória.

Agora de fato executando o código temos o seguinte exemplo:

Figura 10

Podemos observar que temos a criação do processo 0 e que ele pediu para alocar a página 33 na memória primária, como ela estava vazia, entrou na primeira posição do vetor. Em seguida temos o conteúdo da memória. Depois temos o print da nossa tabela, podemos observar que na posição 33 do processo 0 a página está marcada em verde, mas com numéro 0, isso quer dizer que ela acabou de entrar na memória.

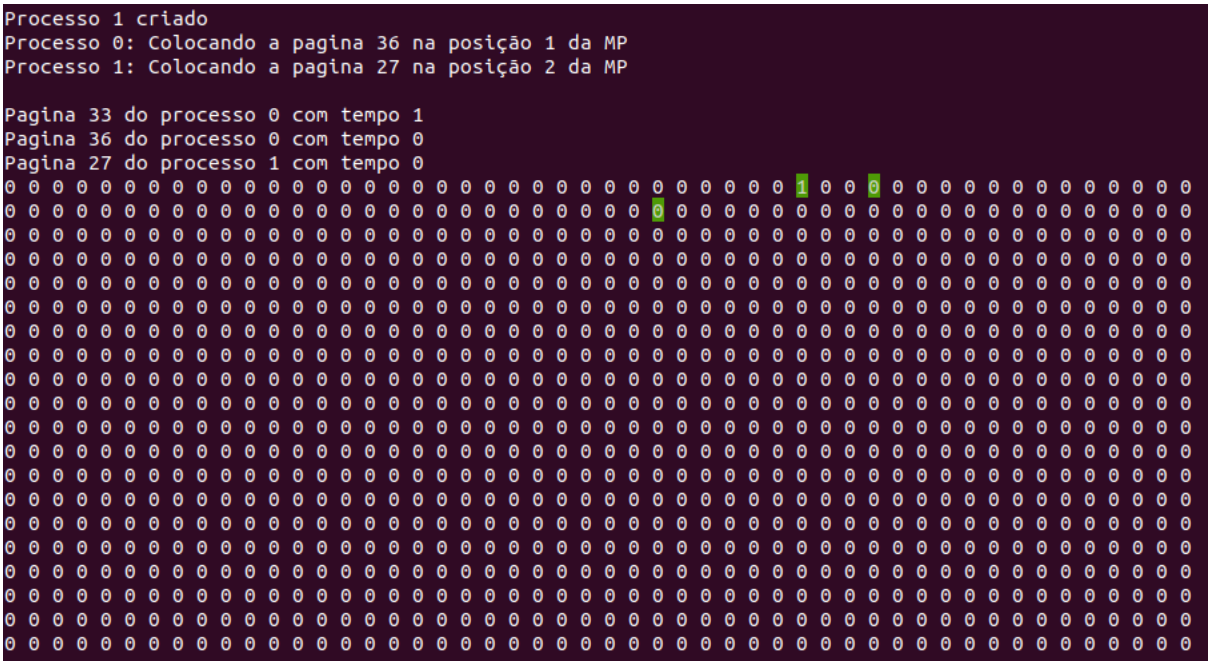

Figura 11

Uma unidade de tempo depois, nosso processo 1 vai ser criado. Como no nosso simulador, todo processo criado pede para alocar uma página na memória por unidade de tempo, então agora o processo 0 e o processo 1 vão pedir para

alocar uma página cada um. Como uma unidade de tempo se passou desde a **figura anterior (Figura 10)**, agora temos que a página anteriormente alocada (33 do processo 0) agora possui seu valor na tabela igual a 1, ou seja, está há 1 unidade de tempo na memória. As que foram alocadas nessa unidade de tempo possuem seu valor igual a 0.

Seguimos com a execução do simulador.

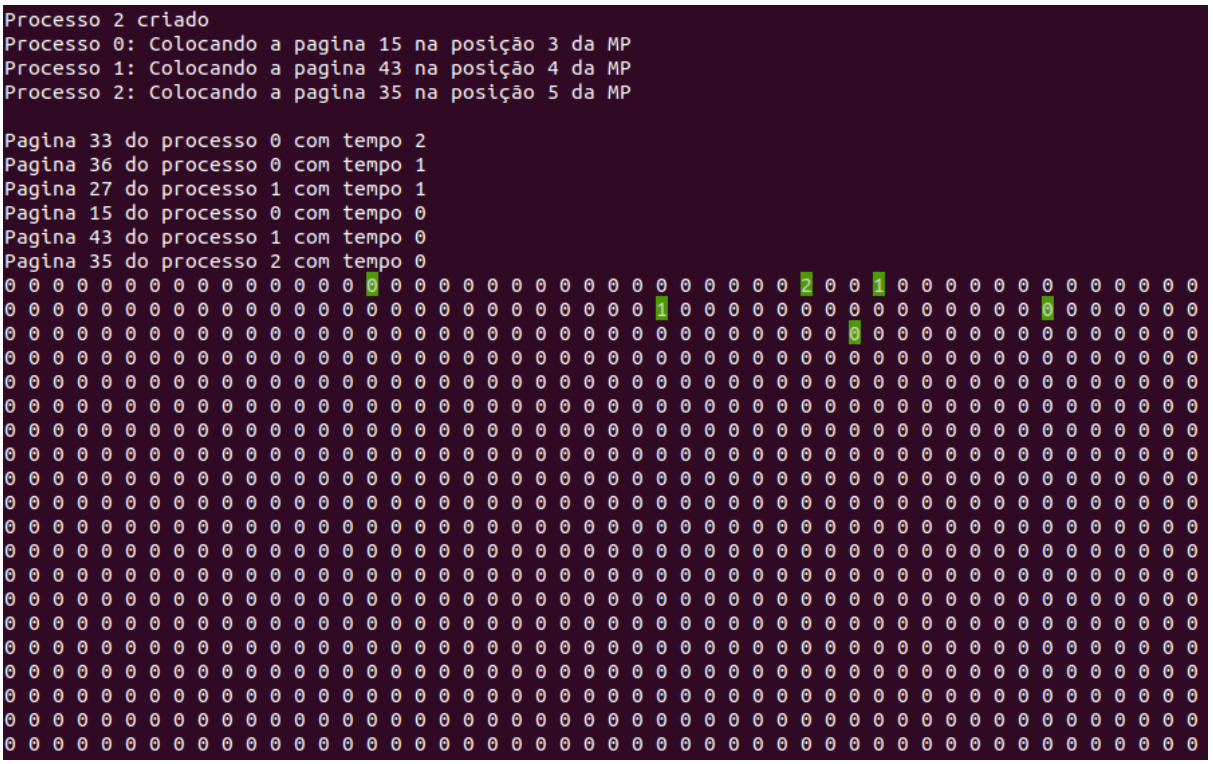

Figura 12

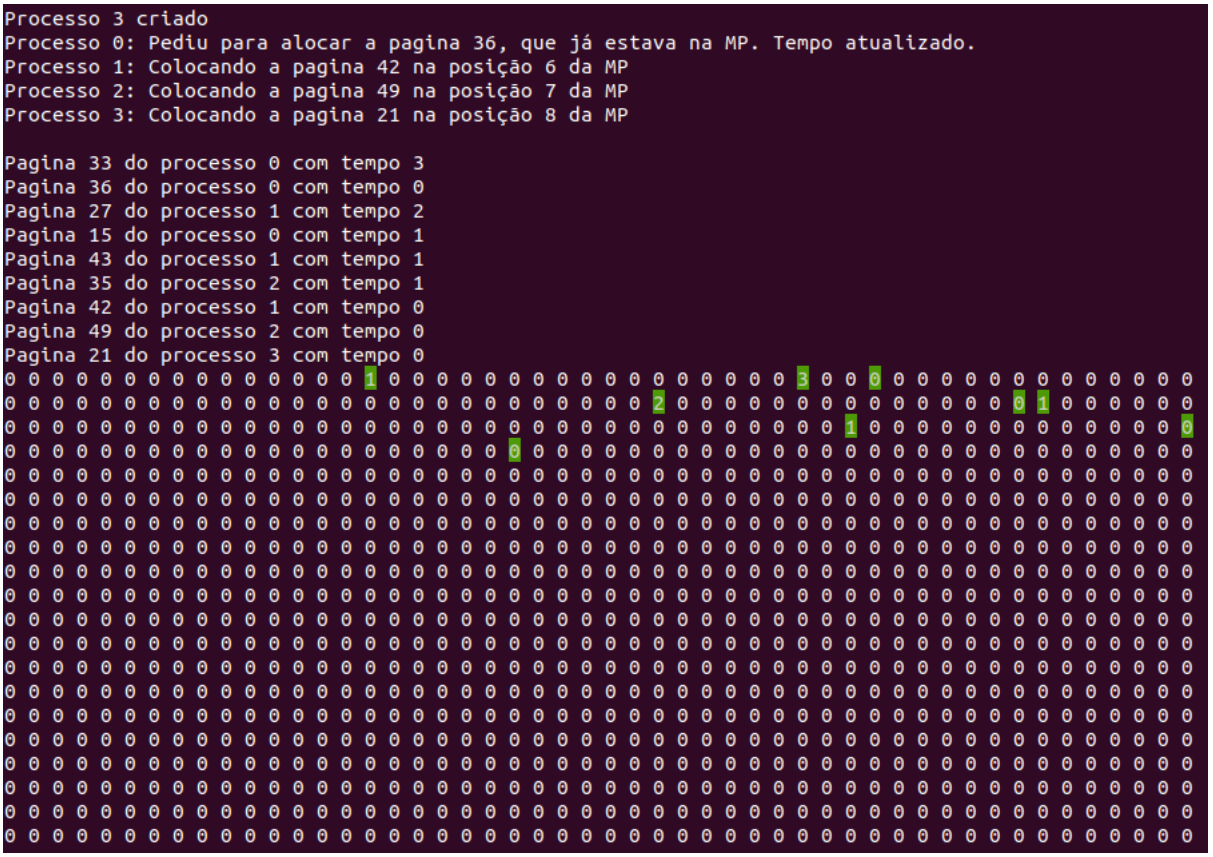

Figura 13

Podemos ver que nesse momento o processo 0 pediu para a alocar a página 36 que já estava na memória há 1 unidade de tempo. Nós então atualizamos o seu tempo na memória para 0.

Seguimos com a execução do código.

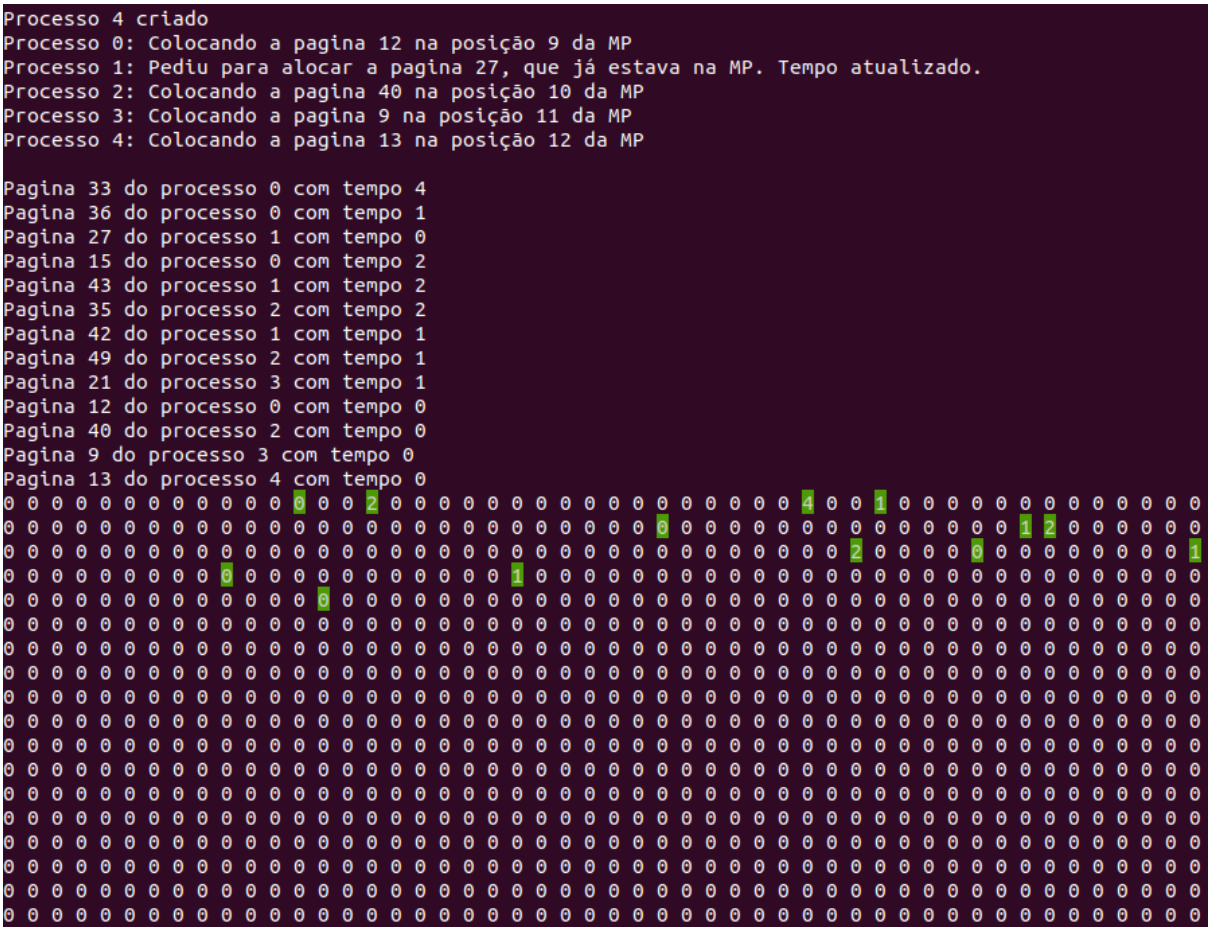
Processo 4 criado
Processo 0: Colocando a pagina 12 na posição 9 da MP
Processo 1: Pediu para alocar a pagina 27, que já estava na MP. Tempo atualizado.
Processo 2: Colocando a pagina 40 na posição 10 da MP
Processo 3: Colocando a pagina 9 na posição 11 da MP
Processo 4: Colocando a pagina 13 na posição 12 da MP
Pagina 33 do processo 0 com tempo 4
Pagina 36 do processo 0 com tempo 1
Pagina 27 do processo 1 com tempo 0
Pagina 15 do processo 0 com tempo 2
Pagina 43 do processo 1 com tempo 2
Pagina 35 do processo 2 com tempo 2
Pagina 42 do processo 1 com tempo 1
Pagina 49 do processo 2 com tempo 1
Pagina 21 do processo 3 com tempo 1
Pagina 12 do processo 0 com tempo 0
Pagina 40 do processo 2 com tempo 0
Pagina 9 do processo 3 com tempo 0
Pagina 13 do processo 4 com tempo 0

Figura 14

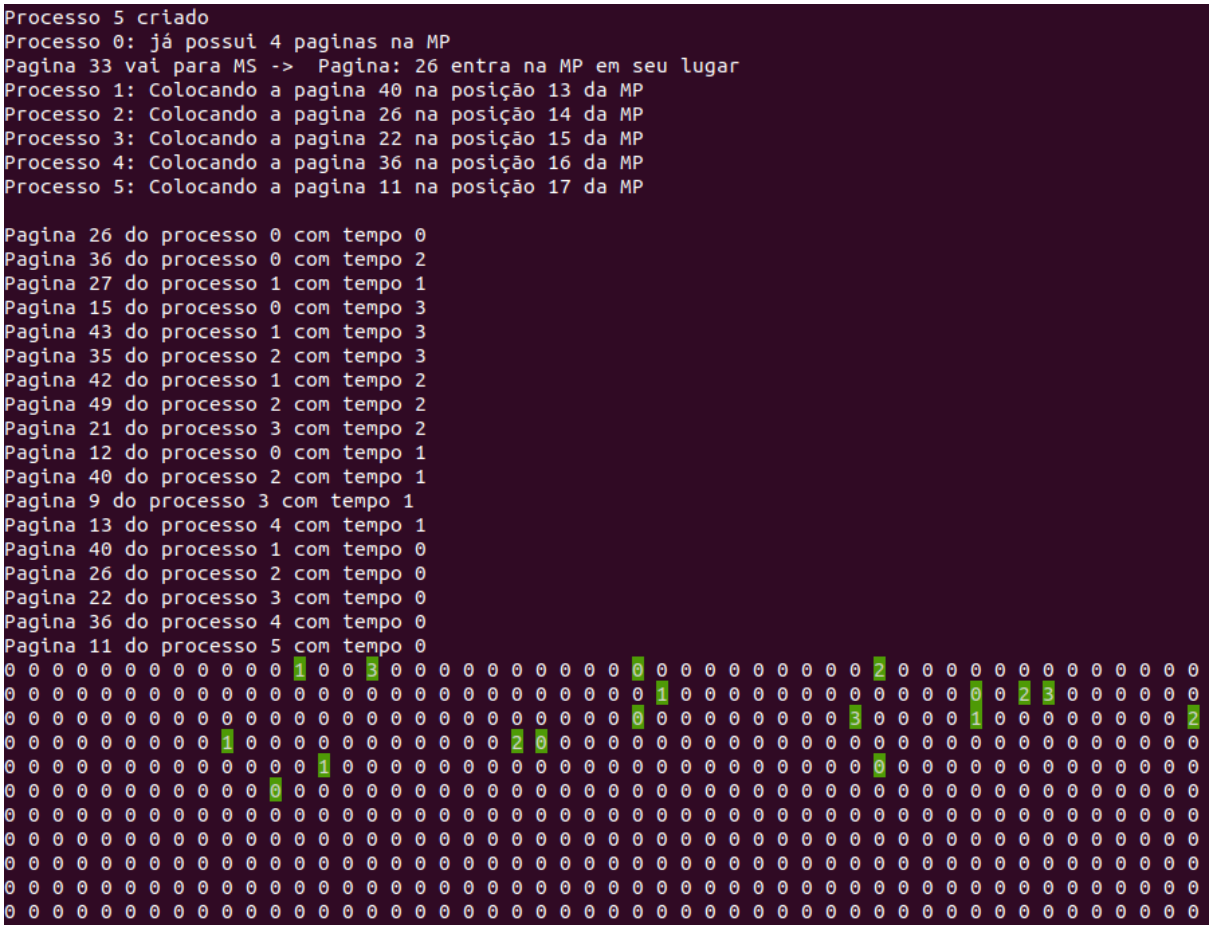

Figura 15

Nesse momento o processo 0 já possui 4 páginas na memória principal, logo para alocar uma nova páginas é preciso que a mais antiga saia para a memória secundária para que a nova ocupe o seu lugar com tempo igual a 0. Podemos observar na **figura anterior (Figura 14)** que a página 33 do processo 0 estava há 4 unidades de tempo na memória, enquanto as outras páginas desse processo estavam com 1,2 e 0 unidades de tempo na memória. A página 33 do processo 0 agora se encontra na memória secundária.

Avançando um pouco agora no tempo:

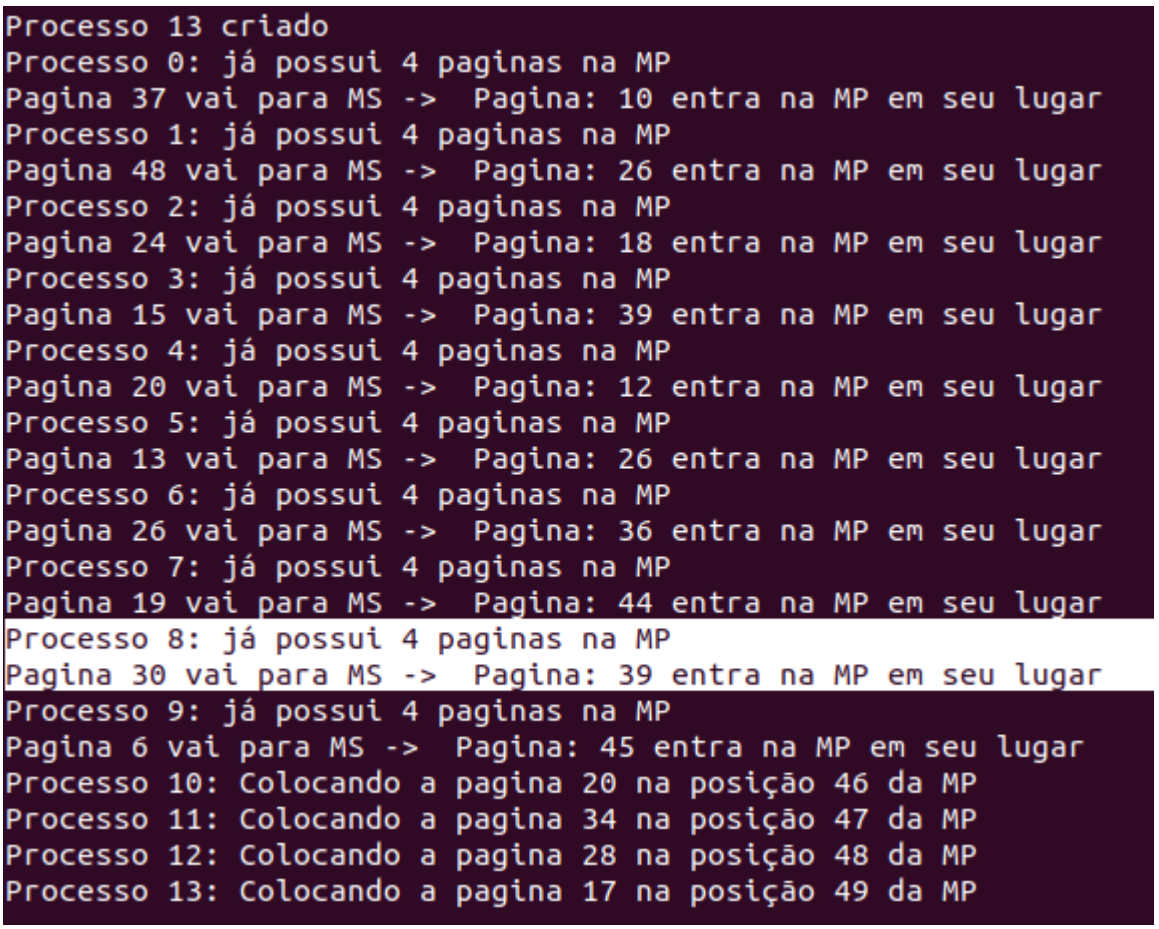

Figura 16

Em algum momento o Processo 8 já possuía 4 páginas na memória principal e pediu para alocar a página 39, como a página 30 era a mais antiga, ela foi para a memória secundária.

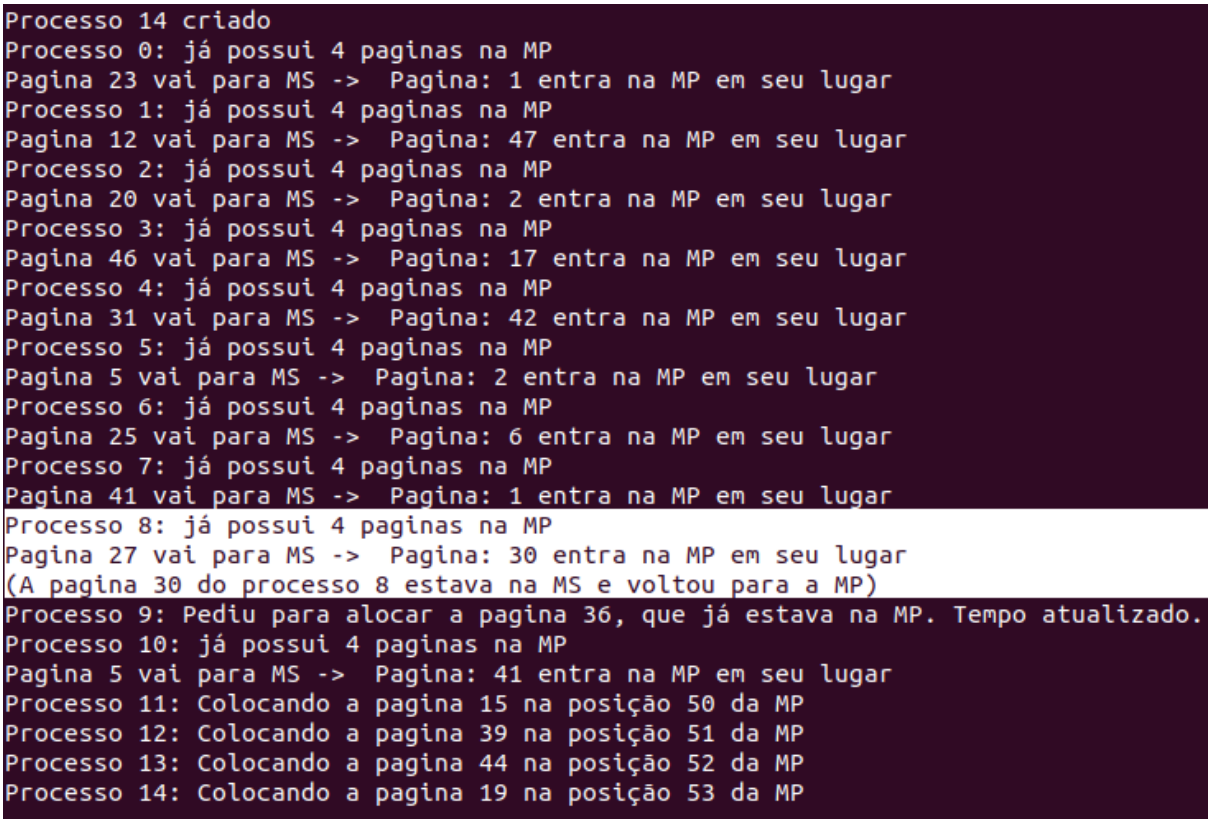

Figura 17

Vimos na **figura anterior (Figura 16)** que a página 30 do processo 8 estava na memória secundária. Nesse momento o processo 8 tenta alocar novamente a página 30, como ela estava na memória secundária, haverá um aviso que agora ela voltou para a memória principal e tomou o lugar da página mais antiga desse processo.

Avançando mais no tempo:

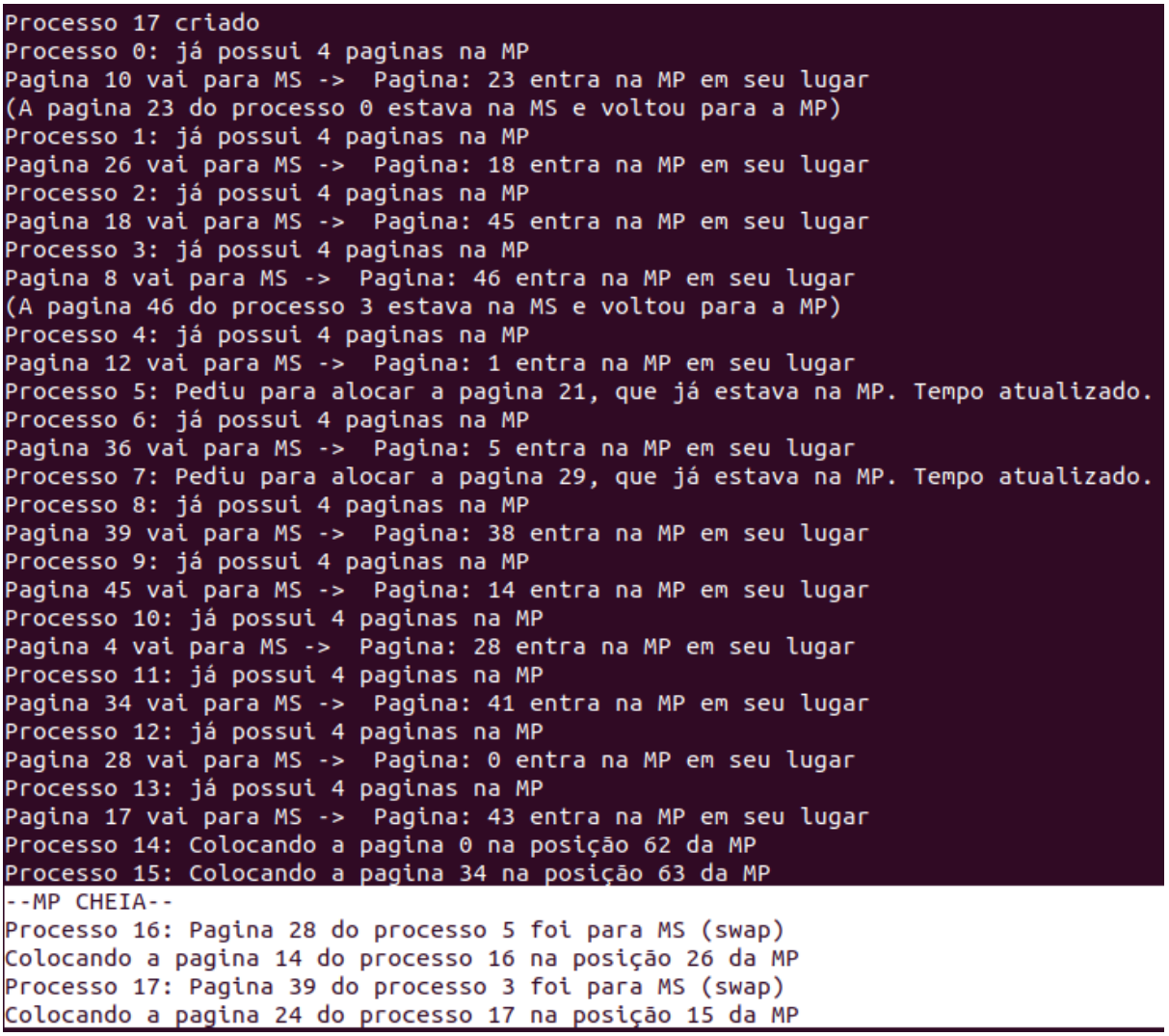

Figura 18

Nesse momento a memória primária vai lotar, ou seja, vamos ter 64 páginas nela. Para que o processo 16 e 17 possam alocar suas páginas, é preciso que a mais antiga seja removida. Para que o processo 16 pudesse alocar a página 14, a página 28 do processo 5 foi para a memória secundária. O mesmo acontece com o processo 17 que acabou de ser criado, como seu working set é menor que 4, o simulador vai transferir outra página mais antiga (39 do processo 3) para a memória secundária para que a 15 do processo 17 seja alocada em seu lugar.

CONHECIMENTOS ADQUIRIDOS NA PRODUÇÃO

O desenvolvimento deste simulador nos ajudou a visualizar como funciona o algoritmo de gerência de memória LRU. Além disso, nos fez refletir sobre a importância da fragmentação da memória principal e a paginação dos processos. Outro ponto interessante foi estudar sobre o swapping e ver essa técnica sendo executada, pudemos observar um pouco como esse método reverte o problema de

insuficiência de memória durante a execução de alguns processos multiprogramados.

ACESSO AO REPOSITÓRIO

Acesse o link a seguir para encontrar o simulador criado:

https://github.com/luizrodrigolace/Gerenciador-de-Memoria---LRU

REFERÊNCIAS

BASTOS, Valéria. Aulas da disciplina de Sistemas Operacionais.
WILLIAN, Stallings. Operating Systems: Internals and Design Principles. 4 ed.