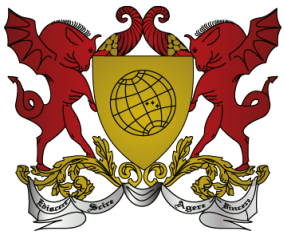

**UNIVERSIDADE FEDERAL DE VIÇOSA**

***Campus* Florestal**

Curso superior de Ciência da Computação

CCF 451 - Sistemas Operacionais

Prof. Daniel Mendes Barbosa

# DOCUMENTAÇÃO TRABALHO PRÁTICO

Bernardo Veloso Resende - 1279

Gustavo Graf de Sousa - 1283

Luis Gustavo Diniz - 1794

Marcos Assis Silva- 1804

Florestal

2017

# SUMÁRIO

# INTRODUÇÃO

O presente trabalho tem por objetivo um entendimento mais aprofundado de como o gerenciamento de memória funciona num sistema operacional. Dessa forma o trabalho foi dividido em duas tarefas (A e B), que são descritas abaixo.

A primeira tarefa do trabalho prático consiste na implementação de um sistema que simula os processos de alocação e desalocação de memória em um sistema operacional utilizando a estrutura lista encadeada. Para que tal simulação fosse implementada quatro algoritmos foram especificados, são eles: *first fit*, *next fit, best fit* e *worst fit*.

A segunda parte do trabalho, denominada tarefa B, é implementado uma simulação rudimentar da memória virtual, em que são utilizadas a estrutura de memória virtual e de moldura de páginas. Além disso é explicitado os detalhes de como os algoritmos de substituição de páginas foram implementados e o seu funcionamento.

Devido a frequente falta de memória física disponível, pois muitos programas precisam de muita memória, há a necessidade da utilização da memória virtual. Com a utilização da mesma, o tamanho total do programa pode exceder a quantidade de memória física. Com isto, o sistema operacional fica responsável em gerenciar as partes que estão ativas na memória virtual e o restante no disco.

Quando as molduras de páginas disponíveis estão todas ocupadas e há requisição de adicionar mais uma página na molduras, é preciso utilizar um algoritmo de substituição de página. Estes algoritmos irão determinar qual página na moldura irá ser substituída pela página da memória virtual. Cada algoritmo tem um funcionamento diferente. Os que foram implementados neste trabalho são: *FIFO, Segunda Chance ( Second Chance ), Relógio, Página Menos Recentemente Utilizada.*

# DESENVOLVIMENTO

As atividades designadas foram divididas em duas tarefas, A e B. A tarefa A, como mencionado acima, consiste na simulação de alocação de memória principal. Ao passo que, a tarefa B consiste na simulação rudimentar da memória virtual.

## 1. Tarefa A:

O objetivo desta tarefa é a implementação de uma versão de um sistema de alocação e desalocação de memória contendo os seguinte algoritmos: *first fit, worst fit, best fit* e *next fit*. Além dos algoritmos mencionados, métodos auxiliares também foram implementados. Tais métodos são a função de desalocação em que a chave é o identificador do processo, uma função capaz de contar quantos fragmentos externos há no segmento de memória (considerando que foi especificado que cada processo ocuparia de três à dez unidades de memória) e uma função capaz de contabilizar estatísticas do sistema em funcionamento.

Para que tudo funcionasse de maneira integrada um tipo abstrato de dados foi criado para que a partir de sua estrutura todo o sistema funcionasse. A estrutura principal é uma lista duplamente encadeada, além dos ponteiros para início e fim da lista há também um ponteiro para chamado *search* para indicar onde a busca deve começar (especificamente para o método next fit). O tamanho do segmento de memória e a quantidade de memória livre também estão presentes na estrutura. A célula da lista representa um pequeno fragmento de memória que pode estar ou não ocupado por um processo. Para que todas essas informações fossem organizadas de forma clara uma estrutura chamada *Segment* foi criada. Essa estrutura contém o identificador do processo (-1 caso o segmento esteja livre), o endereço de início, seu comprimento e status (livre ou ocupado por um processo). Toda a estrutura mencionada se encontra em um único arquivo header.

Há diversos módulos implementados baseados na estrutura acima descrita. Um dos módulos mais importantes é o *Allocation*, que contém todos os quatro algoritmos de alocação especificados na descrição do trabalho. O módulo *ComponenteDeDesalocacao* contém a função capaz de desalocar um pequeno segmento da memória e outra função auxiliar (*merge_free_cells()*) para que o processo ocorra de maneira adequada. O módulo *ComponenteDeMemoria* contém métodos básicos mas essenciais para o sistema, dentre tais métodos podemos citar os que são responsáveis por realizar a inicialização das estruturas, assim como os por mostrar em tempo real e exportar para arquivo (formato txt) o segmento completo. A função capaz de contar a quantidade de fragmentos externos também faz parte do módulo *ComponenteDeMemoria*.

Para que o sistema funcione um componente de requisições automáticas é necessário, dessa forma o módulo *ComponenteDeRequisicao* foi implementado com todos os métodos necessários. Este módulo basicamente gera um determinado número de requisições, especificado por parâmetro, podendo estas serem de alocação ou desalocação. Após a geração e execução das requisições tais dados são persistidos e utilizados para cálculo de métricas especificadas na descrição do trabalho. Tais métricas são exibidas por meio de um método implementado em outro módulo (*ComponenteDeEstatistica*) que apenas organiza as informações de maneira à facilitar a leitura e interpretação dos dados.

Por fim o arquivo main necessita apenas de invocar o método capaz de gerar requisições para que todo o sistema entre em funcionamento.

## 2. Tarefa B:

O objetivo desta tarefa foi a implementação de quatro algoritmos de substituição de páginas, em que a memória virtual possui um tamanho P - 1 e a moldura de página um tamanho M - 1, ambos definidos pela primeira linha de um arquivo de entrada. Além disso, foi necessário fazer o cálculo da taxa de erros ( *miss rate ).* Os algoritmos selecionados para implementação foram: **FIFO**, **Segunda Chance ( Second Chance )**, **Página de Relógio e Página menos recentemente utilizada.**

Em todos os algoritmos, a estrutura utilizada para a implementação da moldura de página foi uma fila e para a memória virtual um vetor. A estrutura fila foi escolhida pois é mais simples fazer a remoção das páginas, além disso, todos algoritmos escolhidos usam a fila. Visando um desempenho maior do sistema, o vetor foi selecionado como melhor estrutura para utilizar como memória virtual, pois além de já sabermos o tamanho dele ( P - 1 ), ele nunca iria mudar de tamanho.

Para a leitura do arquivo de entrada, foi criada uma fila, chamada de *Programa*, em que cada célula irá conter uma linha do mesmo. Isto foi feito para que não houvesse a necessidade de ficar lendo o arquivo toda hora, pois prejudicaria o desempenho do sistema.

A seguir há o detalhamento de como cada algoritmo é implementado.

**a. FIFO:**

Após selecionar a opção 1 do menu, o sistema irá chamar a função *fifo,* que tem como parâmetros um ponteiro para a primeira célula da fila programa, o tamanho da moldura de página, a página virtual que será acessada e por fim o missRate. A declaração da função *fifo* pode ser visto na imagem abaixo. Ao chamar essa função a fila de moldura de páginas, *queuePageFrame*, é criada. Ao final, é chamada a função *percorreProgramaFifo*.

```
void fifo( Programa prog, int sizePageFrame, int *virtualPage, int missRate );
```

A função percorreProgramaFifo irá percorrer a fila em que cada célula contém uma instrução que foi previamente lida do arquivo, até encontrar o final do mesmo. Caso a célula possua um caracter “R”, será chamada a função *percorreLista*, que irá examinar todas as molduras de página pesquisando se a página já está na moldura. Em caso negativo, séra retornado 0 e o missRate será incrementado. Abaixo pode-se observar na imagem o trecho de código que realiza a leitura da moldura de página.

```
if( aux -> ins == 'R'){

    if( percorreLista( queuePageFrame, aux -> numVirtual ) == 0 ){
        missRate++;

    }
}
```

Caso a célula da fila *programa*, possua um caractér “W”, o sistema irá fazer três comparações. A primeira será para verificar se a página está na memória, em caso afirmativo, será mostrado uma mensagem na tela do usuário informando que a página está na memória. A segunda comparação irá averiguar se as molduras de páginas já estão todas ocupadas, se sim, irá retirar a primeira página, inserir no fim da moldura e acrescentar o *missRate*. E por último, se a página não estiver na memória e a mesma não estiver cheia, a página será acrescentada no final e o

*missRate* será acrescido. O trecho de código que realiza os passos citados pode ser visto na imagem abaixo.

```
if( aux -> ins == 'W'){

    //Se a página ja estiver na memória
    if( percorreLista( queuePageFrame, aux -> numVirtual ) == 1 ) {
        printf("Pagina ja esta memoria\n");
    }
    //Caso a página não esteja na memória
    else if( percorreLista( queuePageFrame, aux -> numVirtual ) == 0 ){

        //Se a moldura já estiver cheia
        if( count == sizePageFrame ){
            retiraPage( queuePageFrame );
            inserePage( virtualPage[aux -> numVirtual] , queuePageFrame );

            missRate++;
        }
        else{

            inserePage( virtualPage[aux -> numVirtual] , queuePageFrame );
            count++;
            missRate++;
        }

    }
```

Após ter lido todas as entradas do arquivo, para finalizar, a função fifo irá chamar a função *imprimeMissRate,* que tem como objetivo mostrar na tela do usuário a taxa de *missRate* em porcentagem. A figura abaixo mostra a implementação da função citada.

```
void imprimeMissRate( int missRate, int count_instr ){

    printf("MISS RATE = %d%%\n", (missRate * 100)/ count_instr);

}
```

**b. Segunda Chance ( Second Chance ):**

Selecionado a opção 2 do menu, será selecionado o algoritmo de segunda chance. Primeiramente será criado uma fila chamada *queuePageFrame* e logo após isso será chamada a função *percorreProgramaSC,* como pode ser visto na figura abaixo.

```
void secondChance( Programa prog, int sizePageFrame, int *virtualPage, int missRate ){
    PageFrame queuePageFrame;
    FPVazioPage( &queuePageFrame );
    percorreProgramaSC( prog, sizePageFrame, &queuePageFrame, virtualPage, missRate );

}
```

A função percorreProgramaSC, é semelhante a função percorreProgramaFifo citada anteriormente no tópico FIFO. Porém há diferenças quando os caracteres “R” e “W” são lidos.

Quando o caracter lido é “R”, será chamada a função percorreLista que irá percorrer as molduras de páginas verificando se a página que se está pesquisando se encontra na moldura. Em caso de não estar, será acrescentado o missRate, caso a página estiver será chamada a função setBitR que irá colocar o bit R em 1. Na figura abaixo pode-se visualizar como é feito a implementação citada acima.

```
if( aux -> ins == 'R'){

    // página não está na fila
    if( percorreLista( queuePageFrame, aux -> numVirtual ) == 0 ){
        missRate++;

    }
     if( percorreLista( queuePageFrame, aux -> numVirtual ) == 1 ){
        setBitR( queuePageFrame, aux -> numVirtual );

    }
}
```

Caso o caracter lido seja “W”, serão feitas três comparações. A primeira é para verificar se a página está na memória, em caso afirmativo, será mostrada uma mensagem na tela do usuário informando que a página que vai ser inserida já está na memória. Caso a página não esteja na memória, será verificado se as molduras de página estão todas ocupadas. Se sim, será chamada a função *searchBitR*, se não será inserida a página ao final da lista de molduras. Em ambos os casos o missRate é acrescentado. A função searchBitR percorre a lista de moldura de páginas analisando o bit R de cada página. Se o bit R da página for igual a 0 ela será removida da moldura, e a página que se deseja incluir é colocada ao final da fila. Caso o bit R da página seja 1, o bit R será alterado para 0 e ela será inserida ao final da fila. A pesquisa irá continuar até encontrar a primeira página cujo o bit R seja igual a 0. Nas figuras abaixo pode-se visualizar todo o processo citada acima.

```
else if( aux -> ins == 'W'){

    //Se a página já estiver na memória
    if( percorreLista( queuePageFrame, aux -> numVirtual ) == 1 ) {
        printf("Pagina ja esta memoria\n");
    }
    //Se a página não estiver na memória
    else if( percorreLista( queuePageFrame, aux -> numVirtual ) == 0 ){

        //Se a moldura já estiver cheia
        if( count == sizePageFrame ){
            searchBitR( queuePageFrame, aux -> numVirtual );
            missRate++;
        }
        else{

            inserePage( virtualPage[aux -> numVirtual] , queuePageFrame );
            count++;
            missRate++;
        }
    }
```

```
void searchBitR( PageFrame *pageFrame, int page ){

    PageFrame aux;
    aux.primeiroPage = pageFrame->primeiroPage;


    while( pageFrame -> primeiroPage -> prox!= NULL ){

        if( pageFrame -> primeiroPage-> bitR == 0 ){

            retiraPage( &aux );
            inserePage( page, pageFrame );
            break;
        }
        if( pageFrame -> primeiroPage -> bitR == 1){
            pageFrame -> primeiroPage -> bitR = 0;

            retiraPage(&aux);
            inserePage( pageFrame -> primeiroPage -> numPage, pageFrame );


        }
        pageFrame -> primeiroPage  = pageFrame -> primeiroPage -> prox;
    }

    pageFrame -> primeiroPage = aux.primeiroPage;

}
```

Após ter lido todas as entradas do arquivo, para finalizar, a função *imprimeMissRate* é chamada. Ela tem como objetivo mostrar na tela do usuário a taxa de *missRate* em porcentagem. A figura abaixo mostra a implementação da função citada.

```
void imprimeMissRate( int missRate, int count_instr ){

    printf("MISS RATE = %d%%\n", (missRate * 100)/ count_instr);

}
```

**c. Substituição de página de relógio**

O algoritmo de substituição de página de relógio é muito semelhante ao do segunda chance. Todo o processo é basicamente o mesmo, passando pela construção da fila *queuePageFrame,* e da função que percorre a fila contendo o programada, que recebe o nome *percorrePrograma*Relógio.

A diferença entre os dois algoritmos está apenas na forma como o bit R é usado para realizar a substituição. Neste caso, caso a moldura de página esteja cheia, uma função para percorrê-la é chamada, consultando o bit R. Se o bit R estiver setado em 0, a página é removida. No caso de bit R igual a 1, ele é setado para 0 e o programa continua a percorrer a procura de um bit 0.

Para que o programa continue percorrendo mesmo após o final da fila, caso não ache bit 0, o ponteiro *ultimo->prox* é apontado para a primeira posição da fila, formando uma fila circular.

A função que faz a busca e seta o bit R pode ser vista na figura abaixo:

```
void searchBitRelogio( PageFrame *pageFrame, int page ){

    PageFrame aux;
    aux.primeiroPage = pageFrame->primeiroPage;


    while( pageFrame -> primeiroPage -> prox!= NULL ){

        if( pageFrame -> primeiroPage-> bitR == 0 ){

            retiraPage( &aux );
            inserePage( page, pageFrame );
            break;
        }
        if( pageFrame -> primeiroPage -> bitR == 1){
            pageFrame -> primeiroPage -> bitR = 0;

            pageFrame -> primeiroPage  = pageFrame -> primeiroPage -> prox;

        }
    }

    pageFrame -> primeiroPage = aux.primeiroPage;
```

**d. Página menos recentemente utilizada**

O algoritmo de página menos recentemente utilizada segue o padrão de início de todos os outros três algoritmos. A criação da lista e tratamento dos comandos “W” e “R”.

Para que se possa saber qual a página menos recentemente utilizada, uma fila é mantida, tendo as operações de inserir e retirar com algumas modificações. A inserção de uma nova página acontece sempre no início da fila e a retirada sempre no final, fazendo com que as páginas menos recentemente utilizadas fiquem nas últimas posições.

A implementação das funções de inserção e retiradas podem ser vistas na imagem abaixo:

```
void inserePageNfu(int numPage, PageFrame *pageFrame)
{
    ApontadorPage nova,antigo;
    nova = (ApontadorPage) malloc(sizeof(CelulaPage));
    nova->prox = pageFrame->primeiroPage;
    pageFrame->primeiroPage = nova;
    nova->numPage = numPage;
}

void retiraPageNfu(PageFrame *pageFrame)
{
    ApontadorPage ultimo, penultimo;
    ultimo = pageFrame->primeiroPage->prox;
    penultimo = pageFrame->primeiroPage;

    if(VazioPage(*pageFrame))
    {
        printf("Erro! Moldura de pagina vazia.\n");
        return;
    }

    while(ultimo->prox != NULL)
    {
        penultimo = ultimo;
        ultimo = ultimo->prox;
    }

    penultimo->prox = NULL;
    free(ultimo);
}
```

# RESULTADOS E CONCLUSÃO

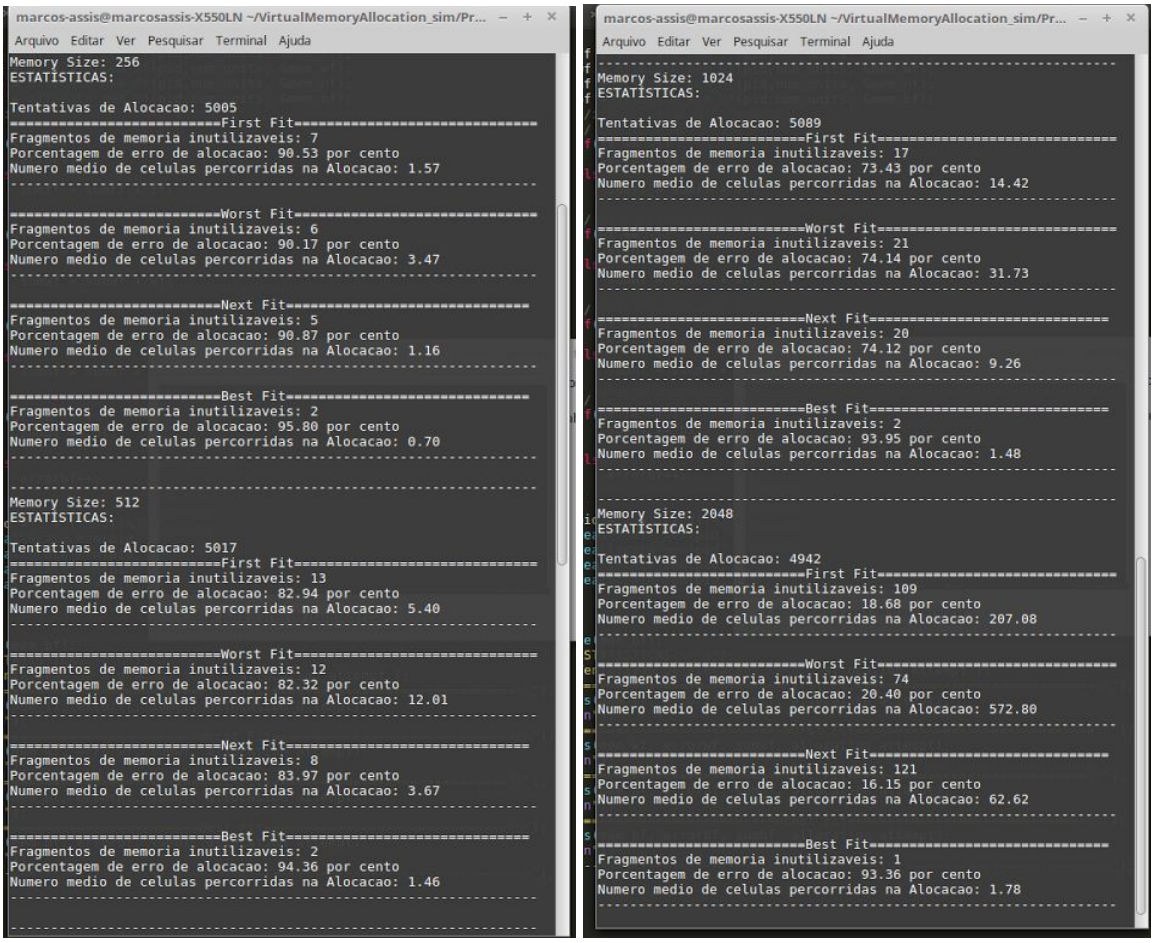


Figura 1 - Estatisticas da Tarefa A

Na tarefa A, como as requisições de alocação e desalocação foram geradas aleatoriamente, o número de alocações e desalocações eram praticamente o mesmo. Percebemos que a taxa de erros diminuiu conforme o tamanho da memória aumenta.

Curiosamente, o algoritmo de best fit foi o que Melhor e pior se comportou, pois ele teve o menor número médio de celular percorridas e também a maior porcentagem de erro. Tirando isso, o Worst Fit é o algoritmo que mais percorre a lista e o o algoritmo Next Fit o que menos percorre.

Na tarefa B, por meio dos testes realizados por meio dos arquivos, é possível perceber que o algoritmo de troca de página *Segunda Chance* ( *Second Chance* ) pode apresentar uma taxa de erros menor que o algoritmo *FIFO*, pois ele permite que páginas muito requisitadas estejam sempre na memória, ou seja, ele leva em

consideração se a página é muito acessada. Além disso, foi possível analisar que existe um caso especial do *Segunda Chance*, que é quando todos os bits R são iguais a 0. Quando isso ocorre, o algoritmo funciona exatamente ao algoritmo *FIFO.*

O algoritmo de substituição de página de relógio sana uma falha que o algoritmo de segunda chance comete, que é manter uma página como prioridade apenas por ela já ter sido referenciada várias vezes, o que não garante que ela ainda continua sendo muito consultada.

O modelo de substituição de página menos recentemente utilizada se mostrou o mais eficiente na questão de número de “miss” e isso sem que tenha um alto custo, pois não é necessário, como nos outros algoritmos, percorrer toda a lista a procura de qual página retirar, bastando retirar a página que ocupa a última posição.

# REFERÊNCIAS


- Tanenbaum, A. Sistemas Operacionais Modernos, 3a Edição, Editora Pearson Prentice Hall, 2010.