Relatório 4: Simulação de algoritmos de substituição de páginas

Ana Clara Mueller Miranda - RA 148806
Bruno Pires Moreira Silva - RA 139892
Igor Ribeiro Ferreira de Matos - RA 140492
Rafaela Cristine dos Santos Uchôas - RA 140351

Para a implementação do algoritmo de página definimos uma lógica parecida com a de processos em que definimos uma estrutura de dados simples em que se pesquisa página por página e verifica se ela está endereçada na memória principal no disco (que simulamos utilizando um loop de tamanho pré-definido), que está inicialmente cheia com páginas aleatórias geradas no início do código. Dessa forma é possível validar se ocorreu um page miss, basta fazer uma simples busca em nossa memória, se a página não foi encontrada é considerado um page miss.

No nosso código nós escolhemos dois algoritmos para implementar e analisar qual deles gera menor número de page miss: Um algoritmo FIFO (First In, First Out; um algoritmo que é composto por uma fila encadeada de páginas) e um algoritmo de fila circular, também chamado de relógio (que também em sua essência é uma fila encadeada, com a diferença de que o fim da fila aponta para o seu começo). A hipótese inicial do grupo foi de que o funcionamento dos dois algoritmos iria ser bem parecido em questão de complexidade, já que a busca em ambos os algoritmos é O(n), mas com o algoritmo de FIFO com custo menor e desempenho ligeiramente pior.

Nossa hipótese inicial é essa pois o algoritmo FIFO é simplesmente uma lista encadeada, logo o custo de busca seria no pior caso percorrer toda a lista, e remoção e inserção são sempre a mesma coisa, retira do começo a adiciona no final, em todos os casos. Enquanto que no algoritmo de relógio no pior caso também se procura toda a lista, porém na hora de decidir qual página deve ser retirada para outra ser adicionada no pior caso ele pode ter de percorrer toda a lista novamente. Vejamos um exemplo: a página x está sendo referenciada por um processo, é feito a busca, na FIFO e no relógio, ela não é encontrada em ambos, até aqui temos que ambos foram percorrida por completo igualmente (dado que nossa memória tem tamanho igual para ambos os algoritmos, percorrer a lista é igual). Agora na hora de decidir a remoção; para FIFO é simples, removesse a primeira e adiciona x ao final da lista, para o relógio começamos olhando o último elemento (apontado pelo ponteiro), caso o bit R=0, podemos fazer a substituição imediatamente, porém se ele for 1 temos de limpar o

bit R, e depois passar para o próximo e fazer essa mesma checagem novamente. No pior caso todos os bits R são iguais a 0, e percorremos toda a lista novamente até chegarmos no primeiro bit R que limpamos que nesse caso ele será 0 e a substituição pode ser feita. Logo o algoritmo irá demorar mais que o FIFO, porém como o algoritmo de relógio retira apenas páginas que não foram referenciadas na última interrupção (pois checa se o bit R=0) a chance dele retirar uma página útil é menor que no algoritmo FIFO que pouco se importa se a página foi referenciada ou não. Apenas no pior caso (onde todos os bits R são iguais a 1) que o algoritmo tem a mesma chance de retirar uma página útil que FIFO.

Além disso, definimos uma estrutura chamada 'pag' com variáveis do tipo uint16_t (variáveis de 16-bits sem sinal, pode armazenar valores entre 0 e 65.535). A sintaxe dessa struct é diferente pois esse tipo de variável tem valor 0 ou 1 caso acabe em ":1" e tenha um valor entre 0 e 12 caso acabe em ":12". Essa struct é a simulação de uma página. Temos também uma outra estrutura padrão de lista, que contém uma página e um ponteiro para a página anterior (foi escolhido dessa forma pois como retiramos do início da fila precisamos saber quem é seu antecessor para podermos atualizar o ponteiro que aponta para o início da lista, e para adicionar fica muito rápido basta apontar o ponteiro do último elemento ao novo elemento), formando uma lista encadeada.

```
typedef struct pag{
    uint16_t presente:1; // esta ou nao mapeado em memoria fisica
    uint16_t referenciada:1; //se processo ja leu a pagina
    uint16_t modificada:1; //se processo alterou
    int numpag:12;
}pag;

typedef struct lista{
    pag pagina;
    struct lista *ant;
}lista;
```

Definição das estruturas

A primeira coisa que fizemos depois da implementação desses dois algoritmos, ou seja, depois de definir como vai ficar a estrutura das páginas, foi encher a memória, como dito anteriormente através de um laço com um tamanho finito definido anteriormente.

Para isso, criamos um vetor com o tamanho de memória definido, e colocamos os todos os números possíveis dentro dele, fizemos então uma função shuffle que dado um vetor,

ela embaralha o vetor, usamos essa ideia antes de inserir para simular um endereço de memória aleatório e não ficar uma sequência.

```c
void shuffle(int array[]){
    srand(time(NULL));
    int i;
    for(i=0;i<numedereco;i++){
        array[i]=i*2;
    }
    if (numedereco > 1) {
        for (i = 0; i < numedereco - 1; i++){
          int j = i + rand() / (RAND_MAX / (numedereco - i) + 1);
          int t = array[j];
          array[j] = array[i];
          array[i] = t;
        }
    }
}
```

Função shuffle

A função adicionar tem o funcionamento de uma lista normal, ou seja, pega o começo da lista, checa se tá vazio e caso sim insere o item, se não vai até o ponteiro fim e insere no final. A função remover também funciona da mesma forma que uma lista normal, no nosso caso ela remove do começo da fila, e é chamada em casos em que ocorre o page miss.

```c
void adicionar(lista *list, pag pagina, int alet){
    lista *ptr = (lista*)malloc(sizeof(lista));

    ptr->pagina.referenciada=1;
    ptr->pagina.numpag=alet;

    ptr->ant=NULL;
    if(inicio==NULL && fim==NULL){
        inicio=ptr;
        fim=ptr;
    }
    else{
        lista *aux = fim;
        aux->ant=ptr;
        fim=ptr;
    }
    return inicio;
}
```

Função adicionar da FIFO.

```
lista *remover(){
    lista *aux = inicio;
    inicio = inicio->ant;
    return aux;
}
```

Função remover da FIFO.

A mesma lógica básica de lista circular foi usada para adicionarCircular e removerCircular, como é possível ver nas imagens a seguir:

```
void adicionarCircular(lista *list, pag pagina, int alet){
    lista *ptr = (lista*)malloc(sizeof(lista));
    if(inicio==NULL){
        ptr->pagina.referenciada=1;
        ptr->pagina.numpag=alet;
        inicio=ptr;
        inicio->ant=inicio;
    }else{
        lista *aux = inicio;
        while (aux->ant != inicio)
            aux = aux->ant;

        aux->ant=ptr;
        ptr->ant=inicio;
        ptr->pagina.numpag=alet;
        ptr->pagina.referenciada=1;
    }
}
```

Função adicionar da lista circular

```
lista *removerCircular(){
    lista *aux = inicio;
    while (aux->ant != inicio)
        aux = aux->ant;

    return aux;
}
```

Função remover da lista circular

A função 'enchememoria' foi feita utilizando um vetor global, logo a memória para ambos os algoritmos começa exatamente igual para ambos começarem de forma justa. A função pega o vetor com os inteiros aleatórios criados e adiciona nas nossas listas. Como chamamos primeiro a função enche memória da lista circular, somente ela chama a função shuffle para encher o vetor com os números aleatórios, a função enche memória da FIFO não precisa então chamar a função shuffle basta adicionar as páginas com os números aleatórios que já foram gerados. Com auxílio do professor decidimos que a quantidade de itens ideal para colocar na lista é 100, já que é uma quantidade grande o suficiente para que vemos diferença de performance mas não grande demais a ponto de dificultar os testes e a análise final.

```
void enchememoria(lista *list){
    //shuffle(nums);
    lista *paginas = (lista *)malloc(sizeof(lista));
    inicia();
    int i;
    for(i=0;i<numedereco;i++){
        pag pg;
        adicionar(paginas, pg, nums[i]);
    }
}

void enchememoriaCircular(lista *list){
    shuffle(nums);
    lista *paginas = (lista *)malloc(sizeof(lista));
    inicia();
    int i;
    for(i=0;i<numedereco;i++){
        pag pg;
        adicionarCircular(paginas, pg, nums[i]);
    }
}
```

Função ‘enchememoria’ dos dois algoritmos

A função ‘procura’ pega como parâmetro uma página e checa se ela está na lista, se ela encontrar a página ela retorna 1 e se não encontrar significa que ocorreu um page miss, ou seja, chama a função pagemiss. Ao chamar a função pagemiss os algoritmos são implementados removendo e adicionando nos locais que cada um especifica.

```c
int procura(lista *Lista, pag pg){ //procura pelo endereco da página que é o numpag
    lista *aux=inicio;
    while(aux->ant!=NULL){
        if(aux->pagina.numpag==pg.numpag){
            return 1; //achou pagina
        }else{
            aux=aux->ant;
        }
    }
    numpagemiss++;
    printf("page miss\n");
    pagemiss(Lista, pg, pg.numpag);
    return 0;//page miss
}
int procuraCircular(lista *Lista, pag pg){ //procura pelo endereco da página que é o numpag
    lista *aux=inicio;
    while(aux->ant!=inicio){
        if(aux->pagina.numpag==pg.numpag){
            return 1; //achou pagina
        }else{
            aux=aux->ant;
        }
    }
    numpagemiss++;
    printf("page miss\n");
    pagemiss(Lista, pg, pg.numpag);
    return 0;//page miss
}
```

Função ‘procura’ dos dois algoritmos.

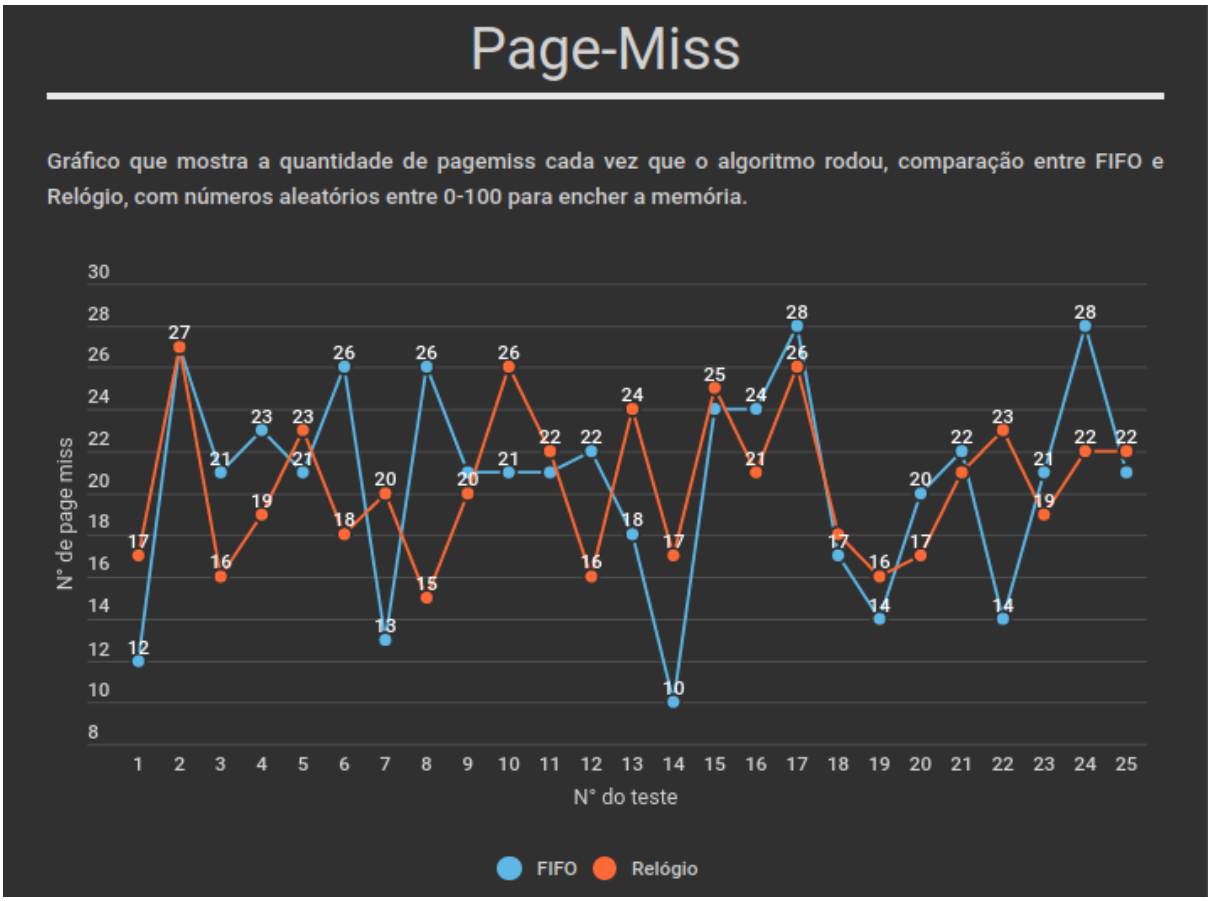
Para nossa main implementamos dois loops, um para lista circular e outro para FIFO, ambos com exatamente a mesma lógica, números aleatórios são gerados e procurados nas listas, se a procura tiver sucesso o loop apenas continua, se não imprimimos qual página que fez com que ocorresse o page miss.

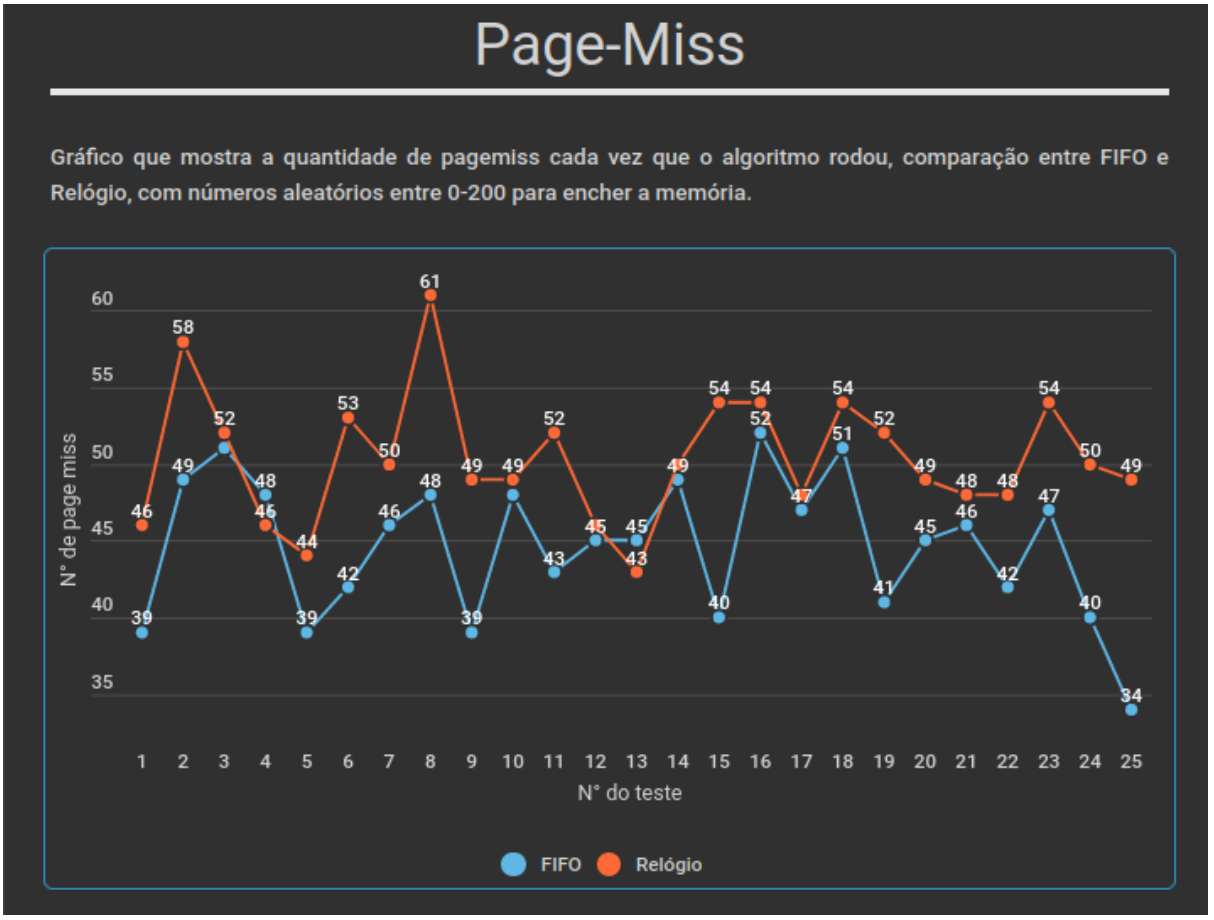
```c
int main(){
    enchememoriaCircular(NULL);
    int i=100;
    lista *circular = inicio;
    pag p;
    printf("Memoria preenchida com valores entre 0-100\n\nAlgoritmo de Relógio\n");
    i = 100;
    circular = inicio;
    int numpagemiss=0;
    while (i){
        p.numpag = rand()%125;
        if (!procuraCircular(inicio, p)){
            printf("%d\n", p.numpag);
            numpagemiss++;
        }
        i-=1;
    }
    printf("Numero de pagemiss: %d\n", numpagemiss);
    limpar(circular);
    printf("\nMemoria preenchida com os mesmos valores entre 0-100\n\nAlgoritmo FIFO\n");
    enchememoria(NULL);
    lista *FIFO = inicio;
    i = 100;
    FIFO = inicio;
    numpagemiss=0;
    while (i){
        p.numpag = rand()%125;
        if (!procura(inicio, p)){
            printf("%d\n", p.numpag);
            numpagemiss++;
        }
```

```c
        }
        i-=1;
    }
    printf("Numero de pagemiss: %d", numpagemiss);

    return 0;

}
```

Abaixo vemos 2 gráficos com os teste que foram feitos, o primeiro gráfico geramos números aleatórios de 0-100 para encher a memória, e vemos que os algoritmos tiveram desempenhos semelhantes, o FIFO se saindo melhor em 11/25 e do Relógio 14/25, porém mesmo quando um se saía melhor a diferença era pequena. O segundo gráfico mostra os resultados quando geramos números aleatórios entre 0-200, vemos nitidamente que o FIFO se sai melhor, 23/25 testes feitos FIFO se saía melhor e vemos que a diferença entre eles também aumentou. Além disso ambos tiveram muito mais page miss, isso acontece pois 100 números aleatórios entre 0-200 teremos muito mais lacuna que 100 números aleatórios entre 0-100 dentro da memória, então há mais chances de termos um page miss. Fizemos esse dois testes diferentes para vermos em situações de grandes lacunas, qual algoritmo se sairia melhor. Como nossa simulação tem algumas partes faltantes, por exemplo a simulação de conjunto de trabalho de cada processo, algumas páginas seriam mais referenciadas, porém

fizemos aleatoriamente por questão de simplificação. Confirmamos uma parte da nossa hipótese, que os algoritmos teriam um desempenho próximo, porém achávamos que o algoritmo relógio teria melhor desempenho, entretanto de modo geral o FIFO teve um desempenho melhor.





Divisão de atividades:

- Algoritmo FIFO: Ana Clara, Rafaela Uchôas
- Algoritmo Relógio: Bruno Pires, Igor Ribeiro
- Relatório: Todos
- Teste+Gráficos: Todos