UNIVERSIDADE FEDERAL DO PIAUÍ - CAMPUS SHNB
SISTEMAS DE INFORMAÇÃO
ESTRUTURAS DE DADOS II

Camila Catiely de Sá Almondes
Firmino Azevedo Neto
Laécio Andrade Rodrigues

Trabalho Prático

1 Introdução

A memória RAM é um local utilizado pelo processador para armazenar as informações de tudo o que estiver em execução no sistema. Dessa forma, a medida que novas aplicações vão sendo abertas, mais memória é consumida. O problema é que o espaço da memória principal dos computadores é limitado, e com a evolução constante das tecnologias, as aplicações tem requerido cada vez mais espaço nessa memória, o que torna comum a situação em que aplicações podem requerer mais espaço do que a memória principal pode oferecer.

Por isso que na memória secundária os arquivos devem conter mais registros do que na memória interna pode armazenar, além do custo para acessar que é maior do que na memória primária. Trabalho também será diminuir o custo de acesso e o tempo, já que na memória secundária acessa apenas um registro em um dado momento (acesso sequencial). Modelo de armazenamento em dois níveis, devido à necessidade de grandes quantidades de memória e o alto custo da memória principal. Uso de uma pequena quantidade de memória principal e uma grande quantidade de memória secundária. Programador pode endereçar grandes quantidades de dados, deixando para o programa a responsabilidade de transferir o dado da memória secundária para a principal.

Transformar o endereço usado pelo programador no endereço físico de memória correspondente. Espaço de Endereçamento seriam os endereços usados pelo programador. Espaço de Memórias são as localizações de memória no computador. O espaço de endereçamento N e o espaço de memória M podem ser vistos como um mapeamento de endereços do tipo: f: N → M. O mapeamento permite ao programador usar um espaço de endereçamento que pode ser maior que o espaço de memória primária disponível na qual irá utilizar um sistema de Paginação

2 Solução Proposta

Uma das soluções mais populares para a lotação da memória principal é a memória virtual, que consiste em arquivos de paginação que nada mais são do que um espaço no disco rígido reservado para ajudar a armazenar os dados da memória RAM quando ela está cheia. É uma forma de estender a quantidade de memória para os dados temporários utilizados pelos aplicativos em execução sem que você precise fazer um upgrade de hardware.

Neste trabalho será implementado em linguagem C, um simulador de memória virtual, que está organizado em uma estrutura de dados do tipo árvore B. Utilizando o princípio de ordem de chegada FIFO (First In First Out), ou seja, assim que a memória virtual estiver cheia e uma inserção for solicitada, o primeiro elemento inserido na lista deve ser removido para que o novo elemento seja incluído, e toda vez que o dado procurado pela aplicação não estiver na memória primária, busca-se então o dado na memória secundária. Caso a memória primária tenha espaço para o dado buscado, o dado será colocado no fim da fila. Caso não haja espaço na memória primária, então

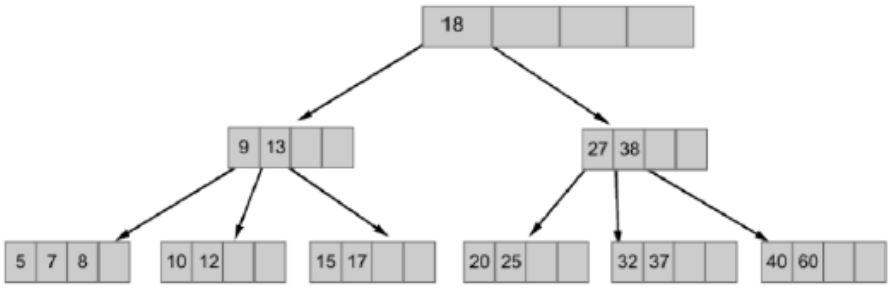
pela política de reposição FIFO, desenfileiramos o primeiro elemento, abrindo assim espaço na memória, e enfileiramos o registro requisitado pela memória. Assim o elemento residente a mais tempo na memória primária sempre ocupará a primeira posição da fila. A própria estrutura de dados fila tem esta característica First In First Out, permitindo a aplicação da política FIFO por meio dela, sempre removendo itens do começo da fila e inserindo itens no final da fila. A complexidade desta política de reposição é O(1), visto que as operações de enfileirar e desenfileirar feitas sobre a fila, são O(1).

Os parâmetros de entrada estão organizados num arquivo de texto chamado “entrada.txt” contendo o seguinte conteúdo:

72 2
18
10 5 7 20 9 13 18 32 15 38 40 8 60 27 17 12 37 25
4
40 5 5 40

Na primeira linha estão contidos dois valores que representam as especificações do espaço em bytes na memória virtual e a ordem da árvore, na segunda linha temos a quantidade de valores que serão inseridos na árvore, a terceira linha contém os valores que serão inseridos na estrutura, a linha seguinte possui o número de valores que serão procurados na árvore, e na última linha estão os valores especificados na linha anterior, que são os que serão procurados na árvore. Os valores apresentados na linha quatro do arquivo de entrada estão organizadas em uma estrutura de dados do tipo fila para que o princípio FIFO seja obedecido.

Esta é a árvore obtida após a inserção das chaves:



A árvore B de ordem M é um tipo de árvore onde cada nó (página) deve conter:

- No mínimo M registros e no máximo 2M registros (com exceção da raiz que pode conter de 1 a 2M registros).
- 2M+1 apontadores.
- No mínimo M+1 descendentes e no máximo 2M+1.

1. Localizar a página apropriada aonde o registro deve ser inserido.

2. Se o registro a ser inserido encontra uma página com menos de 2 m registros, o processo de inserção fica limitado à página.

3. Se o registro a ser inserido encontra uma página cheia, é criada uma nova página, no caso da página pai estar cheia o processo de divisão acontece.

Algumas mudanças foram feitas para atender a especificação. Sendo elas o modo como os vetores de registros e os vetores de apontadores são alocados. Na implementação encontrada, foram usados os parâmetros do arquivo de entrada para definir a ordem da árvore B, enquanto no programa criado o vetor registros e o vetor de apontadores são alocados dinamicamente, pois a ordem da árvore B é passada para o programa. O SMV consiste basicamente no mapeamento entre endereços físicos e virtuais, carregando da memória secundária para a memória primária, em forma de páginas, que são blocos de dados, os dados que esta necessitar. Em caso de a memória primária está cheia, são aplicadas políticas de reposição de página.

Todo o processo é feito por um sistema de Paginação:

O espaço de endereçamento é dividido em páginas de tamanho igual de 512 Kbytes. A memória principal é dividida em molduras de páginas de tamanho igual. As molduras de páginas contêm algumas páginas ativas enquanto os restantes das páginas estão em memória secundária (páginas inativas). O mecanismo possui duas funções: Mapeamento de endereços na qual irá determinar qual página um programa está endereçando, encontrar a moldura, se existir, que contenha a página. Transferência de páginas que irá transferir páginas da memória secundária para a memória primária e transferi-las de volta para a memória secundária quando não estão mais sendo utilizadas.

Memória Virtual contendo o sistema de Paginação com o endereçamento da página contendo uma parte dos bits é interpretada como um número de página e a outra parte como o número do byte dentro da página (offset) e Mapeamento de endereços realizado através de uma Tabela de Páginas.

A p-ésima entrada contém a localização p ′ da Moldura de Página contendo a página número p desde que estejam na memória principal. O mapeamento de endereços é: f (e) = f (p, b) = p ′ + b, onde e é o endereço do programa, p é o número da página e b o número dos bytes. Se não houver uma moldura de página vazia → uma página deverá ser removida da memória principal.

Funções:

```
Fila* cria_f();
void insere_f(Fila* f,int v);
void retira(Fila* f);
int buscar_f(Fila* f, int valor);
void reorganiza(Fila* f);
void libera(Fila* f);
```

Estas funções fazem parte do tipo abstrato de dados fila.

```
int buscar_f(Fila* f, int valor);
```
Buscar um valor na memória principal.

```
FILE * abreArquivoLeitura(char * arquivo);
```

Para Abrir um arquivo de leitura após toda a criação.

FILE * abreArquivoEscrita(char * arquivo);
Começará a execução dessa função de escrita na memória e abrir o arquivo.

Arv * criarNo(int val, Arv *no, int ordem);
Criar um novo nó na árvore B tree.

void adcValorNo(int val, int pos, Arv *no, Arv *novo);
Coloca o valor na posição apropriada da árvore, adiciona no nó.

void dividirNo(int val, int *pval, int pos, Arv *no,Arv *novo, Arv **novoNo, int ordem);
Tira a média do nó e esse nó será dividido e logo após irá surgir um novo nó.

int defineChaveNo(int val, int *pval,Arv *node, Arv **child, int ordem);
define o valor chave no nó, e seta esse valor.

void inserir(int val, int ordem);
inserir chave em B-Tree, o valor.

int buscarChave(Fila* f, int val, int *pos, Arv *myNode);
buscar chave na B-Tree, percorrendo todas páginas e valores.

int main(int argc, char *argv[])
Função principal;

O processo de paginação é feito no Main com a chamada das funções e com as entradas e saídas.

int paginas = (memoria/(4*(2*ordem) + 4 * (2 * ordem + 1)));
Definindo número de páginas;

Analise:

Complexidade

A árvore B tem como sua principal vantagem a complexidade logarítmica nas funções de inserção, retirada e pesquisa, pois a árvore estará sempre balanceada, ou seja, não há folhas em níveis diferentes, o que é princípio básico para a construção de uma árvore B. Assim as operações de retirada, inserção e pesquisa em árvore B têm complexidade O(log(n)), onde n é o número de registros na árvore.

Cada complexidade separada dos algoritmos será O(n), onde n e o numero de páginas na memória, igual para cada um deles. Porém, a complexidade temporal do programa deve levar em consideração a quantidade instâncias, de páginas e de acessos. Assim, nosso cálculo de complexidade poderia ser

definido como número de páginas * número de acessos * 4 . Nesses 4 parâmetros, o que tem um maior peso seria os acessos, pois ele que irá dominar todos os outros. Por isso, podemos dizer então, que a complexidade temporal final do programa será de ´ O(n), porém n aqui será a quantidade de acessos feitos.

Pior caso: $O(n^2(\log(r)))$.

Pseudocódigo das operações:

ProcuraRegistroNaMemoria (Registro);
O registro foi encontrado?
Se sim:
Hit;
TrataHit ();
//Cada política tem tratamento diferente para este caso
Se não: Misses = Misses + 1;
Para cada página do caminhamento
A página está na memória?
Se sim: Hit;
TrataHit();
Se não: Há espaço na memória?
Se sim:
ColocaNaMemória(Pagina);
Se não: RetiraPaginaDaMemória(PaginaParaRetirar);
ColocaNaMemoria(Pagina);
Quando há um hit e quando colocarmos um registro na memória cheia é que entram em ação as políticas de reposição de páginas.

Tempo de Execução

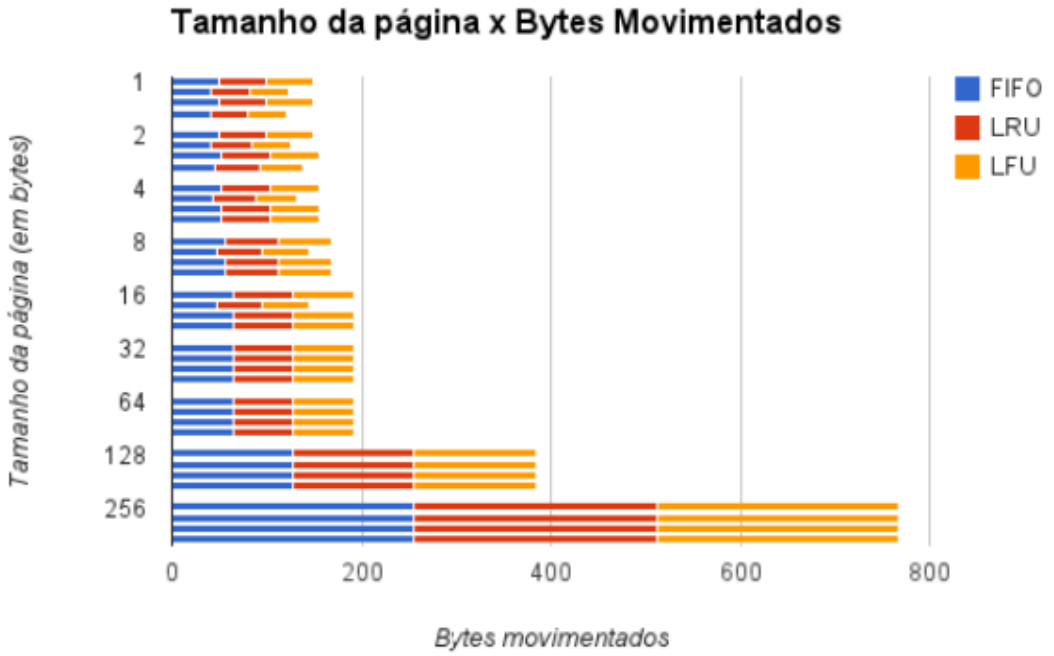
Gráfico 1:

Árvore B de ordem 4 e os registros gerados sendo aleatórios. Foram excluídos 5%, pesquisados 10% e traçado o caminhamento de 5% dos itens iniciais. Para o 6 primeiro experimento, visando o tempo de execução, a memória continha 60 páginas e o número de registros foi variado, de 100 a 50 mil registros. Variando apenas o tamanho da entrada, mantendo o tamanho da memória fixo com 60 páginas.

Gráfico 2 :

Como foi dito na especificação do trabalho páginas maiores irão certamente diminuir o número de falhas, por causar do carregamento desnecessário de muitos dados, então podemos verificar isso ocorrendo de fato em nossos testes. A política adotada foi a seguinte: foi fixado o tamanho da memória em 256 bytes, e foi variados os tamanhos de cada página 1 até 256. Para páginas pequenas (1 a 8 bytes), podemos ver que poucas falhas ocorreram, consequentemente, poucos dados foram movimentados.

Já no caso de páginas maiores (Memória fixa em 256 bytes) a quantidade de falhas reduziu drasticamente, porem a quantidade de bytes movimentados também cresceu vertiginosamente. A performance de cada uma das políticas de reposição nessa situação foi igual para todas. Todas obtiveram o mesmo número de falhas e, consequentemente, a mesma quantidade de bytes movimentados.



Máquina utilizada

4 Conclusão

Este trabalho tem como objetivo nos fazer entender na prática o funcionamento dos Sistemas de Memória Virtual, que gerenciam a memória, dando aparência de um espaço de memória maior do que o existente, que é uma solução proposta para o problema da falta de espaço na memória principal e a demora nas buscar pelos os valores . Aplicando os conhecimentos de estruturas de dados e suas políticas de reposição de páginas para solucionar o problema. Conseguimos a criação da memória secundária e o armazenamento de valores, e a análise de seus dados de entrada e saída.

5 Referências:

Todos os documentos utilizados como base estão nesses repositórios no link :< https://github.com/search?l=C&q=memoria+virtual&type=Repositories>

Roiz, Enzo. Trabalho Prático 1: Sistemas de Memória Virtual.

Miccoli, Sandro. Trabalho Prático 3: Memória Virtual.