Trabalho Prático 1
Sistemas de Memória Virtual

Aluno: Enzo Roiz

Matrícula: 2013062839

Índice

Descrição do problema

A memória primária é a memória visível para as aplicações, pois é endereçada diretamente pelo processador. Porém o espaço de memória primária é bem limitado. Assim quando o conjunto de dados necessários a uma aplicação ocupa mais espaço que a memória primária permite ou quando várias aplicações compartilham a memória, há a necessidade de fazer a busca dos dados em outros dispositivos, como a memória secundária. O acesso a esta memória é muito mais caro, por isso então *SMV*(Sistemas de Memória Virtual) são criados. A ideia é abstrair para os programas memória suficiente para os seus dados, por meio da utilização da memória secundária como uma "extensão" da memória primária, criando a ilusão de um espaço de memória primária maior do que o que realmente existe. 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 estar cheia, são aplicadas políticas de reposição de página. Neste trabalho será então simulado um SMV aplicando as seguintes políticas de reposição de página:

- *FIFO*(First In First Out) - a página residente há mais tempo na memória dá lugar à página solicitada pela aplicação.
- *LRU*(Least Recently Used) – a página acessada há mais tempo dá lugar à página solicitada pela aplicação.
- *LFU*(Least Frequently Used) - a página com menos acessos dá lugar à página solicitada pela aplicação.

Para a simulação será usada a estrutura de dados árvore B para a abstração da memória secundária. No arquivo de entrada serão passados para o programa os valores a serem inseridos na árvore B, valores a serem excluídos da árvore B, valores a serem consultados na árvore B fazendo a simulação do SMV, e valores a serem consultados na árvore B mostrando o caminhamento feito na árvore B para a obtenção destes. No arquivo de saída teremos o número de *page misses,* que são o número de vezes onde a página cujo registro solicitado não estava na memória primária, para cada uma das políticas de substituição de páginas e o caminhamento na árvore B para cada um dos registros solicitados.

Modelagem e Solução Proposta

Para obtermos acesso a qualquer página da árvore B, nesta simulação, é necessário que os nós (páginas) percorridos durante o caminhamento estejam em memória primária. Caso um nó não esteja em memória primária e a memória esteja cheia, políticas de reposição de páginas são utilizadas, sendo elas descritas abaixo. Para a implementação das políticas de reposição de páginas foi utilizada a estrutura de dados fila.

FIFO(First In First Out):

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, este é enfileirado, ou seja, 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).

LRU(Least Recently Used)

Uma das políticas mais usadas. Remove a página menos usada da memória para dar lugar à página solicitada pela aplicação. Cada célula da fila contém uma página. Com a memória inicialmente vazia, ao buscarmos os registros na memória secundária, colocaremos sua página sempre na última posição da fila, até que esta atinja o limite de páginas. Toda vez que um registro existente na memória primária for acessado, ou seja, houver um *hit*, a célula que contém a página do registro será colocada na primeira posição por meio de manipulação de ponteiros da fila. Assim, quando a lista estiver cheia, com esta implementação, garantimos que a primeira célula da lista é sempre a menos recentemente usada, pois uma página trazida da memória secundária para a primária ocupará a última posição na fila e as páginas acessadas em memória primária são sempre colocadas no final da lista. A complexidade desta política também é O(1), visto que não é necessário em nenhum momento percorrer a lista.

LFU(Least Frequently Used)

Leva em consideração o número de acessos que a página obteve e então remove a página que teve menos acessos. Um problema desta política é que uma página recentemente trazida da memória pode ser retirada por ter baixo número de acessos. Para isso cada página guarda o número de acessos que teve. Então toda vez que houver um *hit,* o número de acessos da página é incrementado. Assim, para a remoção de páginas da memória primária a fila é percorrida para descobrir qual foi menos acessada. A página com menos acessos então, por meio de manipulação de ponteiros, passa a ocupar a primeira posição da lista, sendo removida e assim abrindo espaço para que a página da memória secundária seja colocada em memória primária. Como é necessário percorrer a lista buscando a página com menor número de acessos, a complexidade é O(n), onde *n* é o tamanho da memória. Como podem existir páginas com mesmo número de acessos, o critério de desempate utilizado foi a página residente em memória por mais tempo é a *LFU.*

Implementação

Para este trabalho, a estrutura de dados utilizada para a simulação da memória secundária foi a ÁrvoreB. Sua implementação foi retirada de http://www.dcc.ufmg.br/algoritmos/cap6/codigo/c/6.3a6.9-arvore-b.c .

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.

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, é usado #define M 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.

Funções

Para a implementação do trabalho foram usadas as seguintes funções:

void InsereNaPagina(TipoApontador, TipoRegistro, TipoApontador);

void Ins(TipoRegistro, TipoApontador, short *, TipoRegistro *, TipoApontador *, int);

void Insere(TipoRegistro, TipoApontador *, int);

As funções acima são responsáveis por fazer a inserção de um registro na árvore obedecendo às regras para que seja considerada uma árvore B.

void Reconstitui(TipoApontador, TipoApontador, int, short *, int);

void Antecessor(TipoApontador, int, TipoApontador, short *, int);

void Ret(TipoChave, TipoApontador *, short *, int);

void Retira(TipoChave, TipoApontador *, int);

O conjunto de funções acima descrito é responsável por fazer a retirada de um determinado registro da árvore B. Assim como na inserção, as regras para que se tenha uma árvore B devem ser obedecidas.

void FFVazia(TipoFila *);

int Enfileira(TipoPagina, TipoFila *, int, int *);

void Desenfileira(TipoFila *, int *);

int Vazia(TipoFila);

Estas funções fazem parte do tipo abstrato de dados fila. *FFVazia()* cria uma fila sem registros. *Enfileira()* coloca o registro no final da fila. *Desenfileira()* retira o registro do início da fila. *Vazia()* verifica se a fila está vazia.

int EstaNaMemoria(TipoFila, int, ApontadorFila *);

Verifica se o registro buscado na árvore B já está em memória. Caso sim retorna 1 e caso não retorna 0;

void ColocaNaMemoria(TipoPagina, TipoFila *, int, int *, int);

Coloca o registro buscado na árvore B em memória.

void HitLRU(TipoFila *, ApontadorFila, int, int *);

Considera o *hit* quando a política escolhida para reposição de páginas é o *LRU.* Faz a manipulação dos ponteiros da fila para que a página de memória que foi acessada fique sempre na primeir posição da fila.

void HitLFU(ApontadorFila);

void CorrigeLFU(TipoFila *, int *, int);

A função *HitLFU* considera o *hit* quando a política escolhida para reposição de páginas é o *LFU.* Nesta função o número de acessos da página é incrementado. A função *CorrigeLFU()* então percorre a lista procurando a página que tem menor número de acessos, e caso existam outras com mesmo número de acessos igual, a com maior tempo em memória é a *LFU.* Por meio de manipulação de ponteiros a página é colocada na primeira posição da fila.

int Politicas(TipoApontador, int, int, int *, int);

Baseado na política de reposição escolhida, faz a busca na memória dos registros solicitados pelo arquivo de entrada, aplicando as políticas de reposição. Esta função retorna então o número de *misses* que ocorreram durante as buscas.

void Caminhamento(TipoRegistro *, TipoApontador, FILE *);

Feito baseado no algoritmo de pesquisa implementado pela árvore B. Imprime no arquivo de saída todos os registros das páginas por onde passou até chegar ao registro procurado.

Antes do término da execução, é dado free() em todas as estruturas de dados que foram alocadas dinamicamente pelo programa.

Análises

Complexidade temporal

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.

As operações do *SMV(Sistemas de Memória Virtual)* podem ser descritas basicamente da seguinte maneira:

Registro é solicitado pela aplicação:

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.

Analisando então o programa, sua complexidade no **pior** caso será:

$O(n^2(\log(r)))$

Onde **n** é o tamanho da memória e **r** é o número de páginas da árvore B.

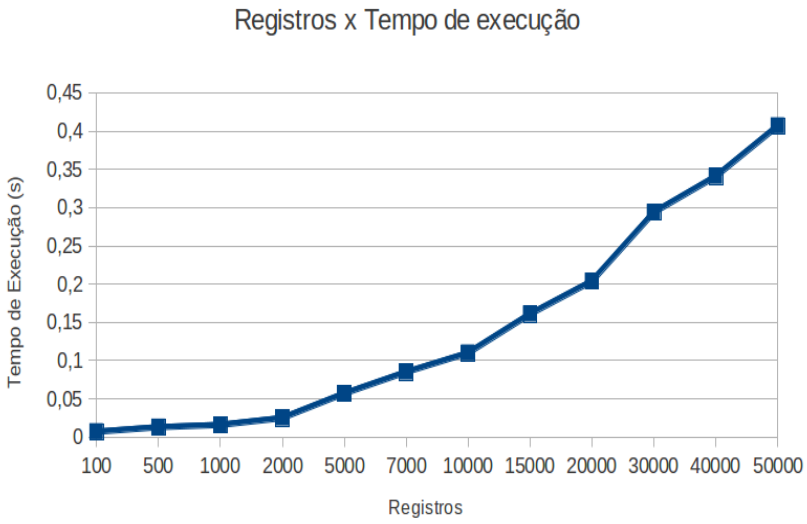
As políticas *FIFO* e *LRU* têm complexidade O(1), porém a política *LFU* tem complexidade O(n), pois percorre a fila em busca da página com menos acessos, então *LFU* domina assintoticamente as outras políticas. Para cada página do caminhamento na árvore, verificamos se esta se encontra em memória, percorrendo a memória para cada página da árvore, então O(n(log(r)). Caso a página não esteja na memória e a memória esteja cheia, para a política *LFU* a memória é percorrida novamente, O(n) * O(n(log(r)), gerando a complexidade acima citada. Porém é necessário que haja um caso extremo para gerar tal complexidade, pois é necessário que nenhuma página do caminhamento esteja em memória e que a memória esteja sempre cheia.

Tempo de execução

Para os experimentos foi considerado: Haver uma única instância, 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

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, obtivemos, para o tempo de execução por entrada, os resultados mostrados no gráfico à direita.



Misses e tamanho de memória

Afim de contabilizar o número de *misses* para cada política, a memória inicialmente teve 500 páginas, sendo variada na proporção de 25%, 50%, e 75%. O número de registros foi mantido fixo com a entrada contendo 10 mil registros, sendo estes aleatórios. Os seguintes gráficos foram então gerados.

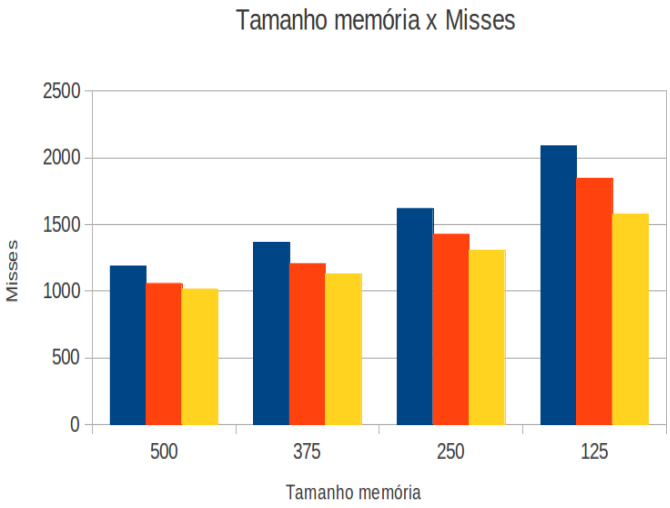


Gráfico 1

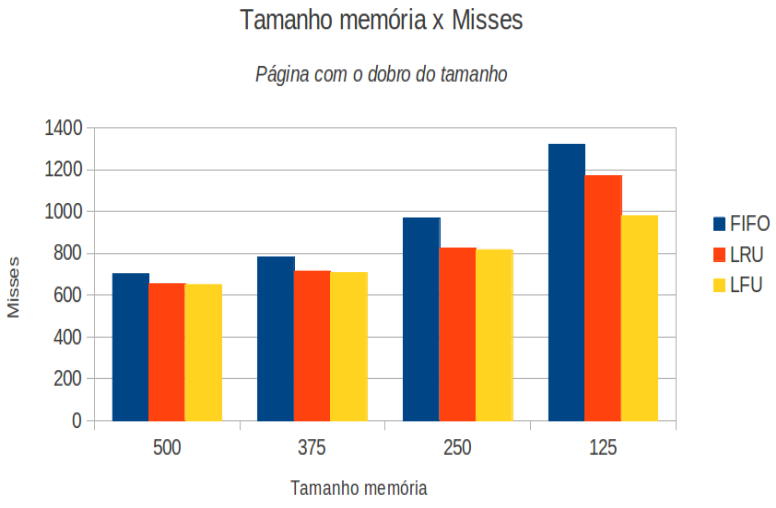


Gráfico 2

O *Gráfico 1* é resultado da simulação para memória com páginas que suportam de 4 a 8 registros. Já para gerar o *Gráfico 2* as páginas tinham o dobro do tamanho, comportando de 8 a 16 registros.

Através do *Gráfico 1* pode-se perceber o aumento do número de *misses* quando a memória diminui. A diminuição da memória implica em menos registros, assim a chance de o registro requerido pela aplicação não estar na memória aumenta, causando maior número de buscas em memória secundária, ou seja, maior quantidade de *misses.*

O *Gráfico 2* nos mostra que há queda considerável no número de misses quando passamos a ter páginas que comportam mais registros. Porém há o *trade-off* entre ter páginas maiores que levando para a memória mais registros, podendo trazer dados desnecessário à aplicação, e ter páginas com tamanho pequeno, trazendo os dados necessário, mas correndo o risco de haver maior número de *misses.* Cabe então a quem vai implementar o *SMV* decidir o tamanho de página que melhor atende aos seus requisitos.

Por ambos os gráficos percebe-se que a política de reposição de páginas mais eficiente para a simulação foi a *LFU(Least Frequently Used*), fazendo menor número de buscas em memória secundária. Em segundo lugar vem a *LRU(Least Frequently Used)* que para tamanhos de memória consideráveis é quase tão eficiente quanto a *LFU* e por último a *FIFO(First In First Out)* que para todos os casos dos gráficos se mostrou a pior política de reposição.

Máquina utilizada

A máquina utilizada no desenvolvimento e testes dos algoritmos foi um notebook Samsung RF511, com sistema operacional Linux **64 bits,** em ambiente virtual, com 2,9Gb de memória RAM DDR3 e processador Intel Core i7, 2.2GHz. A implementação foi feita utilizando a IDE Code::Blocks 12.11.

Conclusão

Através deste trabalho pudemos entender na prática o funcionamento dos Sistemas de Memória Virtual, que gerenciam a memória, dando a ilusão de espaço de memória maior do que o existente. Aplicando suas políticas de reposição de páginas pudemos perceber como esta prática é eficiente ao disponibilizar dados para as aplicações. Apesar de o começo do trabalho ter sido um pouco conturbado devido à dificuldade inicial em entender o que estava sendo proposto e a constante alteração na especificação, creio o objetivo com este ter sido atingido, visto que para todos os testes feitos os resultados obtidos pelo programa implementado estavam certos. Além disto foi grande aprendizado com relação ao assunto abordado.