UNIVERSIDADE TUIUTI DO PARANÁ

Luiz Fernando Nogueira Capitulino

DESEMPENHO DA ÁRVORE DE AFUNILAMENTO NO ARMAZENAMENTO DAS ÁREAS DE MEMÓRIA VIRTUAL DO KERNEL LINUX

CURITIBA
2009

Luiz Fernando Nogueira Capitulino

DESEMPENHO DA ÁRVORE DE AFUNILAMENTO NO ARMAZENAMENTO DAS ÁREAS DE MEMÓRIA VIRTUAL DO KERNEL LINUX

Trabalho de Graduação apresentado como requisito parcial avaliativo para obtenção do título de Tecnólogo em Análise e Desenvolvimento de Sistemas da Faculdade de Ciências exatas e de Tecnologia da Universidade Tuiuti do Paraná.

Orientador: Islenho de Almeida.

CURITIBA
2009

DEDICATÓRIA

Este trabalho é dedicado à minha esposa, Sheyla Maria Branco do Vale, e a meu filho, Luiz Gabriel do Vale Capitulino, pois sem o apoio e compreensão de ambos, eu jamais teria retornado à Universidade.

AGRADECIMENTOS

Várias pessoas ajudaram durante a execução deste projeto, o mínimo que posso fazer para demonstrar minha gratidão é mencioná-las neste espaço. Agradeço a: Eduardo Habkost, por ter revisado a primeira versão do trabalho e por vários bate-bapos informais sobre o assunto; Eugeni Dodonov, pela imensa ajuda na interpretação dos resultados; Felipe Arruda, por responder várias questões referentes ao *OpenOffice*; Gustavo Niemeyer, pelo modelo pronto do *OpenOffice* com as normas da UTP; Islenho de Almeida, por ter aceitado orientar um trabalho muito "diferente" para o curso de Tecnologia; Wanderlei Cavassin, por permitir que dedicasse parte do meu horário de trabalho na Mandriva/Conectiva ao projeto.

Finalmente, e mais importante, agradeço à minha esposa, Sheyla Maria Branco do Vale, por cuidar de nosso filho enquanto eu escrevia esta monografia.

EPÍGRAFE

The best way to learn what an algorithm is all about is to try it.
(Dolnald E. Knuth)

RESUMO

Avaliação do desempenho de árvores de afunilamento no gerenciamento das Áreas de Memória Virtual dos processos no *kernel* Linux. Tal estudo é importante porque a estrutura de dados citada ainda não foi experimentada no *kernel* Linux, desse modo seus possíveis benefícios são desconhecidos. O objetivo principal é comparar o desempenho da árvore de afunilamento com a árvore rubro-negra atualmente usada pelo *kernel*. Para isso, o subsistema de gerenciamento de memória do Linux 2.6.29.2 foi alterado para usar uma árvore de afunilamento no gerenciamento das áreas de memória virtual dos processos. Foram executados testes de desempenho que simulam ambientes da vida real e seus resultados expostos e discutidos.

Palavras-chave: Estruturas de dados. Árvores de afunilamento. Sistemas operacionais. Gerenciamento de memória. *Kernel* Linux.

LISTA DE FIGURAS

FIGURA 1: ÁRVORE..11
FIGURA 2: ÁRVORE BINÁRIA..13
FIGURA 3: ÁRVORE DE BUSCA BINÁRIA..14
FIGURA 4: ÁRVORE DE BUSCA BINÁRIA DEGENERADA.......................22
FIGURA 5: CASO ZIG: ROTAÇÃO SIMPLES..25
FIGURA 6: CASO ZIG-ZIG: DUAS ROTAÇÕES SIMPLES.........................25
FIGURA 7: CASO ZIG-ZAG: ROTAÇÃO DUPLA..26
FIGURA 8: ESPAÇO DE ENDEREÇAMENTO DE UM PROCESSO.........35
FIGURA 9: RELAÇÃO ENTRE AS ESTRUTURAS MM_STRUCT E VM_AREA_STRUCT..40

LISTA DE QUADROS

QUADRO 1: ALGORITMO DE BUSCA EM ÁRVORE DE BUSCA............................15
QUADRO 2: ALGORITMO DE INSERÇÃO EM ÁRVORE DE BUSCA BINÁRIA......17
QUADRO 3: ALGORITMO PARA RETORNAR O SUCESSOR DE UM NÓ.............19
QUADRO 4: ALGORITMO DE BAIXO NÍVEL PARA REMOVER UM NÓ DA ÁRVORE DE BUSCA BINÁRIA..19
QUADRO 5: ALGORITMO PARA REMOVER UM NÓ DE UMA ÁRVORE DE BUSCA 20
QUADRO 6: ALGORITMO PARA ATUALIZAR O NÓ PAI DE UM NÓ SENDO REMOVIDO DA ÁRVORE DE BUSCA BINÁRIA...21
QUADRO 7: ALGORITMO PARA AFUNILAMENTO TOP-DOWN............................27
QUADRO 8: ALGORITMO DE BUSCA PARA ÁRVORE DE AFUNILAMENTO........29
QUADRO 9: ALGORITMO DE INSERÇÃO PARA ÁRVORE DE AFUNILAMENTO. 30
QUADRO 10: ALGORITMO DE REMOÇÃO PARA ÁRVORE DE AFUNILAMENTO... 31
QUADRO 11: MEMBROS RELEVANTES DA ESTRUTURA MM_STRUCT.............36
QUADRO 12: MEMBROS DA ESTRUTURA VM_AREA_STRUCT...........................38
QUADRO 13: ASSINATURA DAS FUNÇÕES DE BUSCA DE VMAS......................41
QUADRO 14: MÁQUINA LAPTOP...45
QUADRO 15: MÁQUINA SERVIDOR..45
QUADRO 16: PRINCIPAIS FERRAMENTAS UTILIZADAS NO PROJETO..............46
QUADRO 17: FUNÇÕES E MACROS DO KERNEL LINUX MODIFICADAS............47
QUADRO 18: NOMES ADOTADOS PARA OS KERNELS UTILIZADOS..................49
QUADRO 19: TAMANHO DAS ESTRUTURAS EM BYTES......................................50
QUADRO 20: DIMINUIÇÃO DAS ESTRUTURAS EM PORCENTAGEM..................50
QUADRO 21: TEMPO DE CONSTRUÇÃO DO KERNEL LINUX EM MODO KERNEL (SEGUNDOS)..52
QUADRO 22: SYSBENCH MÁQUINA LAPTOP: 1 A 8 THREADS...........................55
QUADRO 23: RESULTADOS DO TESTE HACKBENCH MÁQUINA SERVIDOR....58
QUADRO 24: RESULTADOS LMBENCH PARA CRIAÇÃO DE PROCESSOS (MÁQUINA SERVIDOR)...59

SUMÁRIO

1 INTRODUÇÃO......9
2 ÁRVORES......11
2.1 ÁRVORE BINÁRIA......13
2.2 ÁRVORE DE BUSCA BINÁRIA......14
2.2.1 Busca......15
2.2.2 Inserção......16
2.2.3 Remoção......18
2.3 BALANCEAMENTO......22
2.4 ÁRVORE DE AFUNILAMENTO......24
2.4.1 Afunilamento top-down......26
2.4.2 Busca......28
2.4.3 Inserção......29
2.4.4 Remoção......30
3 LINUX......32
3.1 ESPAÇO DE ENDEREÇAMENTO DOS PROCESSOS......34
3.1.1 Descritor de memória......36
3.1.2 Áreas de Memória Virtual......37
3.1.3 Busca de VMAs......40
4 METODOLOGIA DE DESENVOLVIMENTO......42
4.1 TESTES DE DESEMPENHO......43
4.1.1 Tempo de construção do kernel Linux......43
4.1.2 Hackbench......43
4.1.3 LMbench......44
4.1.4 Sysbench......44
4.2 INFRAESTRUTURA DE HARDWARE......45
4.3 PROGRAMAS E FERRAMENTAS......46
5 IMPLEMENTAÇÃO......47
6 RESULTADOS......49
6.1 TAMANHO DAS ESTRUTURAS......50
6.2 TEMPO DE CONSTRUÇÃO DO KERNEL LINUX......52
6.3 SYSBENCH......53
6.3.1 De 1 a 8 threads......53
6.3.2 De 16 a 100 threads......56
6.3.3 De 100 a 800 threads......57
6.4 HACKBENCH......58
6.5 LMBENCH......59
7 CONCLUSÃO......60
8 TRABALHOS FUTUROS......61
REFERÊNCIAS......62

1 INTRODUÇÃO

Uma das principais tarefas de um sistema operacional é gerenciar as áreas de memória que cada processo pode acessar, bem como as permissões de acesso.

No *kernel* Linux essas informações são armazenadas em uma estrutura de dados chamada Área de Memória Virtual (*Virtual Memory Area*, ou VMA em Inglês). Um único processo pode ter algumas dezenas ou até milhares de VMAs e elas são criadas, destruídas e pesquisadas em diversas operações fundamentais como por exemplo, alocação de memória e criação de novos processos.

Para o funcionamento eficaz do sistema é de fundamental importância que a estrutura de dados usada para o armazenamento das VMAs tenha um bom desempenho. Atualmente o *kernel* Linux utiliza uma árvore rubro-negra para esse fim (LOVE, 2005, p. 259).

Embora a árvore rubro-negra implementada no *kernel* Linux tenha desempenho satisfatório, experimentos realizados por Ptaff (2004) sugerem que sob determinadas condições árvores de afunilamento (são árvores de busca binária com a propriedade de mover o elemento recentemente acessado para a raiz da árvore) podem ter desempenho superior a árvores rubro-negras no armazenamento das VMAs.

No entanto, os experimentos de Ptaff, embora relevantes no contexto de sua pesquisa, limitaram-se a simulações em espaço de usuário de algumas operações específicas as quais não representam com fidelidade o ambiente e as condições nas quais o *kernel* Linux manipula as VMAs.

Por exemplo, não foram feitas simulações com acesso concorrente na árvore. Além disso, alguns cenários como falha de página, são extremamente difíceis de simular em espaço de usuário.

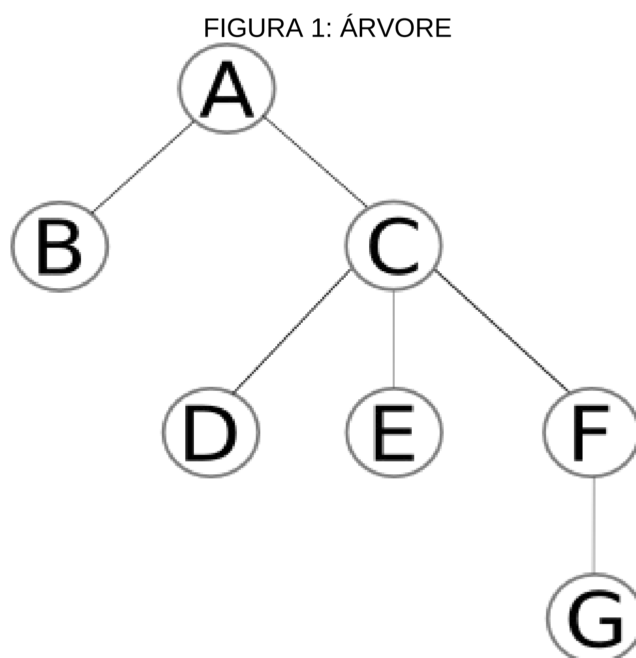
O objetivo deste trabalho é modificar o *kernel* Linux 2.6.29.2 para utilizar uma árvore de afunilamento para o armazenamento de VMAs e assim comparar o seu desempenho com a árvore rubro-negra atual. Como este experimento não é uma simulação, espera-se que a árvore de afunilamento possa ser testada e avaliada em casos de uso reais sob condições abrangentes.

Este trabalho está organizado da seguinte maneira. O capítulo 2 contém toda a teoria sobre árvores utilizada no trabalho, em especial algoritmos para árvore de busca binária e árvore de afunilamento são explicados em detalhes. O capítulo 3 descreve como o *kernel* Linux gerencia o espaço de endereçamento dos processos, assim como detalhes importantes sobre o código de VMA. O capítulo 4 apresenta a metodologia de desenvolvimento adotada. O capítulo 5 discute alguns detalhes sobre a implementação dos algoritmos das árvores de busca binária e afunilamento no *kernel* Linux. O capítulo 6 expõe todos os resultados dos testes de desempenho executados. Finalmente, o capítulo 7 contêm a conclusão da pesquisa e o capítulo 8 oferece algumas sugestões para projetos futuros.

2 ÁRVORES

Árvores são estruturas de dados que simulam o formato das árvores da natureza no computador, elas são utilizadas para o armazenamento de dados que possuem uma relação de hierarquia (ou “ramificação”) entre si. O que é simulado na verdade são os galhos da árvore, como exemplificado na figura 1.

FIGURA 1: ÁRVORE



FONTE: KNUTH, 1997, P. 309

Os dados armazenados em uma árvore são chamados elementos ou nós da árvore, na figura 1 visualiza-se uma árvore cujos nós são letras do alfabeto.

A terminologia usada para descrever estrutura de dados do tipo árvore é a mesma usada para descrever as árvores da natureza, sendo que alguns termos são específicos de árvores genealógicas. Os termos mais importantes e relevantes para este trabalho são explicados a seguir (KNUTH, 1997; LOUDON, 1999):

- O nó no topo da árvore é chamado nó raiz. Na figura 1 o nó raiz é o nó cujo conteúdo é a letra A
- Cada nó da árvore pode ter um ou mais filhos. Na figura 1 B e C são filhos de A; D, E e F são filhos de C; e G é filho do nó F
- Todo nó, com exceção do nó raiz, possui um pai. Na figura 1 o nó A é pai de B e C; C é pai de D, E e F; e F é pai de G
- Os termos ancestral e descendente também costumam ser utilizados. Na figura 1 os nós D, E, F e G são descendentes de C e A é ancestral de F
- Nós folhas ou terminais são nós que não possuem filhos. Na figura 1 os nós B, D, E e G são folhas

Além da terminologia básica existem também algumas definições que devem ser observadas (KNUTH, 1997; LOUDON, 1999):

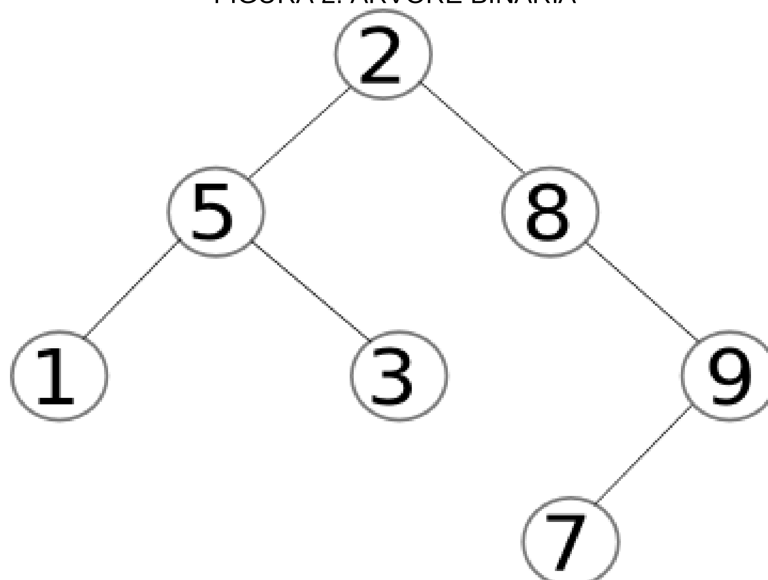
- O número máximo de filhos que um nó de uma árvore pode ter chama-se fator de ramificação
- A altura de um nó da árvore é a distância entre esse nó e o seu descendente mais afastado. Por exemplo, na figura 1 a altura do nó C é dois, pois o seu descendente mais afastado é o nó G
- A altura do nó raiz é a altura da árvore
- Todo nó de uma árvore é a raiz de alguma sub-árvore, a qual está contida na árvore original

A definição de sub-árvore diz que uma árvore é formada por outras árvores. Isso significa que árvores são estruturas de dados recursivas e grande parte dos algoritmos que trabalham com árvores podem ser descritos utilizando-se recursão.

2.1 ÁRVORE BINÁRIA

Uma árvore binária é uma estrutura de dados do tipo árvore cujos nós podem ter no máximo dois filhos, isto é, o fator de ramificação da árvore é igual a dois. Os nós filhos, quando presentes, são chamados de sub-árvore da esquerda e sub-árvore da direita. A figura 2 mostra uma árvore binária que armazena números inteiros. É importante observar que os nós 5 e 8 são, respectivamente, as raízes das sub-árvores da esquerda e direta do nó 2.

FIGURA 2: ÁRVORE BINÁRIA



Existem vários algoritmos que trabalham com árvores, uma classe especial desses algoritmos são os usados para visitar os nós da árvore ou mais precisamente para “percorrer” a árvore.

De acordo com Knuth (1997, p. 319) os três algoritmos mais importantes para percorrer árvores binárias são: pré-ordem, em ordem (ou ordem simétrica) e pós-ordem.

2.2 ÁRVORE DE BUSCA BINÁRIA

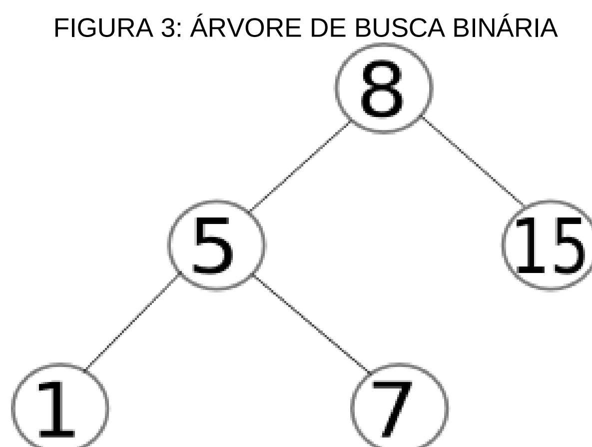
Uma árvore de busca binária é uma árvore binária especificamente organizada para se fazer buscas. Todos os nós em uma árvore de busca binária são armazenados de uma maneira que satisfaça a seguinte propriedade (CORMEN, 2001, p. 254):

> “Seja x um nó da árvore de busca binária. Se y é um nó na sub-árvore esquerda de x, então chave[y] ≤ chave[x]. Se y é um nó na sub-árvore direita de x, então chave[x] ≤ chave[y]”.

A convenção de pseudo-código utilizada neste trabalho será a mesma que descrita em (CORMEN, 2001, p. 19) com pequenas alterações, as quais devem ser fáceis de deduzir. A definição mais importante da convenção é: seja x um objeto em memória, se chave é membro de x então se deve escrever “chave[x]” para acessar esse membro.

A figura 3 mostra uma árvore de busca binária.

FIGURA 3: ÁRVORE DE BUSCA BINÁRIA



Existem diversas operações que uma árvore de busca binária pode executar, no entanto para este trabalho somente as operações de busca, inserção e remoção serão implementadas e brevemente discutidas.

2.2.1 Busca

Buscar um nó em uma árvore de busca binária é uma das operações mais comuns e simples de ser implementada, o algoritmo para essa operação é apresentado no quadro 1.

QUADRO 1: ALGORITMO DE BUSCA EM ÁRVORE DE BUSCA

```
busca(arvore, chave)
    no ← raiz[arvore]
    enquanto no ≠ NULO E chave ≠ chave[no]
        se chave < chave[no]
            no ← esquerda[no]
        senão
            no ← direta[no]
    retorna no
```

FONTE: CORMEN, 2001, P. 257 - ADAPTADO

O algoritmo recebe uma árvore e a chave a ser encontrada. Para cada nó da árvore, começando pelo nó raiz, o algoritmo compara a chave do nó atual com a chave que foi passada por parâmetro. Se elas forem iguais a busca terminou e o nó encontrado é retornado. Se a chave sendo procurada for menor que a chave do nó atual, o algoritmo desce para a sub-árvore da esquerda. Caso contrário, se a chave sendo procurada for maior que a chave do nó atual, o algoritmo desce para a sub-árvore da direita. Se um nó nulo for encontrado, significa que a chave sendo procurada não existe na árvore, então o algoritmo retorna nulo.

Um detalhe importante a ser observado é que o algoritmo apresentado poderia ser escrito utilizando-se recursividade, mas de acordo com Cormen (2001, p. 257) a versão apresentada é mais eficiente na maioria dos computadores. Então não será utilizado recursão para nenhum dos algoritmos deste trabalho.

2.2.2 Inserção

Inserir um novo nó na árvore consiste em percorrê-la da mesma maneira que é feito no algoritmo de busca (quadro 1), a diferença é que se torna necessário manter uma referência para o nó pai de cada nó visitado. Se durante a busca a chave a ser inserida for encontrada, então essa chave já existe e não é necessário inserir novamente. Caso contrário, deve-se inserir a nova chave como sub-árvore da esquerda ou sub-árvore da direita do nó pai encontrado.

O quadro 2 mostra o pseudo-código para esse algoritmo.

QUADRO 2: ALGORITMO DE INSERÇÃO EM ÁRVORE DE BUSCA BINÁRIA

```
insere(arvore, chave)
        no ← raiz[arvore]
        nopai ← NULO
        enquanto no ≠ NULO
                if chave = chave[no]
                        retorna falso
                nopai ← no
                if chave < chave[no]
                        no ← esquerda[no]
                senão
                        no ← direita[no]
        no ← aloca(chave, nopai)
        if nopai = NULO
                /* Árvore está vazia */
                raiz[arvore] ← no
        senão se chave[no] < chave[nopai]
                esquerda[nopai] ← no
        senão
                direita[nopai] ← no
        tamanho[arvore] ← tamanho[arvore] + 1
        retorna verdadeiro
```

FONTE: CORMEN, 2001, P. 261 - ADAPTADO

O algoritmo do quadro 2 recebe uma árvore e a chave a ser inserida. Primeiro ele percorre a árvore até que o nó pai do nó a ser inserido seja encontrado, se durante a busca um nó com a mesma chave existir a inserção não é feita. Considerando que o nó pai foi encontrado, a função aloca() é chamada. Essa função aloca memória para o novo nó e inicializa os membros chave e nopai com os valores de seus respectivos argumentos. Não será apresentado o pseudo-código para aloca(), pois alocação de memória dinâmica é dependente de linguagem de programação.

Armazenar a referência para o nó pai não é necessário. Porém é conveniente, pois o algoritmo de remoção (apresentado mais adiante) se torna ligeiramente mais simples: ele pode usar o algoritmo do quadro 1 para buscar o nó a ser removido.

O último passo executado pelo algoritmo do quadro 2 consiste em realmente inserir o novo nó na árvore. Há três casos de inserção a serem considerados. Seja x o novo nó sendo inserido:

1. Se a árvore está vazia, x deve ser a raiz da árvore
2. Caso contrário existe um nó pai, se chave[x] é menor que a chave do nó pai, x deve ser a raiz da sub-árvore da esquerda do nó pai
3. Caso contrário chave[x] é maior que a chave do nó pai, então x deve ser a raiz da sub-árvore da direita do nó pai

2.2.3 Remoção

Remover um nó da árvore é a operação mais complexa que será apresentada para árvores de busca binária. Existem três casos a serem considerados. Seja x um nó da árvore sendo removido (PTAFF, 2002, p. 39):

1. Se x não tem o filho da direita, deve-se substituir a referência que leva a x por esquerda[x]
2. Senão, se o filho da direita de x não tem filho da esquerda, deve-se substituir x por direita[x] e anexar a sub-árvore da esquerda de x a sub-árvore da esquerda do novo nó
3. Senão, o filho da direita de x tem o filho da esquerda, então se deve substituir x pelo seu sucessor em ordem

Seja y um nó da árvore de busca binária, o sucessor de y é definido como sendo “o nó com a menor chave maior que chave[y]” (CORMEN, 2001, p. 258). Devido à estrutura da árvore de busca binária, é possível encontrar o sucessor de qualquer nó da árvore apenas percorrendo a sub-árvore da esquerda da sub-árvore

da direita do nó dado.

O algoritmo do quadro 3 retorna o sucessor de um nó da árvore de busca binária. É importante notar que esse algoritmo deve receber a sub-árvore da direita do nó.

QUADRO 3: ALGORITMO PARA RETORNAR O SUCESSOR DE UM NÓ

```
sucessor(sub_dir)
        no ← sub_dir
        enquanto esquerda[no] ≠ NULO
                no ← esquerda[no]
        retorna no
```

FONTE: CORMEN, 2001, P. 259 - ADAPTADO

O algoritmo do quadro 4 é apresentado a seguir e implementa todos os casos de remoção discutidos no início desta seção.

QUADRO 4: ALGORITMO DE BAIXO NÍVEL PARA REMOVER UM NÓ DA ÁRVORE DE BUSCA BINÁRIA

```
__remove(arvore, no)
        se direita[no] = NULO
                /* caso 1 */
                atualiza_nopai(arvore, no, esquerda[no])
        senão
                r ← direita[no]
                se esquerda[r] = NULO
                        /* caso 2 */
                        esquerda[r] ← esquerda[no]
                        se esquerda[no] ≠ NULO
                                pai[esquerda[no]] ← r
                        atualiza_nopai(arvore, no, r)
                senão
                        /* caso 3 */
                        s ← sucessor(r)
                        __remove(arvore, s)
                        esquerda[s] ← esquerda[no]
                        se esquerda[s] ≠ NULO
                                pai[esquerda[s]] ← s
                        direita[s] ← direita[no]
                        se direita[s] ≠ NULO
                                pai[direita[s]] ← s
                        atualiza_nopai(arvore, no, s)
```

FONTE: PTAFF, 2002, P. 40. - ADAPTADO

Os dois caracteres de sub-linha no início do nome do algoritmo é uma notação própria deste trabalho e indica que o algoritmo em questão é de baixo nível. Algoritmos de baixo nível necessitam que alguma preparação seja feita antes de serem chamados, ou que algum trabalho de pós processamento seja feito quando eles retornam. Desse modo, algoritmos de baixo nível normalmente não são chamados diretamente pela aplicação e são implementados como privados em linguagens orientadas a objetos.

O algoritmo do quadro 4 recebe a árvore e o nó a ser removido, então neste caso é necessário buscar o nó antes de chamar o algoritmo. Também é necessário liberar o espaço em memória ocupado pelo nó assim que este é removido.

O algoritmo do quadro 5 implementa os procedimentos que devem ser feitos antes e depois de remover um nó, este é o algoritmo que deve ser chamado diretamente pela aplicação.

QUADRO 5: ALGORITMO PARA REMOVER UM NÓ DE UMA ÁRVORE DE BUSCA

```
remove(arvore, chave)
        no ← busca(arvore, chave)
        se no = NULO
                retorna falso
        __remove(arvore, no)
        libera(no)
        tamanho[arvore] ← tamanho[arvore] - 1
        retorna verdadeiro
```

Para buscar o nó a ser removido da árvore o algoritmo do quadro 5 utiliza o algoritmo de busca do quadro 1. Então o algoritmo de baixo nível é chamado para remover o nó e assim que isso é feito o espaço ocupado pelo nó é liberado pela função libera() (o pseudo-código desta função não será apresentado, pois ela é dependente de linguagem de programação).

O último algoritmo necessário para a remoção de um nó é utilizado pelo algoritmo do quadro 4 e apresentando no quadro 6. Este atualiza o membro pai de um nó da árvore sempre que há uma remoção.

QUADRO 6: ALGORITMO PARA ATUALIZAR O NÓ PAI DE UM NÓ SENDO REMOVIDO DA ÁRVORE DE BUSCA BINÁRIA

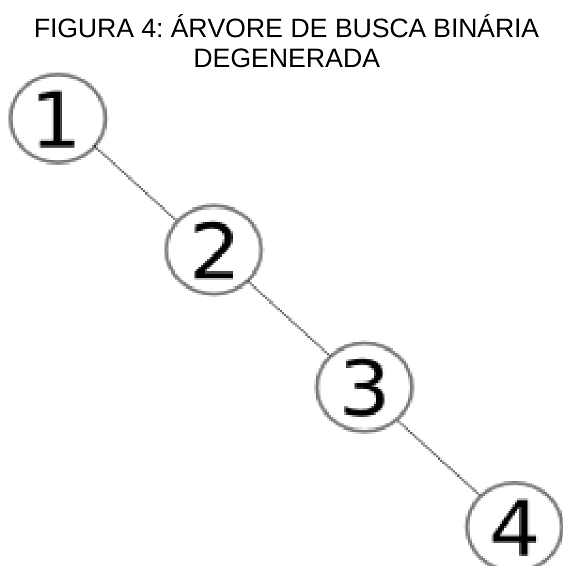
```
atualiza_nopai(arvore, no, novo_filho)
        nopai ← pai[no]
        se nopai ≠ NULO
                se esquerda[nopai] = no
                        esquerda[nopai] ← novo_filho
                senão
                        direita[nopai] ← novo_filho
        senão
                /* nova raíz da árvore */
                raiz[arvore] ← novo_filho
        se novo_filho ≠ NULO
                pai[novo_filho] ← nopai
```

Após discutir os algoritmos de inserção, busca e remoção, será apresentado na seção 2.3 o conceito de balanceamento de árvores.

2.3 BALANCEAMENTO

Existe um problema sério com a árvore de busca binária que não foi discutido na seção anterior. A figura 4 mostra o que acontece quando são inseridos elementos em ordem na árvore, neste exemplo foram inseridas as chaves: 1, 2, 3 e 4.

FIGURA 4: ÁRVORE DE BUSCA BINÁRIA DEGENERADA



Nesse caso a árvore de busca binária se torna uma lista ligada. Uma busca nessa árvore seria como se fazer uma busca linear em uma lista, essa operação pode ser extremamente lenta se existe um grande número de elementos na árvore. Quando a árvore se encontra nesse estado se diz que ela está degenerada ou não balanceada.

De acordo com Knuth (1998, p. 459), uma árvore está balanceada se a altura da sub-árvore da esquerda de cada nó nunca é diferente de +-1 da altura de sua sub-árvore da direita.

É possível que uma árvore de busca binária fique balanceada se a chave de seus nós sejam garantidamente randômicos. Na prática, no entanto, é muito difícil

garantir isso. Uma solução melhor seria um algoritmo que fosse capaz de reordenar a árvore toda vez que esta não esteja balanceada.

De fato, existem vários algoritmos que são capazes de manter uma árvore de busca binária balanceada. Dois dos mais utilizados são o AVL (Adelson-Velsky e E. M. Landis) explicado em detalhes em (KNUTH, 1998) e a árvore rubro-negra, explicada em (CORMEN, 2001).

Apesar de na prática funcionarem bem, esses algoritmos são complexos e não possuem a capacidade de se adaptar ao padrão de busca da aplicação que os utiliza. Por exemplo, supondo-se que uma determinada árvore tenha milhares de nós e que uma aplicação tem o padrão de acesso que acaba muitas vezes buscando algum nó folha, seria desejável que esse nó fosse movido para próximo da raiz, pois desse modo a busca seria executada em menor tempo.

Existe um algoritmo para árvores de busca binária que resolve esse problema, ele é o assunto da próxima seção.

2.4 ÁRVORE DE AFUNILAMENTO

Árvore de afunilamento (ou *splay tree*, em Inglês) é uma árvore de busca binária com a propriedade de mover o último nó acessado para a raiz da árvore, dessa maneira os nós recentemente acessados ficam próximos do topo.

Os criadores da árvore de afunilamento (SLEATOR; TARJAN, 1985) conjecturam que para suficientemente longas seqüências de acesso essa estrutura de dados é tão eficiente quanto qualquer forma de atualização dinâmica de árvores, mesmo uma árvore sinteticamente construída para a exata seqüência de acesso da aplicação.

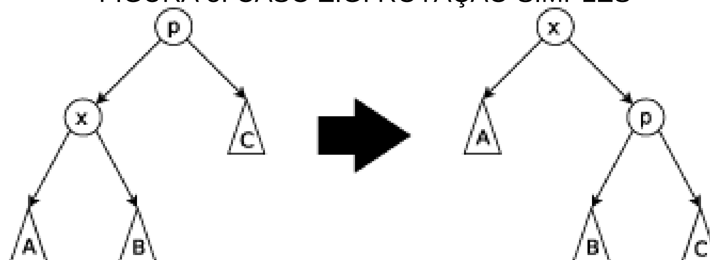
A principal operação em uma árvore de afunilamento chama-se *splaying*, essa operação é responsável por mover um determinado nó da árvore para a sua raiz. Todas as outras operações (busca, inserção e remoção) são implementadas utilizando-se *splaying*. O termo afunilamento será utilizado como tradução de *splaying* neste trabalho.

Outra característica importante da operação de afunilamento é que além de mover o nó recentemente acessado para a raiz ela também diminui em aproximadamente pela metade a profundidade de todos os nós que estão no caminho percorrido pela operação (SLEATOR; TARJAN, 1985, p. 656).

Seja x um nó da árvore recentemente acessado, a operação de afunilamento consiste em aplicar os seguintes casos até que x seja a raiz da árvore (SLEATOR; TARJAN, 1985, p. 655):

- Caso 1 (zig). Se o nó pai de x, ou seja nopai[x], é a raiz da árvore, faça a rotação da referência que une x e nopai[x] como exemplificado na figura 5 (este caso é terminal)

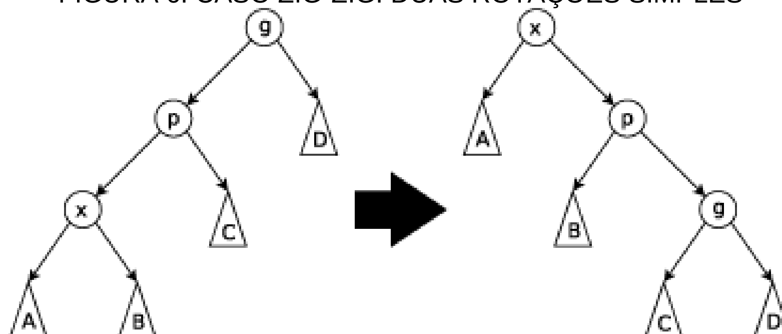
FIGURA 5: CASO ZIG: ROTAÇÃO SIMPLES



FONTE: (SPLAY...)

- Caso 2 (zig-zig). Se nopai[x] não é a raiz e x e nopai[x] são ambos filhos da esquerda ou ambos filhos da direita, faça a rotação da referência que une nopai[x] e o nó avô de x, e então faça a rotação da referência que une x e nopai[x] como exemplificado na figura 6

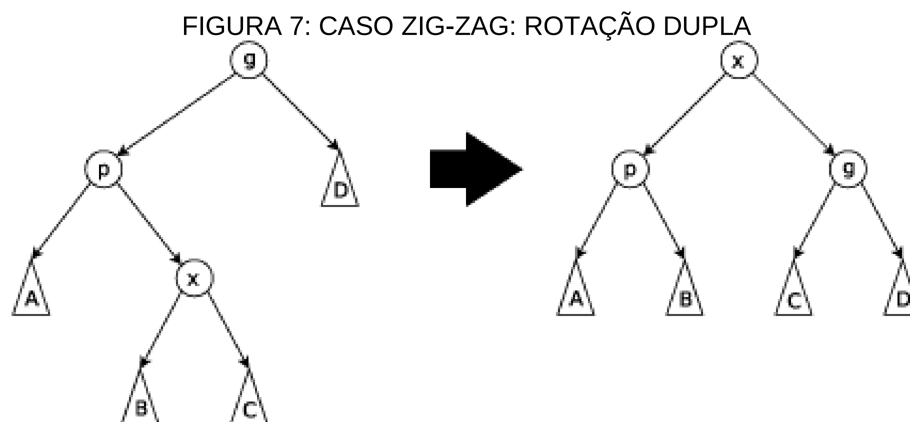
FIGURA 6: CASO ZIG-ZIG: DUAS ROTAÇÕES SIMPLES



FONTE: (SPLAY...)

- Caso 3 (zig-zag). Se nopai[x] não é a raiz e x é um filho da esquerda e nopai[x] um filho da direita, ou vice-versa, faça a rotação da referência que une x com nopai[x] e então faça a rotação da referência que une x com o novo nopai[x], como exemplificado na figura 7

FIGURA 7: CASO ZIG-ZAG: ROTAÇÃO DUPLA



FONTE: (SPLAY...)

2.4.1 Afunilamento *top-down*

Sleator e Tarjan apresentam dois algoritmos para a operação de afunilamento. O primeiro é fazer uma busca na árvore como apresentado no quadro 1, então se aplica a operação de afunilamento no caminho reverso utilizado pela busca até que o nó encontrado seja a raiz da árvore. Essa abordagem é chamada de *bottom-up* (SLEATOR; TARJAN, 1985, p. 665), pois o afunilamento é feito “de baixo para cima”.

Uma outra abordagem, chamada *top-down*, consiste em aplicar os casos de afunilamento ao mesmo tempo em que a árvore é percorrida. De acordo com os autores a abordagem *top-down* é mais eficiente (SLEATOR; TARJAN, 1985, p. 670), sendo assim esta será a abordagem utilizada neste trabalho.

É importante notar que em ambos os casos se o nó sendo procurado não existe na árvore a operação de afunilamento move o último nó acessado durante a busca para a raiz.

O algoritmo para afunilamento *top-down* é apresentado no quadro 7.

QUADRO 7: ALGORITMO PARA AFUNILAMENTO TOP-DOWN

FONTE: SLEATOR; TARJAN, 1985, 669 - ADAPTADO

```
afunilamento_topdown(no, chave)
    direita[no_nulo] ← esquerda[no_nulo] ← NULO
    l ← r ← no_nulo
    enquanto chave ≠ chave[no]
        se chave < chave[no]
            se esquerda[no] = NULO
                pare
            if chave < chave[esquerda[no]]
                /* rotação a direita */
                tmp ← esquerda[no]
                esquerda[no] ← direita[tmp]
                direita[tmp] ← no
                no ← tmp
                se esquerda[no] = NULO
                    pare
            /* ligação a direita */
            esquerda[r] ← no
            r ← no
            no ← esquerda[no]
        senão /* chave > chave[no] */
            se direita[no] = NULO
                pare
            se chave > chave[direita[no]]
                /* rotação a esquerda */
                tmp ← direita[no]
                direita[no] ← esquerda[tmp]
                esquerda[tmp] ← no
                no ← tmp
                se direita[no] = NULO
                    pare
            /* ligação a esquerda */
            direita[l] ← no
            l ← no
            no ← direita[no]
    /* montagem */
    direita[l] ← esquerda[no]
    esquerda[r] ← direita[no]
    esquerda[no] ← direita[no_nulo]
    direita[no] ← esquerda[no_nulo]
    retorna no
```

O algoritmo do quadro 7 recebe o nó raiz atual e a chave a ser encontrada, ele retorna o novo nó raiz. Seja x este nó, se a chave sendo procurada existe na árvore, então chave[x] é igual a chave informada ao algoritmo. Caso contrário, a

chave sendo procurada não existe e x é o último nó encontrado no caminho percorrido pelo algoritmo.

De acordo com Sleator e Tarjan (1985, p. 667) o algoritmo do quadro 7 funciona da seguinte maneira. Durante a busca, a árvore é dividida em três partes: árvore da esquerda, árvore do meio e árvore da direita. A árvore da esquerda e direita contêm todos os nós da árvore original até o momento que são menor que chave ou maior que chave, respectivamente. A árvore do meio consiste na sub-árvore da árvore original, sendo que sua raiz é sempre o nó atual no caminho sendo percorrido. Inicialmente esse nó é a raiz da árvore e as árvores da esquerda e direita estão vazias. Para fazer o afunilamento, o algoritmo percorre a árvore à procura da chave, dois nós por vez, desfazendo ligações ao longo do caminho e sempre adicionando cada sub-árvore removida da árvore original no final da sub-árvore da direita da árvore da esquerda ou na sub-árvore da esquerda da árvore da direita. Caso dois passos sejam tomados na mesma direção (esquerda-esquerda ou direita-direita) uma rotação é feita antes de se desfazer a ligação da árvore original. O algoritmo termina quando a chave sendo procurada é encontrada ou quando um nó folha é alcançado, neste caso o último passo é “remontar” a árvore resultante.

2.4.2 Busca

Como dito no início deste capítulo, em uma árvore de afunilamento as operações de busca, inserção e remoção são implementadas utilizando-se o algoritmo de afunilamento do quadro 7. A operação de busca em particular não necessitaria ser implementada, pois o algoritmo do quadro 7 é, na verdade, uma operação de busca: ele retorna o nó correspondente a chave informada por parâmetro.

Entretanto, é conveniente ter um algoritmo de busca com a mesma semântica que o algoritmo do quadro 1. O pseudo-código para ele é apresentado no quadro 8.

QUADRO 8: ALGORITMO DE BUSCA PARA ÁRVORE DE AFUNILAMENTO

```
busca(arvore, chave)
        se tamanho[arvore] = 0
                retorna NULO
        no ← afunilamento_topdown(raiz[arvore], chave)
        raiz[arvore] ← no
        se chave[no] = chave
                returna no
        returna NULO
```

2.4.3 Inserção

Inserir um novo nó na árvore é uma operação relativamente simples. Se a árvore está vazia o novo nó é inserido diretamente na raiz. Caso contrário a operação de afunilamento é aplicada. Se um nó com a mesma chave existe, então não é necessário inserir novamente. Senão um novo nó é alocado e inserido na árvore conforme as condições explicadas a seguir.

Seja x o nó sendo inserido na árvore de afunilamento e seja y o nó retornado pela operação de afunilamento, y é a nova raiz da árvore. Sendo assim:

1. Se chave[x] é menor que chave[y], o nó em esquerda[y] deve ser inserido em esquerda[x] e y é inserido em direita[x]
2. Senão chave[x] é maior que chave[y], então o nó em direita[y] deve ser inserido em direita[x] e y é inserido em esquerda[x]
3. x se torna a nova raiz da árvore

O algoritmo para inserção é apresentado no quadro 9.

QUADRO 9: ALGORITMO DE INSERÇÃO PARA ÁRVORE DE AFUNILAMENTO

```
insere(arvore, chave)
        se tamanho[arvore] = 0
                raiz[arvore] ← aloca(chave)
                tamanho[arvore] ← 1
                retorna
        no ← afunilamento_topdown(arvore[raiz], chave)
        se chave = chave[no]
                arvore[raiz] ← no
                retorna
        novo ← aloca(chave)
        se chave < chave[no]
                esquerda[novo] ← esquerda[no]
                direita[novo] ← no
                esquerda[no] ← NULO
        senão /* chave > chave[no] */
                direita[novo] ← direita[no]
                esquerda[novo] ← no
                direita[no] ← NULO
        raiz[arvore] ← novo
        tamanho[arvore] ← tamanho[arvore] + 1
```

FONTE: SLEATOR, 1992 - ADAPTADO

2.4.4 Remoção

O algoritmo de remoção de uma árvore de afunilamento é consideravelmente mais simples do que algoritmo de remoção da árvore de busca binária (seção 2.2.3). O primeiro passo consiste em buscar o nó a ser removido. Se o nó não existe nada é feito, caso contrário o nó encontrado se tornou a raiz da árvore, seja x este nó há dois casos a serem considerados:

1. Se a sub-árvore esquerda[x] está vazia, então direita[x] é a nova raiz da árvore e o espaço ocupado por x pode ser liberado
2. Caso contrário, é necessário aplicar a operação de afunilamento na sub-

árvore esquerda[x]. Seja y o nó retornado nessa operação, faz-se direita[y] apontar para direita[x]. Então o nó y se torna a nova raiz da árvore e o espaço ocupado por x pode ser liberado

O algoritmo do quadro 10 implementa todos os casos discutidos nesta seção.

QUADRO 10: ALGORITMO DE REMOÇÃO PARA ÁRVORE DE AFUNILAMENTO

FONTE: SLEATOR, 1992 - ADAPTADO

```
remove(arvore, chave)
        se tamanho[arvore] = 0
                retorna
        no ← afunilamento_topdown(raiz[arvore], chave)
        se chave ≠ chave[no]
                raiz[avore] ← no
                retorna
        se esquerda[no] = NULO
                nova_raiz ← direita[no]
        senão
                nova_raiz ← afunilamento_topdown(esquerda[no], chave)
                direita[nova_raiz] ← direita[no]
        libera(no)
        tamanho[arvore] ← tamanho[arvore] - 1
        raiz[arvore] ← nova_raiz
```

3 LINUX

Em 1969 uma equipe de cientistas dos Laboratórios Bell, liderada por Ken Thompson, criou o sistema operacional Unix. Este sistema, devido a sua elegância, simplicidade e funcionalidades inovadoras, não demorou a se espalhar por centros universitários e empresas de grande porte (UNIX...).

Como o código-fonte do Unix estava disponível, diversos clones deste sistema foram desenvolvidos nos anos que se seguiram a sua criação. Embora o *design* original e principais funcionalidades estavam sempre presentes nos novos sistemas, novas características foram sendo adicionadas ao longo do tempo.

De acordo com Salus (1994, p. 2) o Unix teve uma profunda influência em grande parte dos sistemas operacionais criados nas décadas seguintes incluindo, por exemplo, o Windows NT e os sistemas operacionais da Apple.

Inicialmente desenvolvido por Linus Torvalds em 1991, Linux é um *kernel* de sistema operacional baseado no Unix, mas não é um descendente direto por não ter sido escrito a partir do código-fonte do Unix original (LOVE, 2005, p. 3).

Desde a sua introdução, o *kernel* Linux vem sendo desenvolvido através da Internet por centenas de pessoas de diferentes nacionalidades (LINUX KERNEL...). Atualmente ele possui mais de nove milhões de linhas de código (DATA...) e pode ser executado em computadores embarcados, com poucos *megabytes* de memória RAM (*Random Access Memory*), e até super computadores, com milhares de processadores e memória RAM na ordem dos *petabytes* (LAMETER, 2008).

Algumas características do *kernel* Linux são listadas a seguir (LINUX...):

- A maior parte do código está em linguagem C, sendo que uma pequena parte está em linguagem de máquina (*assembly*)
- Suporta mais de 20 arquiteturas diferentes

- Compatível com o padrão POSIX (*Portable Operating System Interface*)
- Multitarefa e multiprocesso
- Memória virtual com suporte a paginação, *copy-on-write* e carregamento de páginas sob demanda
- Suporte a bibliotecas compartilhadas
- Suporte a diversos protocolos de rede incluindo TCP/IP (IPv4 e IPv6)
- Suporte a diversos tipos de dispositivos incluindo USB (*Universal Serial Bus*) e PCI (*Peripheral Component Interconnect*)
- Código-fonte disponível sob a Licença Pública Geral GNU versão 2

A palavra Linux é normalmente empregada para se referir a ambos, distribuições Linux[1] ou o *kernel* Linux, sendo que seu significado real pode ser extraído do contexto. Neste trabalho, no entanto, Linux é sempre empregado para se referir ao *kernel*.

Devido ao seu tamanho e complexidade, está fora do escopo deste trabalho descrever o Linux como um todo, mesmo que superficialmente. Ao invés disso as próximas seções descrevem apenas o conteúdo pertinente a este trabalho e tudo o que for descrito se aplica à versão 2.6.29.2.

1 As distribuições são encarregas de integração e “empacotamento” de *softwares* para Linux.

3.1 ESPAÇO DE ENDEREÇAMENTO DOS PROCESSOS

O espaço de endereçamento de um processo é o intervalo de endereços de memória que o processo tem permissão para usar. Por exemplo, em processadores x86 de 32-*bits* um programa pode gerar endereços de 0 até 4G – 1, mas o *kernel* também precisa de endereços e reserva o último 1G para uso próprio. Dessa maneira, o espaço de endereçamento dos processos nessas máquinas começa em 0 e termina em 3G – 1[2].

No Linux cada processo tem o seu próprio espaço de endereçamento. Isso significa que um mesmo endereço de memória virtual em dois processos distintos não tem nenhuma relação entre si e não é permitido a um processo acessar o espaço de endereçamento de outros processos. Existem apenas duas exceções a essa regra: processos podem compartilhar certas regiões de memória ou compartilhar todo seu espaço de endereçamento. *Threads,* no Linux, são processos que compartilham um único espaço de endereçamento.

A figura 8 mostra o espaço de endereçamento de um processo. Observa-se nesta que o espaço de endereçamento está divido e contêm as regiões A, B e C. Estas são as VMAs (*Virtual Memory Areas* ou Áreas de Memória Virtual) do processo. Elas são utilizadas para armazenar dados, os quais podem ser (LOVE, 2005, p. 252):

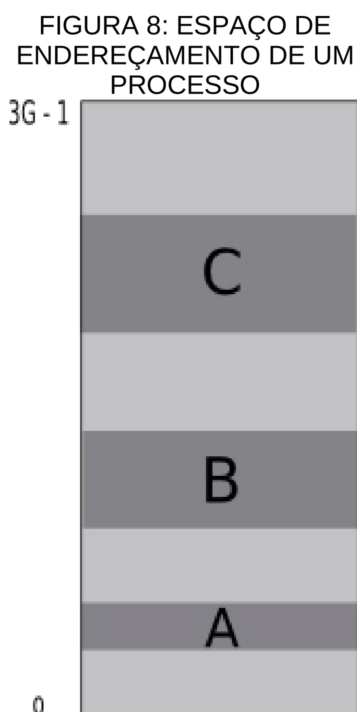
- Um mapa de memória do arquivo executável, chamado seção de texto
- Um mapa de memória das variáveis globais inicializadas do arquivo executável, chamada de seção de dados

---

2 Esse é um parâmetro configurável, mas o padrão é conforme o que foi explicado.

- Um mapa de memória da *zero page* (uma página de memória cujo conteúdo são zeros) contendo variáveis globais não inicializadas, chamada seção bss (*block started by symbol*)
- A pilha do processo
- Memória alocada com a função de biblioteca malloc()
- Texto, dados e bss de bibliotecas compartilhadas
- Segmentos de memória compartilhada
- Arquivos mapeados em memória

FIGURA 8: ESPAÇO DE ENDEREÇAMENTO DE UM PROCESSO



As VMAs possuem permissões de acesso. Caso um processo faça um acesso inválido (como tentar escrever em um endereço cuja VMA especifique que seja somente de leitura) ele é terminado pelo *kernel* com o sinal *SIGSEGV* e a famosa mensagem *Segmentation Fault* (ou falha de segmentação, em Português) é impressa. Acessar um endereço que não esteja dentro de nenhuma VMA também é caracterizado como acesso inválido.

As próximas seções explicam as principais estruturas de dados utilizadas pelo Linux para gerenciar o espaço de endereçamento dos processos.

3.1.1 Descritor de memória

O espaço de endereçamento de um processo é representado pela estrutura mm_struct, também chamada de descritor de memória. Ela é declarada no arquivo de cabeçalho <linux/mm_types.h>. Cada processo de usuário possui apenas uma instância dessa estrutura.

O quadro 11 mostra os membros da mm_struct que são diretamente relacionados a este trabalho.

QUADRO 11: MEMBROS RELEVANTES DA ESTRUTURA MM_STRUCT

Tipo	Nome do membro	Comentário
struct vm_area_struct *	mmap	Lista ligada de VMAs
struct rb_root	mm_rb	Raiz da árvore rubro-negra de VMAs
struct vm_area_struct *	mmap_cache	Última VMA pesquisada
int	map_count	Quantidade de VMAs
struct rw_semaphore	mmap_sem	Semáforo para acesso concorrente

O Linux armazena as VMAs de um processo em duas estruturas de dados distintas, uma lista ligada e uma árvore rubro-negra. A lista é usada em operações em que é necessário percorrer todas as VMAs, enquanto que a árvore é usada para se pesquisar determinados endereços (LOVE, 2005, p. 254).

O ponteiro mmap armazena o primeiro elemento da lista ligada de VMAs, a partir dele é possível percorrer todos os outros elementos. O membro mm_rb é raiz da árvore rubro-negra. Sempre que uma pesquisa na árvore é feita, o elemento

encontrado é armazenado no membro mmap_cache. Acesso concorrente na árvore ou na lista é controlado pelo semáforo mmap_sem.

Estruturas mm_struct são alocadas quando o *kernel* constrói o espaço de endereçamento de novos processos, isso normalmente acontece quando processos são criados com a chamada de sistema clone() ou novos binários são executados com execve(). Como dito anteriormente *threads* compartilham o mesmo espaço de endereçamento, logo elas compartilham a mesma instância da estrutura mm_struct.

A destruição de uma mm_struct ocorre quando o *kernel* destrói o espaço de endereçamento do processo, isso acontece quando o processo finaliza, é terminado pelo *kernel* ou faz uma chamada a _exit()[3].

3.1.2 Áreas de Memória Virtual

Áreas de memória virtual (ou VMAs) são representadas pela estrutura vm_area_struct, declarada no arquivo <linux/mm_types.h>. Uma instância dessa estrutura é alocada para cada VMA que o processo possui.

Todos os seus membros são listados no quadro 12.

3 A função exit() (sem o caractere de sublinha) não é uma chamada de sistema, ela é uma função da biblioteca C padrão que faz algum trabalho de limpeza e então chama _exit() (a chamada de sistema)

QUADRO 12: MEMBROS DA ESTRUTURA VM_AREA_STRUCT

Tipo	Nome do membro	Comentário
struct mm_struct *	vm_mm	Espaço de endereçamento desta VMA
unsigned long	vm_start	Primeiro endereço válido dentro da VMA
unsigned long	vm_end	Primeiro endereço fora da VMA
struct vm_area_struct *	vm_next	Próxima VMA na lista ligada
pgprot_t	vm_page_prot	Permissão de acesso das páginas físicas
unsigned long	vm_flags	Permissão e *flags*
struct rb_node	vm_rb	Nó da árvore rubro-negra
union (struct vm_set e struct raw_prio_tree_node)	shared	Estrutura de dados usadas para mapeamento reverso
struct list_head	anon_vma_node	Lista duplamente ligada de regiões anônimas
struct anon_vma *	anon_vma	Ponteiro para a estrutura anon_vma
struct vm_operations_struct *	vm_ops	Ponteiro para os métodos da VMA
unsigned long	vm_pgoff	*Offset* dentro do arquivo mapeado em memória
struct file *	vm_file	Ponteiro para o arquivo mapeado
void *	vm_private_data	Dado privado da VMA
unsigned long	vm_truncate_count	Usado para liberar um endereço linear em um arquivo mapeado não-linear
struct vm_region *	vm_region	Região mapeada NOMMU
struct mempolicy *	vm_policy	Política para máquinas NUMA

Uma área de memória virtual é um intervalo de endereços virtuais. Os membros vm_start e vm_end contêm, respectivamente, o primeiro endereço dentro da VMA e o primeiro endereço fora da VMA. Assim, pode-se usar a subtração

vm_start – vm_end para se obter o tamanho da área de memória virtual.

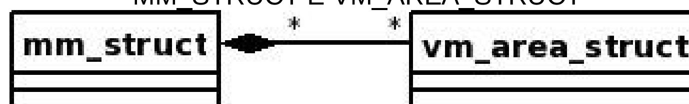
O ponteiro vm_mm guarda o endereço da instância da estrutura mm_struct que a VMA pertence. Como dito anteriormente as VMAs são armazenadas em uma lista ligada e uma árvore rubro-negra, o ponteiro vm_next guarda o endereço da próxima vm_area_struct na lista e o membro vm_rb contêm os dados utilizados pela árvore rubro-negra.

As permissões de acesso da VMA são armazenadas no membro vm_flags. Essas permissões são próprias do Linux e para fazerem sentido para o *hardware* de memória virtual, elas devem ser traduzidas para os *bits* de proteção usados nas tabelas de páginas. Esse trabalho é feito pela função vm_get_page_prot(), cujo resultado é armazenado no membro vm_page_prot o qual é acessado sempre que uma nova página física é alocada para a VMA. É importante notar que todas as páginas físicas que constituem uma VMA possuem a mesma permissão de acesso.

Uma VMA pode representar um arquivo mapeado em memória, nesse caso o membro vm_file contêm um ponteiro para o arquivo e vm_pgoff contêm o deslocamento (*offset*) atual, dentro do arquivo. A VMA também pode representar uma área para alocação dinâmica de memória, conhecida como *heap*. As páginas alocadas para esse fim são chamadas de anônimas pelo *kernel*, pois elas não estão relacionadas com nenhum arquivo. Os membros anon_vma_node e anon_vma são utilizados para gerenciar essa área.

Os membros restantes da vm_area_struct (union shared, vm_private_data, vm_truncate_count, vm_region e vm_policy) possuem usos específicos e não serão discutidos neste trabalho.

FIGURA 9: RELAÇÃO ENTRE AS ESTRUTURAS MM_STRUCT E VM_AREA_STRUCT



É muito importante observar a relação entre a estrutura mm_struct (discutida na seção anterior) e a estrutura vm_area_struct, conforme apresenta a figura 9.

Na figura se visualiza um diagrama UML (*Unified Modeling Language*) que representa as estruturas como classes. Embora a linguagem C não suporte classes, é conveniente representar dessa maneira pois se torna fácil visualizar a relação. Nesta, observa-se que uma vm_area_struct pode estar vinculada a várias mm_struct. Esse caso acontece quando uma mesma VMA é compartilhada entre processos. Também se nota a relação de composição, pois instâncias de vm_area_struct só existem como parte de mm_struct, usadas para representar o espaço de endereçamento dos processos.

3.1.3 Busca de VMAs

De acordo com Love (2005, p. 261) o *kernel* frequentemente tem que buscar áreas de memória no espaço de endereçamento do processo, seguindo alguns critérios de busca. Mais especificamente as seguintes operações fazem com que o *kernel* busque uma VMA:

- Quando o processo acessa um endereço de memória virtual que ainda não tem páginas físicas alocadas
- Quando é necessário achar um intervalo de endereços livres dentro do espaço de endereçamento do processo

- Em várias operações de alteração do espaço de endereçamento executadas pelo próprio processo, como por exemplo as chamadas de sistema brk(), mprotect(), madvise(), mincore(), shmdt() entre outras

O quadro 13 mostra as assinaturas das principais funções de busca de VMAs utilizadas pelo *kernel,* todas elas são definidas no arquivo mm/mmap.c.

QUADRO 13: ASSINATURA DAS FUNÇÕES DE BUSCA DE VMAS

```
struct vm_area_struct *find_vma(struct mm_struct *mm,
                                unsigned long addr);

struct vm_area_struct *find_vma_prev(struct mm_struct *mm,
                                     unsigned long addr,
                                     struct vm_area_struct **pprev);

struct vm_area_struct *find_vma_prepare(struct mm_struct *mm,
                                     unsigned long start_addr,
                                     struct vm_area_struct **pprev,
                                     struct rb_node ***rb_link,
                                     struct rb_node **rb_parent);
```

De acordo com Love (2005, p. 262), a função find_vma() é utilizada para se buscar a primeira VMA de um processo cujo membro vm_end é maior que addr. Essa função é a mais importante na busca de VMAs, pois é utilizada por qualquer subsistema ou *driver* que tenha que fazer a pesquisa. Ela também é utilizada pelo *page fault handler* quando há uma falha de página.

As outras funções, embora importantes, são secundárias. find_vma_prev() funciona como find_vma(), mas também retorna a VMA que se encontra antes de addr. Essa função é utilizada em alguns casos específicos, um deles é quando existe a necessidade de se verificar se é possível unir uma VMA com a anterior.

A última função, find_vma_prepare(), assim como find_vma_prev() também retorna a VMA que se encontra antes de addr, porém ela também retorna informações utilizadas para se inserir uma nova VMA na árvore rubro-negra.

4 METODOLOGIA DE DESENVOLVIMENTO

Como o objetivo deste projeto é a implementação de uma estrutura de dados para a análise de desempenho, a utilização de uma metodologia de desenvolvimento para sistemas de grande porte, como o RUP (*Rational Unified Process*), gera uma grande quantidade de trabalho e traz poucos benefícios.

Para maior produtividade este projeto utilizou uma metodologia *ad hoc* que consistiu nas seguintes fases:

1. As estruturas de dados e os algoritmos apresentados no capítulo 2 foram implementados na linguagem *Python*. Por ser interpretada, possuir tipos de alto nível e tipagem dinâmica, *Python* é uma ótima linguagem para *rapid prototyping* (WHAT...). O objetivo desta fase é verificar, através de testes de unidade, que as principais propriedades dos algoritmos funcionam conforme o esperado
2. A implementação em *Python* foi convertida para a linguagem C, para que seja utilizada no *kernel* Linux. No entanto, a implementação em C pode ser feita e testada em espaço de usuário e utilizada no *kernel* Linux com poucas modificações, pois os algoritmos são auto contidos
3. Todos os testes de desempenho (discutidos na seção 4.1) foram executados para a árvore rubro-negra (implementação atual), árvore de busca binária e árvore de afunilamento. A árvore de busca binária foi incluída nos testes para servir como parâmetro de comparação

As próximas seções explicam os procedimentos de testes, as ferramentas e a infraestrutura de *hardware* utilizada no processo de desenvolvimento.

4.1 TESTES DE DESEMPENHO

Os testes de desempenho listados nesta seção tem o objetivo de gerar dados com qualidade e quantidade suficientes para se fazer uma análise comparativa entre o desempenho da árvore rubro-negra e a árvore de afunilamento no armazenamento das VMAs de um processo.

O critério para a escolha dos testes seguiu a mesma linha apresentada por Ptaff (2004, p. 5) no que ele chamou de *Real-World Data Sets* (Conjunto de Dados do Mundo Real), ou seja, foram apenas escolhidos testes capazes de simular ambientes da vida real.

4.1.1 Tempo de construção do *kernel* Linux

De acordo com Balsa (1997), o teste mais simples que se pode executar para medir o desempenho de componentes de um sistema Linux é medir o tempo de construção do próprio *kernel*, pois essa tarefa exercita muitas áreas que também são exercitadas em outros testes de desempenho mais específicos.

4.1.2 Hackbench

Hackbench é um programa que mede a escalabilidade, desempenho e *overhead* do escalonador de processos do Linux. O teste consiste em criar grupos de processos ou *threads* que trocam mensagens entre si, usando *sockets*.

Hackbench pode criar centenas ou milhares de processos e *threads,* e a troca de mensagens exercita o código de VMA.

A versão utilizada se encontra disponível no endereço http://people.redhat.com/mingo/cfs-scheduler/tools/hackbench.c.

4.1.3 LMbench

LMbench é uma suite de testes de desempenho. Ela faz medições de vários componentes do *kernel* e foi criada para ser portável entre sistemas operacionais Unix e clones, podendo assim ser usada para se fazer comparações de desempenho entre sistemas operacionais diferentes.

A versão utilizada foi a 3.0-a9, a qual pode ser encontrada na página oficial do projeto em http://sourceforge.net/projects/lmbench.

4.1.4 Sysbench

Sysbench é uma ferramenta de teste de performance utilizada para se avaliar parâmetros do sistema operacional que são importantes para sistemas que executam banco de dados com carga intensiva. Para isso, acessos simultâneos a um banco de dados MySQL são simulados.

A versão utilizada foi a 0.4.12, a qual pode ser encontrada na página oficial do projeto em http://sysbench.sourceforge.net.

4.2 INFRAESTRUTURA DE *HARDWARE*

Os testes foram executados nas máquinas listadas nos quadros 14 e 15. Foram escolhidos, respectivamente, um *laptop* com baixo poder de processamento, e uma máquina servidor com alto poder de processamento.

QUADRO 14: MÁQUINA *LAPTOP*

N. CPUs	Memória RAM	Processador	Cache L1 (instruções/dados)	Cache L2	Espaço em disco
1	512M	Intel Celeron M	32K/32K	1024K	40G

QUADRO 15: MÁQUINA SERVIDOR

N. de Cores	Memória RAM	Processador	Cache L1 (instruções/dados)	Cache L2	Espaço em disco
4	2G	Intel Core2 Quad Q6600	32K/32K	4096K	250G

4.3 PROGRAMAS E FERRAMENTAS

Os principais programas e ferramentas necessários para reproduzir o ambiente de desenvolvimento são listados no quadro 16. A distribuição Linux utilizada foi o Mandriva Linux 2009.0, sendo assim todas as ferramentas utilizadas provêm deste.

QUADRO 16: PRINCIPAIS FERRAMENTAS UTILIZADAS NO PROJETO

Nome	Versão	Função
Bash	3.2-10.1mdv2009.0	Interpretador de comandos
Biblioteca C padrão GNU (glibc)	2.8-1.20080520.5mnb2	Biblioteca C padrão
Gnuplot	4.2.3-2mdv2009.0	Criação de gráficos
GNU *Binary Utilities* (binutils)	2.18.50.0.8-1mnb2	Coleção de programas para a manipulação de arquivos objetos
GNU *Compiler Collection* (GCC)	4.3.2-3mnb2	Compilador C
GNU make	3.81-3mdv2009.0	Sistema de construção de *software*
Linux *Utilities* (util-linux-ng)	2.14.1-4.1mdv2009.0	Coleção de programas de sistema para Linux
Module-init-tools	3.5-2.1mdv2009.0	Coleção de programas para a manipulação de módulos do *kernel*
MySQL	5.0.81-0.2mdv2009.0	Sistema de banco de dados relacional
Procps	3.2.7-6mdv2009.0	Coleção de programas de sistema para Linux
Quilt	0.46-1mdv2008.1	Gerenciamento de *patches*

5 IMPLEMENTAÇÃO

Este capítulo descreve alguns detalhes relacionados com as modificações feitas no *kernel* Linux para comportar as árvores de afunilamento e busca binária implementadas neste projeto.

O quadro 17 lista as funções e macros do *kernel* Linux que foram modificadas por utilizarem diretamente o código da árvore de VMA.

QUADRO 17: FUNÇÕES E MACROS DO *KERNEL* LINUX MODIFICADAS

Função/Macro	Arquivo	Descrição
copy_vma()	mm/mmap.c	Copia uma VMA para uma nova localização
detach_vmas_to_be_unmapped()	mm/mmap.c	Remove VMAs de um espaço de endereçamento dado
do_brk()	mm/mmap.c	Implementa parte da chamada de sistema brk()
dup_mmap()	kernel/fork.c	Duplica o espaço de endereçamento de um processo
find_vma()	mm/mmap.c	Busca uma VMA
find_vma_prepare()	mm/mmap.c	Busca uma VMA, mas retorna mais informação
find_vma_prev()	mm/mmap.c	Busca VMA anterior a um endereço dado
__insert_vm_struct()	mm/mmap.c	Insere uma VMA em diversas estrutura de dados
INIT_MM()	include/linux/init_task.h	Inicializa tabela de memória do processo init
mmap_region()	mm/mmap.c	Mapeia uma região de memória
vma_link()	mm/mmap.c	Insere uma VMA em diversas estruturas de dados
__vma_link()	mm/mmap.c	Insere uma VMA na árvore e lista de VMAs

__vma_link_list()	mm/mmap.c	Insere uma VMA na lista de VMAs
__vma_link_rb()	mm/mmap.c	Insere uma VMA na árvore rubro-negra
__vma_unlink()	mm/mmap.c	Remove uma VMA da árvore e da lista de VMAs

A árvore de busca binária seguiu o mesmo modelo da árvore rubro-negra e sua implementação foi relativamente simples. Uma vez implementada no *kernel* Linux, apenas foi necessário substituir chamadas à árvore rubro-negra por chamadas à árvore de busca binária.

Por outro lado, a implementação da árvore de afunilamento trouxe dois desafios. Primeiro, a função find_vma_prev() faz uma busca na árvore com uma semântica ligeiramente diferente de find_vma(), então foi necessário adaptar a operação de afunilamento à sua semântica. Segundo, e mais importante, muitas partes do *kernel* assumem que buscar um nó na árvore é uma operação *read-only* (apenas-leitura) e desse modo um semáforo *read-only* é usado para proteger a árvore contra acesso paralelo. Porém, a árvore de afunilamento modifica a árvore durante a busca, então foi necessário alterar todas funções do *kernel* que buscam na árvore para usar o semáforo *read-write* (leitura-escrita). De acordo com o sumário da modificação gerado pelo programa *quilt*, foram alterados 23 arquivos adicionais apenas para resolver esse problema.

6 RESULTADOS

Este capítulo apresenta e discute os resultados de todos os testes de desempenho executados.

A nomenclatura utilizada para os *kernels* modificados segue o padrão utilizado pelos desenvolvedores oficiais do *kernel* Linux, o qual é a versão do *kernel* seguida da abreviatura que descreve a modificação em Inglês (KERNELTREES...).

O quadro 18 lista os nomes adotados.

QUADRO 18: NOMES ADOTADOS PARA OS *KERNELS* UTILIZADOS

Nome do *kernel*	Versão	Abreviatura	Descrição
2.6.29.2-rbtree	2.6.29.2	rbtree	*Red-Black Tree* (árvore rubro-negra)
2.6.29.2-bstree	2.6.29.2	bstree	*Binary Search Tree* (árvore de busca binária)
2.6.29.2-sptree	2.6.29.2	sptree	*Splay Tree* (árvore de afunilamento)

6.1 TAMANHO DAS ESTRUTURAS

O quadro 19 apresenta o tamanho, em *bytes,* das estruturas que estão diretamente relacionadas com a implementação das árvores de VMA.

QUADRO 19: TAMANHO DAS ESTRUTURAS EM *BYTES*

Kernel	mm_struct	vm_area_struct	Nó	Raiz
2.6.29.2-rbtree	408	84	12	4
2.6.29.2-bstree	408	84	12	4
2.6.29.2-sptree	404	80	8	4

Observa-se no quadro 19 que a implementação da árvore de busca binária não altera o tamanho das estruturas. Porém, a árvore de afunilamento diminui em 4 *bytes* todas as estruturas, exceto a utilizada para representar a raiz da árvore.

O quadro 20 apresenta a diminuição em porcentagem.

QUADRO 20: DIMINUIÇÃO DAS ESTRUTURAS EM PORCENTAGEM

Kernel	mm_struct	vm_area_struct	Nó	Raiz
2.6.29.2-sptree	0,98%	4,76%	33,33%	0.00%

Essa redução se deve ao fato de a árvore de afunilamento apenas armazenar dois ponteiros em cada nó (filho da esquerda e direita) e dispensar o uso do membro mmap_cache na estrutura mm_struct. A árvore rubro-negra, por sua vez, necessita de espaço para armazenar a informação da cor do nó, enquanto que a árvore de busca binária guarda o endereço do nó pai de cada nó.

É importante observar que estruturas menores trazem dois benefícios importantes para qualquer programa. Primeiro, há redução no consumo de memória.

Segundo, quanto menor o tamanho, maior as chances das instâncias dessas estruturas caberem e permanecerem no *cache* do processador.

6.2 TEMPO DE CONSTRUÇÃO DO *KERNEL* LINUX

Este teste consistiu em construir o *kernel* Linux 2.6.29.2 na máquina Servidor utilizando-se 4, 16, 32 e 64 *jobs* do programa *make*. O quadro 21 mostra o tempo (em segundos) gasto em modo *kernel* durante o processo de construção.

QUADRO 21: TEMPO DE CONSTRUÇÃO DO *KERNEL* LINUX EM MODO *KERNEL* (SEGUNDOS)

Kernel	*Jobs*/Tempo em modo *kernel* (segundos)			
	4	16	32	64
2.6.29.2-rbtree	9.21	8.72	8.92	8.91
2.6.29.2-bstree	9.33	8.79	8.78	8.92
2.6.29.2-sptree	9.15	8.85	8.82	8.88

Conforme apresenta o quadro 21, a variação entre os tempos de construção é mínima. Uma investigação mais profunda mostrou que, na verdade, a árvore de VMA dos processos do programa *make* e do compilador GCC são muito pequenas (em torno de 25 nós). Assim, é provável que mudanças na árvore de VMA não influenciam substancialmente no tempo de construção do *kernel* Linux.

6.3 SYSBENCH

De todos os testes escolhidos, o *sysbench* foi o que mostrou maior variação entre as árvores, permitindo assim uma melhor análise para este tipo de uso do sistema. Por essa razão, esse teste foi executado com diversos parâmetros diferentes, os quais serão apresentados nas seções seguintes, utilizando-se o número de *threads* de cada teste para denomina-las.

Em todas as execuções, o *sysbench* foi configurado para simular acesso a um banco de dados MySQL, o qual tem que gerenciar o acesso paralelo de uma quantidade especificada de *threads,* as quais fazem transações para ler dez mil registros de uma determinada tabela.

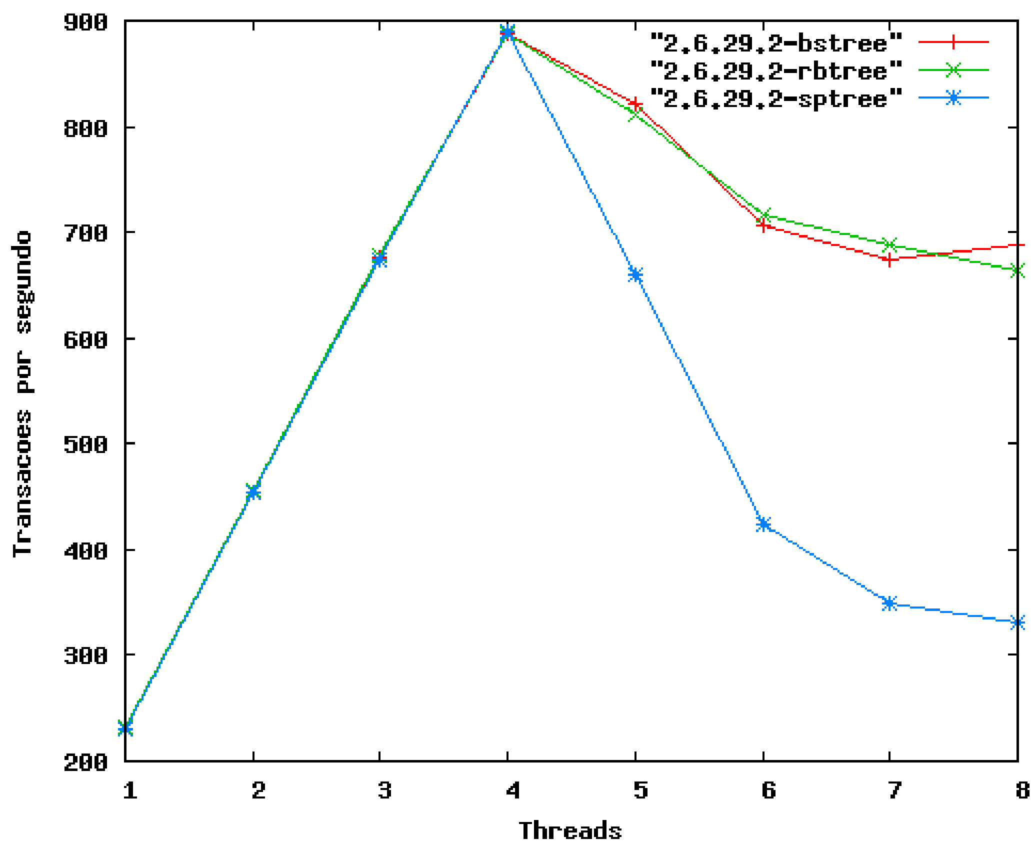
Todas as seções apresentam um gráfico e uma discussão sobre ele. Em todos, o eixo x representa o número de transações por segundo que o teste foi capaz de executar. O eixo y, por sua vez, representa o número de *threads* utilizadas no teste. O melhor desempenho é aquele que consegue executar o maior número de transações por segundo.

6.3.1 De 1 a 8 *threads*

Neste teste o *sysbench* foi executado com 1, 2, 3, 4, 5, 6, 7 e 8 *threads*, utilizando-se 60 segundos para cada execução.

No gráfico 1, observa-se dois comportamentos importantes. Primeiro, de 1 a 4 *threads* todos os *kernels* tiveram o mesmo desempenho. Isso se deve ao fato de as árvores de VMA serem pequenas (máximo de 86 nós para 8 *threads*).

GRÁFICO 1: SYSBENCH MÁQUINA SERVIDOR: 1 A 8 *THREADS*



Segundo, e mais importante, com mais de 4 *threads* o *kernel* com a árvore de afunilamento degrada muito mais rapidamente que os outros dois *kernels*. A hipótese mais provável diz respeito ao uso do semáforo utilizado para controlar o acesso a árvore de VMA.

No caso das árvores rubro-negra e busca binária, conforme explicado no capítulo 5 Implementação, o semáforo usado para pesquisar a árvore é *read-only* (apenas-leitura), o qual permite que várias *threads* pesquisem a árvore ao mesmo tempo. Entretanto, como a árvore de afunilamento faz alterações durante a pesquisa, torna-se necessário utilizar o semáforo *read-write* (leitura-escrita), o qual permite que apenas uma *thread* pesquise a árvore por vez, **provavelmente** causando o problema evidenciado no gráfico.

Existem algumas maneiras de diminuir ou talvez resolver esse problema, elas são discutidas no capítulo 8 Trabalhos futuros.

O quadro 22 apresenta os resultados para os mesmos testes executados na máquina *laptop*. Como se pode observar a variação é muito pequena. Como a taxa de erro da medição é alta, deve-se assumir que para o teste executado não houve variação significativa.

QUADRO 22: SYSBENCH MÁQUINA LAPTOP: 1 A 8 THREADS

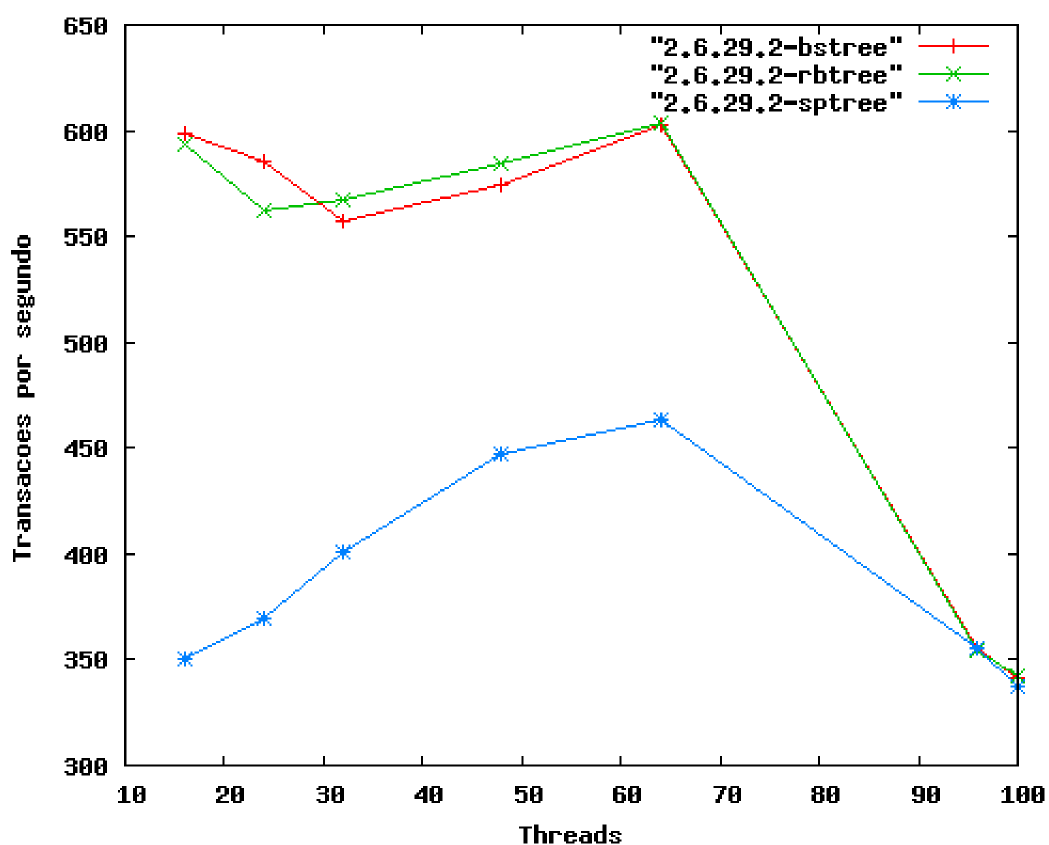
Kernel	*Threads*/Transações por segundo							
	1	2	3	4	5	6	7	8
2.6.29.2 -rbtree	116.64	116.95	117.09	116	115.92	115.23	116.92	116.04
2.6.29.2 -bstree	117.19	117.24	116.62	116.39	115.97	116.39	115.22	115.5
2.6.29.2 -sptree	117.07	117.04	116.5	116.02	116	115.45	115.56	115.39

Observou-se que na máquina *laptop* todos os testes do *sysbench* seguiram o mesmo comportamento apresentado no quadro 22, ou seja, as árvores tiveram desempenho similares. O que pode evidenciar que em máquinas pequenas modificações na árvore de VMA causam pouco ou nenhum impacto. Por esse motivo, apenas os resultados obtidos na máquina Servidor serão discutidos nas próximas seções.

6.3.2 De 16 a 100 threads

Neste teste o *sysbench* foi executado com 16, 24, 32, 48, 64, 96 e 100 *threads,* utilizando-se 60 segundos para cada execução.

GRÁFICO 2: SYSBENCH MÁQUINA SERVIDOR: 16 A 100 THREADS



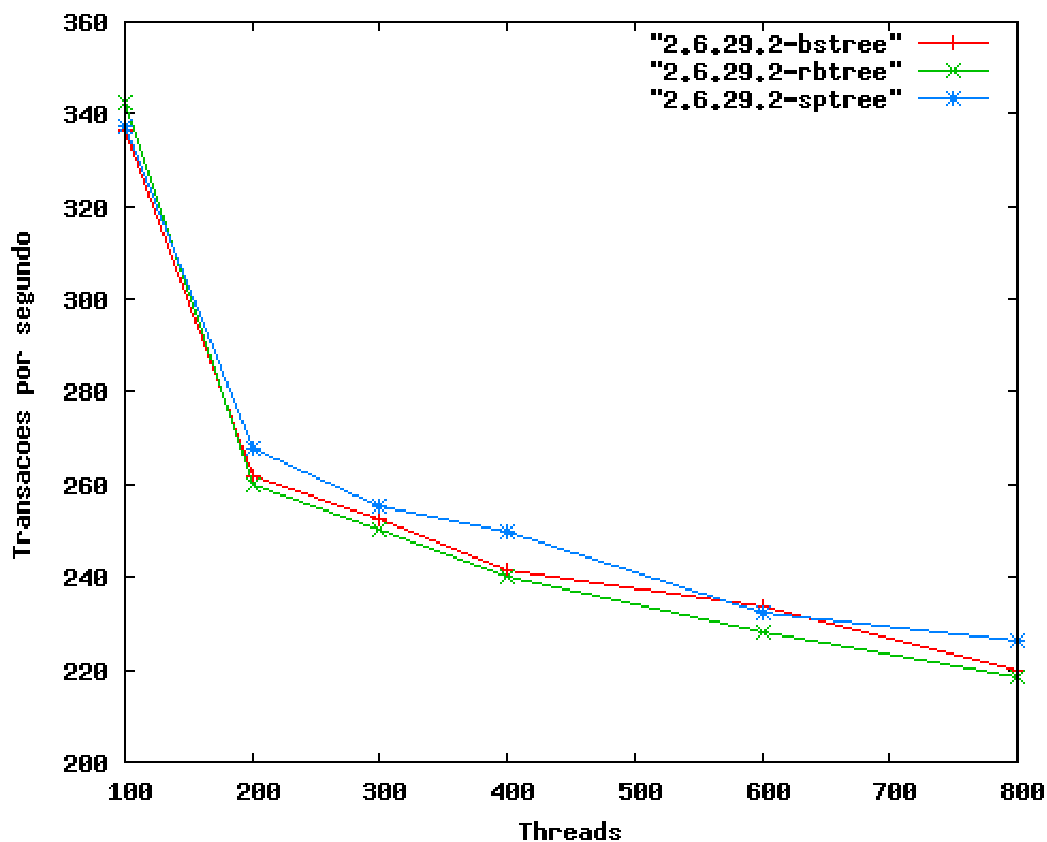
No gráfico 2, visualiza-se que os *kernels* com a árvore de busca binária e rubro-negra tiveram desempenho similares. A pequena variação entre eles pode ser desprezada, uma vez que a taxa de erro da medição é alta. O *kernel* com a árvore de afunilamento teve o pior desempenho, a razão mais provável é o problema de contenção do semáforo da árvore, conforme explicado na seção anterior.

Porém, deve-se observar que entre 16 e 64 *threads* o *kernel* com a árvore de afunilamento **aumentou** seu desempenho enquanto que os outros oscilaram.

6.3.3 De 100 a 800 threads

Neste teste o *sysbench* foi executado com 100, 200, 300, 400, 600 e 800 *threads,* utilizando-se 180 segundos para cada execução.

GRÁFICO 3: SYSBENCH MÁQUINA SERVIDOR: 100 A 800 THREADS



No gráfico 3, observa-se que os *kernels* com a árvore rubro-negra e árvore de busca binária apresentaram desempenho similares, porém o *kernel* com a árvore de afunilamento mostrou desempenho ligeiramente melhor, principalmente com a carga entre 200 e 600 *threads*.

Este resultado faz surgir o questionamento que se a árvore de afunilamento teria desempenho ainda melhor com um grande número de nós na árvore e uma seqüência de acesso longa.

6.4 HACKBENCH

Ao contrário do esperado e assim como o teste de construção do *kernel* (seção 6.2), os resultados para o teste da ferramenta *hackbench* não apresentou variação significativa entre as árvores.

O quadro 23 apresenta os resultados em segundos para o grupo de processos especificados.

QUADRO 23: RESULTADOS DO TESTE *HACKBENCH* MÁQUINA SERVIDOR

	Grupo de processos / tempo em segundos				
Kernel	75	100	125	150	175
2.6.29.2-rbtree	2.59	3.48	4.30	5.20	6.09
2.6.29.2-bstree	2.61	3.44	4.32	5.19	6.09
2.6.29.2-sptree	2.59	3.46	4.34	5.23	6.10

6.5 LMBENCH

Foi executado apenas um teste da *suite* Lmbench: a medição do tempo de criação de processos com as chamadas fork() e exec().

O quadro 24 apresenta os resultados do teste, o tempo é dado em microsegundos.

QUADRO 24: RESULTADOS LMBENCH PARA CRIAÇÃO DE PROCESSOS (MÁQUINA SERVIDOR)

Kernel	fork()	exec()
2.6.29.2-rbtree	115.3	122.91
2.6.29.2-bstree	113.44	119.25
2.6.29.2-sptree	115.87	122.14

No quadro 24, observa-se que entre os *kernels* com árvores rubro-negra e afunilamento houve pequena variação. Essa informação é importante, pois se pode concluir que a árvore de afunilamento não altera substancialmente o desempenho daquelas chamadas.

Nota-se também que o *kernel* com a árvore de busca binária foi o mais eficiente. Esse fato é esperado, pois na criação de novos processos a operação mais usada nas árvores é a de inserção, a qual pode ser mais rápida na árvore de busca binária por esta não implementar o balanceamento.

7 CONCLUSÃO

A proposta inicial desta pesquisa tinha como objetivo principal implementar uma árvore de afunilamento no *kernel* Linux e compará-la com a árvore rubro-negra atual, a qual é utilizada no armazenamento das Áreas de Memória Virtual (*Virtual Memory Areas*, ou VMA em Inglês) dos processos.

Portanto, como considerações finais do trabalho proposto, é possível afirmar que o objetivo principal foi parcialmente alcançado. Embora a árvore de afunilamento tenha sido devidamente implementada, esperava-se que os testes de desempenho selecionados e as máquinas escolhidas fossem gerar dados em maior quantidade e qualidade. Apesar de revelar algumas características importantes sobre a árvore de afunilamento, não foi possível simular o ambiente que realmente colocaria a árvore de VMA em todo o seu limite.

Entretanto, os resultados obtidos com essa pesquisa são importantes como comparação inicial entre as árvores escolhidas, e servem para mostrar em quais tipos de uso de um sistema Linux a estrutura de dados usada para armazenar as VMAs dos processos podem ter influência maior ou menor.

A maior dificuldade encontrada para a realização desta pesquisa foi selecionar os testes de desempenho que simulam usos do sistema na vida real e ao mesmo tempo dependem de um espaço de endereçamento suficientemente grande e segmentado, o qual dependeria muito de um bom desempenho da árvore de VMAs.

8 TRABALHOS FUTUROS

Existem duas variações do algoritmo da árvore de afunilamento que podem minimizar o problema de contenção do semáforo evidenciado no gráfico 1, sendo assim, sua implementação é sugerida como trabalho complementar a este estudo.

O primeiro deles se chama “Semi-afunilamento” e é sugerido no próprio artigo de Sleator e Tarjan (p. 670). Neste algoritmo, o objetivo não é levar o nó até a raiz, mas sim diminuir sua profundidade na árvore pela metade. Dessa maneira, espera-se que a operação de afunilamento termine mais rapidamente com menos escritas na memória.

O segundo algoritmo se chama Árvore de Afunilamento Randômica (Albers; Karpinski), neste a árvore é alterada para executar a operação de afunilamento apenas com uma certa probabilidade *p*.

Uma terceira sugestão de trabalho seria investigar o comportamento da árvore de VMA em ambientes que utilizam sistemas de banco de dados com uma carga excessivamente grande, por exemplo, com máquinas de 64-*bits* que possuam dezenas de processadores e centenas de *gigas* de memória RAM.

REFERÊNCIAS

ALBERS S.; KARPINSKI M. Randomized Splay Trees: Theoretical and Experimental Results. Disponível em http://www.informatik.uni-freiburg.de/~salbers/ipl02.ps.gz. Acesso em 2009-05-19.

BALSA, D. A. Linux Benchmarking HOWTO, 1997. Disponível em http://oss.sgi.com/LDP/HOWTO/Benchmarking-HOWTO.html. Acesso em 2009-05-17.

CORMEN, T. H.; *et al*. *Introduction to Algorithms.* Segunda Edição. Cambridge: MIT Press, 2001.

DATA and statistics on the latest versions of the Linux kernel. *The H Open Source*. Disponível em http://www.h-online.com/open/Improvements-throughout-what-s-new-in-Linux-2-6-25--/features/110526/5. Acesso em 2009-04-20.

KERNEL TREES. *Kernelnewbies*. Disponível em http://kernelnewbies.org/FAQ/VariousKernelTrees. Acesso em 2009-05-19.

KNUTH, D. E. *The Art of Computer Programming: Fundamental Algorithms.* Terceira Edição. California: Addison – Wesley, 1997.

KNUTH, D. E. *The Art of Computer Programming: Sorting and Searching.* Segunda Edição. California: Addison – Wesley, 1998.

LAMETER, Christoph. Bazillions of pages: The future of memory management under Linux, 2008. Disponível em: http://ols.fedoraproject.org/OLS/Reprints-2008/lameter-reprint.pdf. Acesso em 2008-10-08.

LINUX kernel. *Wikipedia*. Disponível em: http://en.wikipedia.org/wiki/Linux_kernel. Acesso em 2008-09-17.

LINUX KERNEL Development. *The Linux Foundation*. Disponível em https://www.linuxfoundation.org/publications/linuxkerneldevelopment.php. Acesso em 2009-04-20.

LOUDON, K. *Mastering Algorithms with C*. Primeira Edição. Carlifornia: O'Reilly, 1999.

LOVE, R. *Linux Kernel Development*. Segunda Edição. Indiana: Novell Press, 2005.

PTAFF, Ben. An Introduction to Binary Search Trees and Balanced Trees, 2002. Disponível em: ftp://ftp.gnu.org/pub/gnu/avl/avl-2.0.2.pdf.gz. Acesso em 2009-05-24.

PTAFF, Ben. Performance Analysis of BSTs in System Software, 2004. Disponível em: http://www.stanford.edu/~blp/papers/libavl.pdf. Acesso em 2008-08-24.

SALUS, Peter. *A Quarter Century of UNIX*. Primeira Edição. Boston: Addison-Wesley, 1994.

SLEATOR, D. D. An implementation of top-down splaying, 1992. Disponível em http://www.link.cs.cmu.edu/link/ftp-site/splaying/top-down-splay.c. Acesso em 2008-09-14.

SLEATOR, D. D.; TARJAN, R. E. Self-Adjusting Binary Search Trees. *Journal of the ACM*, 1985. Disponível em http://www.cs.cmu.edu/~sleator/papers/self-adjusting.pdf. Acesso em 2008-09-14.

SPLAY tree. *Wikipedia*. Disponível em: http://en.wikipedia.org/wiki/Splay_tree. Acesso em 2008-09-24.

UNIX. Wikipedia. Disponível em: http://en.wikipedia.org/wiki/Unix. Acesso em 2009-04-20.

WHAT is Python. Disponível em http://www.python.org/doc/essays/blurb/. Acesso em 2008-10-07.