Implementação de Tabelas em Aplicações de Tempo Real Usando Alocação Contígua

Rômulo Silva de Oliveira*

Olinto José Varela Furtado+

* I I - Univ. Fed. do Rio Grande do Sul
Caixa Postal 15064
Porto Alegre-RS, 91501-970, Brasil
romulo@inf.ufrgs.br

+ INE - Univ. Fed. de Santa Catarina
Caixa Postal 476
Florianopolis-SC, 88070-900, Brasil
olinto@inf.ufsc.br

Resumo

Este artigo analisa a implementação de tabelas em sistemas de tempo real. São consideradas as operações de inserção e pesquisa quando tabelas são implementadas através de listas contíguas alocadas estaticamente. Uma lista desordenada permite inserção rápida, mas a pesquisa é demorada. Uma lista ordenada permite uma pesquisa rápida, mas a inserção é demorada. O artigo propõe uma implementação através de vários segmentos, cada segmento correspondendo a uma lista ordenada. O resultado é uma estrutura de dados onde tanto a inserção como a pesquisa podem ser feitas em tempos razoáveis, além de suas velocidades relativas poderem ser controladas pelo programador. O gasto adicional com memória na solução proposta é pequeno. Além dos resultados analíticos, são descritas experiências onde os tempos de execução das soluções proposta e tradicionais foram comparados.

Abstract

This paper discusses the implementation of tables in real-time systems. It considers the insertion and search operations when tables are implemented as statically allocated contiguous lists. An unsorted list allows fast insertions, but searches are slow. An ordered list allows fast searches, but insertions are slow. This paper proposes an implementation based on several segments, where each segment corresponds to an ordered list. That results in a data structure where insertion as well as search can be made in reasonable time. Also, their relative speed can be controlled by the programmer. The memory overhead associated with the proposed solution is small. Besides analytical results, we describe experiments where the execution time of both proposed and classical solutions are compared.

Palavras-chave: Tabelas, Listas, Tempo-real.

1. Introdução

Sistemas computacionais de tempo real são identificados como aqueles sistemas submetidos a requisitos de natureza temporal. Nestes sistemas, os resultados devem estar corretos não somente do ponto de vista lógico, mas também devem ser gerados no momento correto. Requisitos de natureza temporal são tipicamente descritos na forma de deadlines. As abordagens descritas na literatura para a construção de sistemas tempo real são normalmente classificadas em duas grandes categorias [STA 93]. Sistemas tempo real do tipo *hard* são caracterizados por deadlines que devem ser sempre cumpridos. O não atendimento do requisito temporal representado por um deadline *hard* pode resultar em consequências catastróficas em termos financeiros, ambientais ou até mesmo vidas humanas. Em contrapartida, nos sistemas de tempo real *soft* os deadlines são apenas indicações de quando seria o melhor momento para concluir a tarefa. Nestes casos é aceito que o deadline não seja cumprido muitas vezes. Seja como for, o desenvolvimento de um sistema de tempo real demanda especial cuidado com o seu comportamento temporal, isto é, com o tempo de execução das tarefas e suas interdependências.

Uma das estruturas de dados básicas da computação é a tabela [SED 83][HOR 87]. Uma tabela pode ser descrita como uma matriz onde cada linha corresponde a um registro e cada coluna corresponde a um campo de registro. Desta forma, a tabela é composta por registros que, por sua vez, são compostos por campos. As três operações mais frequentes são a inserção de um

registro, a remoção de um registro e a pesquisa por um registro. No caso da pesquisa, um campo é escolhido como chave. O registro procurado é aquele que contém no seu campo chave o valor fornecido como parâmetro para a pesquisa. Existem muitas variações possíveis com respeito ao emprego de chaves em tabelas. Neste artigo serão consideradas apenas tabelas onde existe um único campo usado como chave e não existem dois registros com o mesmo valor no campo chave.

Existe um leque enorme de implementações possíveis para tabelas. As formas mais usadas são: lista contígua, lista encadeada, árvore binária e tabela de dispersão (*hash table*). Entre estas, a lista contígua é a forma mais simples de ser implementada. A tabela assume a forma de um array de registros. Quando os registros da tabela são mantidos ordenados pelo campo chave dentro do array, a procura por um registro em particular pode ser feita através de pesquisa binária. Caso o array não seja mantido ordenado, é necessária uma pesquisa sequencial. Embora os autores desconheçam pesquisas neste sentido, é razoável afirmar que tabelas implementadas através de arrays estão entre as estruturas de dados mais utilizadas. Por exemplo, esta implementação minimiza a memória adicional necessária além dos próprios registros (*memory overhead*), uma vantagem em sistemas de tempo real embutidos (*embedded real-time systems*).

A vantagem da pesquisa binária é necessitar $\lfloor \log N \rfloor + 1$ comparações no pior caso, quando existem **N** registros na tabela. Entretanto, para manter a tabela ordenada, a inserção de um novo elemento poderá necessitar até **N-1** deslocamentos de registros (quando o registro a ser inserido é o novo primeiro registro da tabela). Por outro lado, a pesquisa sequencial necessita de **N** comparações e, como não é necessário manter a tabela ordenada, nenhum deslocamento no momento de uma inserção.

Sistemas de tempo real são frequentemente programados como diversas tarefas que compartilham variáveis (comunicação entre processos baseada em memória compartilhada). Neste caso, algum protocolo é utilizado para impedir que duas tarefas acessem a mesma estrutura de dados ao mesmo tempo. Por exemplo, podem ser utilizados semáforos especiais (*priority ceiling*) [SHA 90]. Desta forma, o tempo de uma operação sobre a tabela acaba sendo afetado pelo tempo das demais operações. Por exemplo, suponha que o array é mantido ordenado. Uma tarefa **P** desejando fazer uma pesquisa necessita realizar apenas $\lfloor \log N \rfloor + 1$ comparações. Entretanto, caso a tabela esteja sendo acessada por uma outra tarefa **Q** fazendo uma inserção, a tarefa **P** estará sujeita a um bloqueio que poderá demorar até **N-1** deslocamentos. Neste caso, não adianta a pesquisa ser rápida se outras operações sobre a tabela forem lentas.

O objetivo deste trabalho é analisar a implementação de tabelas como listas contíguas (arrays). Especificamente, busca-se uma estrutura de dados onde os tempos de inserção e pesquisa não sejam valores muito diferentes entre si, ao mesmo tempo que o somatório dos dois valores fica próximo do obtido com implementações tradicionais. A análise apresentada neste artigo é semelhante a "análise amortizada" (*amortized analysis*) descrita na literatura de estruturas de dados [BEN 85]. Entretanto, os autores desconhecem qualquer análise deste tipo destinada especificamente a atender os requisitos dos sistemas de tempo real.

O restante do artigo está organizado da seguinte forma: a seção 2 resume as propriedades das pesquisas sequencial e binária. A seção 3 propõe uma implementação que, baseada em listas contíguas, apresenta um melhor comportamento global do que as soluções tradicionais. A seção 4 compara a proposta deste trabalho com as duas formas de implementação tradicionais, através de experiências práticas. Finalmente, na seção 5 são feitos os comentários finais sobre o trabalho.

2. Implementação Usual de Tabela Usando Array

Existe uma vasta literatura a respeito de tabelas e suas possíveis implementações. O objetivo desta seção é estabelecer as implementações clássicas que serão usadas para fins de comparação com a implementação proposta na próxima seção.

A figura 1 apresenta uma tabela típica, usada por uma aplicação que supervisiona em tempo real um conjunto de injetoras plásticas. Cada registro contém a ordem de serviço, o código da peça, a quantidade total de peças a serem produzidas, a quantidade produzida até o momento e o tempo de injeção até o momento. O campo "Ordem de Serviço" é o campo chave. Neste trabalho é suposto que dois registros não podem ter o mesmo valor para chave, o que significa que cada ordem de serviço fabrica uma única peça.

Ordem de Serviço	Código da peça	Quantidade total	Quantidade produzida	Tempo de injeção em minutos
OS-190	LALO	1500	1115	2980
OS-198	PAR-2	100	32	300
OS-203	XYZ	50	50	931
OS-215	KP99	120	0	0
OS-222	ARQT	710	342	510

Figura 1 - Tabela hipotética com 5 registros.

A tabela como apresentada na figura 1 é uma entidade abstrata. Ela pode ser implementada de várias formas. Este trabalho preocupa-se com a implementação através de lista contígua, ou array de registros. O código abaixo ilustra como esta tabela poderia ser implementada desta forma, usando a linguagem C.

```
struct descritor_OS {
    char os[10];
    char peca[10];
    unsigned quant_total;
    unsigned quant_prod;
    long tempo_min;
    }
int ocupadas;
struct descritor_OS tabela[100];
```

A partir desta implementação é possível detalhar as operações de inserção e consulta. Para garantir que não existem dois registros com a mesma chave, cada inserção é precedida de uma consulta. Embora a operação de remoção não seja analisada neste artigo, existem várias possibilidades para a sua implementação. Por exemplo, a estrutura de dados pode ser compactada após a remoção de cada registro. Também é possivel utilizar uma tarefa periódica para fazer a compactação. Neste caso é utilizado um código especial no campo "Ordem de Serviço" para identificar uma entrada livre no meio do array. Este código não pode ser usado por nenhuma ordem de serviço verdadeira.

Quando uma pesquisa sequencial é feita, o tempo no pior caso está associado com **N** comparações, onde **N** é o número de entradas. Por outro lado, a inserção é imediata. Desta forma, o tempo de inserção corresponde simplesmente à cópia dos dados para o registro.

No caso de pesquisa binária, o tempo no pior caso está associado com $\lfloor \log N \rfloor + 1$ comparações. Por sua vez, a inserção poderá exigir o deslocamento de todas as entradas do array, ou seja, o pior caso está associado com **N-1** deslocamentos (quando o n-ésimo elemento é inserido).

O tempo máximo de execução dos algoritmos em segundos depende de uma série de fatores, tais como arquitetura do computador, linguagem de programação, experiência do programador, etc. Entretanto, é seguro afirmar que o fator preponderante na inserção são os deslocamentos, ao mesmo tempo que o fator preponderante nas pesquisas são as comparações.

3. Estrutura de Dados Proposta

A estrutura de dados proposta neste artigo divide a tabela em segmentos de mesmo tamanho. Sendo **N** o número máximo de entradas e **K** o número de segmentos utilizado, cada segmento possuirá capacidade para **S** registros, isto é, **S = N / K**. Cada segmento é mantido ordenado, mas não existe nenhum tipo de relação entre os conteúdos dos diversos segmentos. Em geral, durante a execução do programa, alguns segmentos estarão lotados, outros estarão vazios e outros ainda estarão parcialmente ocupados. Não existe uma ordem estabelecida para a ocupação dos segmentos. Por exemplo, a figura 2 mostra uma tabela onde temos **N = 12**, **K = 4** e **S = 3**. Todos os segmentos estão lotados. A tabela representada pela figura 2 mostra cada segmento ordenado internamente, mas não existe qualquer tipo de ordenação intersegmentos.

segmento 1				segmento 2				segmento 3				segmento 4		
aaa				bbb				ddd				fff		
ccc				iii				eee				hhh		
ggg				kkk				lll				mmm		

Figura 2 - Tabela com 4 segmentos, cada segmento possui 3 registros.

O código abaixo ilustra como esta estrutura de dados pode ser programada em C. Observe que agora a tabela é um array com **K** segmentos. Cada segmento tem capacidade para armazenar **S** registros.

```
struct descritor_OS {
     char os[10];
     char peca[10];
     unsigned quant_total;
     unsigned quant_prod;
     long tempo_min;
     }
struct segmento {
     int ocupadas;
     struct descritor_OS dados[S];
     }
struct segmento tabela[K];
```

No código acima é usada alocação estática para obter a memória de toda a tabela. Esta é uma prática usada em sistemas de tempo real, quando as incertezas associadas com a alocação dinâmica de memória não são aceitáveis (por exemplo, o que fazer quando falta memória). Por outro lado, pode-se facilmente alterar a estrutura *segmento* para conter não um array de registros mas um apontador para array de registros. Desta forma, cada segmento pode ser alocado dinamicamente, conforme a necessidade da aplicação. A escolha entre alocação dinâmica e alocação estática é uma questão de projeto.

A inserção de um registro em um segmento não lotado exige no pior caso **S-1** deslocamentos. Para determinar qual segmento ainda tem espaço livre é feita uma pesquisa sequencial sobre o campo *ocupadas* de cada segmento. No pior caso, existe espaço livre somente no último segmento. O tempo máximo de uma inserção está associado com { K comparações de inteiros + (N / K) -1 deslocamentos }.

A pesquisa em um segmento não vazio funciona como a pesquisa binária convencional. No pior caso são necessárias $\lfloor \log S \rfloor + 1$ comparações. Substituindo **S** temos $\lfloor \log(N \div K) \rfloor + 1$. Entretanto, não é possível determinar a priori em qual segmento encontra-se a chave buscada. No

pior caso será feita uma pesquisa binária em cada um dos segmentos. Temos então que o tempo máximo de pesquisa está associado com { $K \times \left(\lfloor \log(N \div K) \rfloor + 1 \right)$ comparações de chaves }.

A diferença entre "comparações de inteiros" e "comparações de chaves" pode ser muito significativa. Por exemplo, considere as chaves do tipo "OS-999" usadas na tabela da figura 1. A comparação ocorre entre strings de tamanhos iguais onde os primeiros caracteres são iguais. Neste caso, a "comparação de chaves" é uma operação mais custosa que a "comparação de inteiros".

A solução proposta é na verdade uma mistura da pesquisa binária com a pesquisa sequencial. Isto fica óbvio quando consideramos dois casos extremos. Quando **K = 1**, temos um único segmento cujo comportamento é igual ao da pesquisa binária. Quando **K = N**, cada segmento contém um único registro e a pesquisa torna-se sequencial. A título de exemplo, considere uma tabela onde **N = 1024**. Aplicando as equações apresentadas antes, temos o seguinte custo computacional para inserção e pesquisa:

	Inserção	**Pesquisa**
Lista Desordenada	0 desl.	1024 comp.chave
Lista Ordenada	1023 desl.	11 comp.chave
Proposto, K=2	511 desl. + 2 comp.int.	20 comp.chave
Proposto, K=8	127 desl. + 8 comp.int.	64 comp.chave
Proposto, K=32	31 desl. + 32 comp.int.	192 comp.chave

A estrutura de dados proposta neste artigo, para uso em sistemas de tempo real, foi inspirada na estrutura de dados sugerida pelo exercício 18-2 do livro [COR 90]. Ambas oferecem uma mistura entre pesquisa sequencial e pesquisa binária. Entretanto, as duas apresentam diferenças importantes. A solução em [COR 90] utiliza sempre segmentos com tamanhos diferentes, estabelece regras rígidas para o tamanho dos segmentos e por vezes realiza a unificação e a divisão de segmentos. A solução proposta neste artigo não estabelece regras para o tamanho dos segmentos e jamais realiza qualquer unificação ou divisão de segmentos. É possível afirmar que a solução apresentada nesta seção é mais simples e fácil de programar do que aquela que aparece no livro antes citado.

4. Comparação entre as Implementações

A seção anterior propôs uma estrutura de dados que apresenta um desempenho misto entre lista ordenada e lista desordenada, com respeito as operações de inserção e pesquisa. Esta seção utiliza uma experiência prática para avaliar o tempo de execução destas três soluções. Elas foram implementadas na linguagem C e executadas em um computador com processador Intel 486, 66 Mhz e 16 Mbytes de memória principal. O programa foi compilado com o Borland Turbo C 2.0 e executado sobre MS-DOS 6.22.

A tabela implementada possui tamanho máximo de 2048 entradas e foi alocada estaticamente nos três casos. Cada entrada da tabela corresponde a um registro tipo `struct descritor_OS`, apresentado antes. O código da experiência garante que todos os registros utilizados possuem campo chave com valor diferente. As operações de inserção não são precedidas de uma pesquisa pela chave a ser inserida.

No teste DES (lista desordenada, pesquisa sequencial) foram feitas 2048 inserções, a partir de uma tabela vazia. Em seguida, foram feitas 2048 pesquisas usando como chave o campo chave do último registro inserido. Desta forma, a pesquisa sequencial realiza 2048 comparações, o pior caso.

No teste ORD (lista ordenada, pesquisa binária) também foram feitas 2048 inserções, a partir de uma tabela vazia. Os registros foram sendo inseridos na ordem inversa daquela na qual

ficariam na lista. Desta forma, para cada inserção todos os registros já presentes na lista devem ser deslocados, o pior caso. Também foram feitas 2048 pesquisas usando como chave o primeiro valor da lista, com o objetivo de forçar um grande número de comparações.

Os tempo total em segundos para 2048 operações de cada tipo, em cada teste, é apresentado abaixo. Em função da forma como o sistema operacional computa a passagem do tempo, existe uma margem de erro de 0.05 S. Os valores apresentados correspondem a média de 12 execuções.

DES (lista desordenada, pesq. sequencial) INSERÇÃO = 0,01 PESQUISA = 11,27
ORD (lista ordenada, pesquisa binária) INSERÇÃO = 12,16 PESQUISA = 0,07

No teste PRO (estrutura de dados proposta) novamente foram feitas 2048 inserções, a partir de uma tabela vazia. Os registros foram sendo inseridos na ordem inversa daquela na qual ficariam na lista. Desta forma, para cada inserção todos os registros já presentes no segmento em questão devem ser deslocados, o pior caso. Também foram feitas 2048 pesquisas usando como chave o primeiro valor do último segmento, com o objetivo de forçar a pesquisa em todos os segmentos e também forçar um grande número de comparações no último segmento.

O tempo total em segundos para 2048 operações de cada tipo, no teste PRO, é apresentado abaixo. Foram feitas experiências com o número de segmentos variando entre 1 e 2048. Novamente, existe uma margem de erro de 0.05 S. Os valores apresentados correspondem a média de 12 execuções.

K	**1**	**2**	**4**	**8**	**16**	**32**	**64**	**128**	**256**	**512**	**1024**	**2048**
Inserção	12.16	6.13	3.06	1.51	0.77	0.40	0.23	0.17	0.18	0.28	0.55	1.09
Pesquisa	0.08	0.16	0.28	0.50	0.88	1.49	2.56	4.12	6.26	8.61	17.31	18.60

Se considerarmos apenas a operação de inserção, uma lista desordenada é a melhor solução. Ela obteve um tempo de apenas 0.01 S. Este tempo é menor do que a implementação proposta obteve em seu melhor desempenho, com K=128, resultando em 0.17 S. No caso da lista ordenada, este tempo é muito maior, 12.16 S. Por outro lado, considerando apenas a operação de pesquisa, uma lista ordenada e pesquisa binária é a melhor solução. Ela obteve o tempo 0.07 S. Este tempo é menor do que a implementação proposta obteve em seu melhor desempenho, com K=1, resultando em 0.08 S. No caso da lista desordenada, este tempo é de 11.27 S.

A grande vantagem da solução proposta é oferecer um comportamento intermediário. Por exemplo, com K = 16 temos a inserção em 0.77 S e a pesquisa em 0.88 S. Considerando as duas operações em conjunto a solução proposta é melhor do que as duas outras analisadas.

O comportamento intermediário da solução proposta aparece claramente nas figuras 3, 4 e 5. A figura 3 mostra os tempos medidos para 2048 inserções na forma de um gráfico. No caso de implementação convencional, o valor medido não depende de **K**. Com K=1, a solução proposta usa apenas um segmento e, portanto, tem o comportamento semelhante ao da lista ordenada. Quanto maior o **K**, menores os segmentos, o que significa uma inserção mais rápida. Entretanto, quando **K** cresce além de determinado valor, o custo da pesquisa sequencial para encontrar um segmento com entrada livre torna-se significativo e o desempenho piora. No caso das condições deste teste, o desempenho para inserções foi máximo com K = 128.

A figura 4 mostra os tempos medidos para 2048 pesquisas. Com K = 1 a solução proposta faz uma única pesquisa binária sobre um segmento único com 2048 entradas. O resultado é um tempo praticamente igual ao obtido pela lista ordenada. Na medida em que **K** (o número de segmentos) cresce, ocorre um maior número de comparações. Com **K=N**, cada segmento possui apenas uma entrada, o que significa uma degradação para pesquisa sequencial. Entretanto, em

função da maior complexidade do algoritmo, o tempo acaba sendo quase o dobro daquele obtido por uma simples lista desordenada.

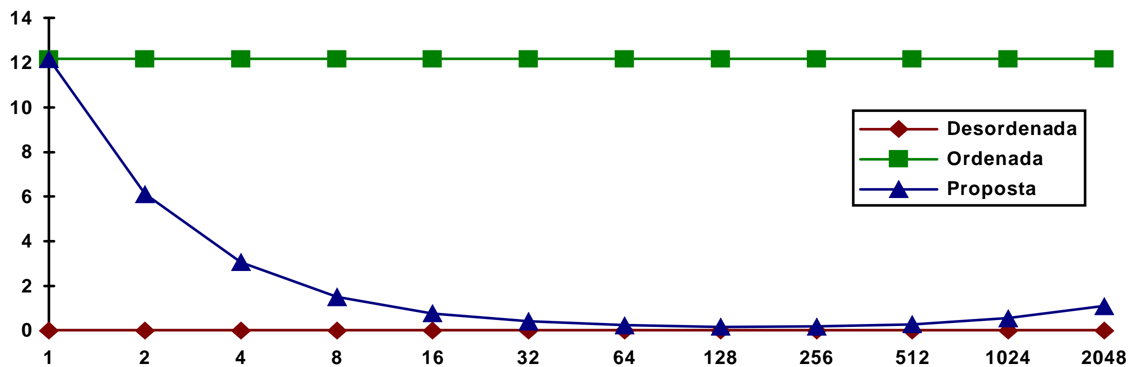

Figura 3 - Tempo medido em segundos para 2048 inserções em função de K.

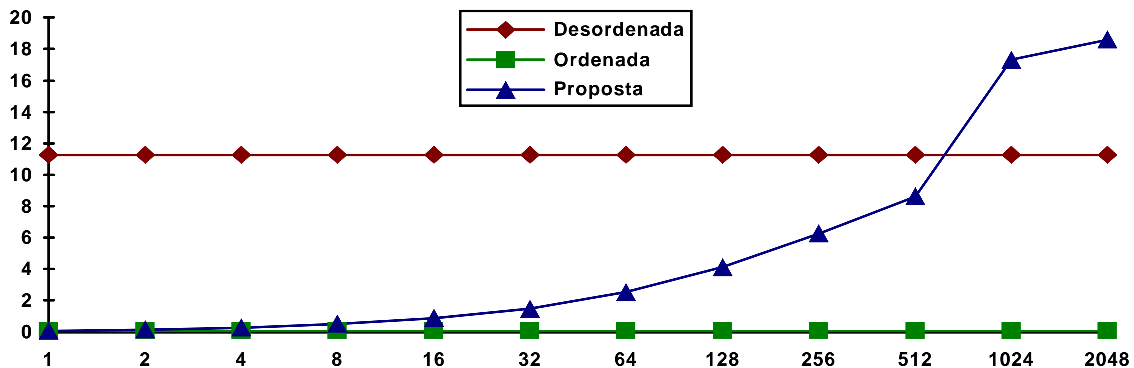

Figura 4 - Tempo medido em segundos para 2048 pesquisas em função de K.

Uma questão importante é a escolha do melhor valor para **K**. Esta decisão depende da aplicação e de suas restrições temporais. Caso o objetivo seja tornar os tempos de inserção e pesquisa semelhantes, a escolha pode ser feita através de uma simples inspeção do gráfico mostrado na figura 5. Ele mostra os tempos de inserção e pesquisa da solução proposta em função de **K**. Observa-se que os dois valores são semelhantes quando K=16. Seja qual for o objetivo da escolha, um gráfico como o da figura 5 deixa claro para o projetista as opções a sua escolha, permitindo um ajuste fino dos aspectos temporais do sistema.

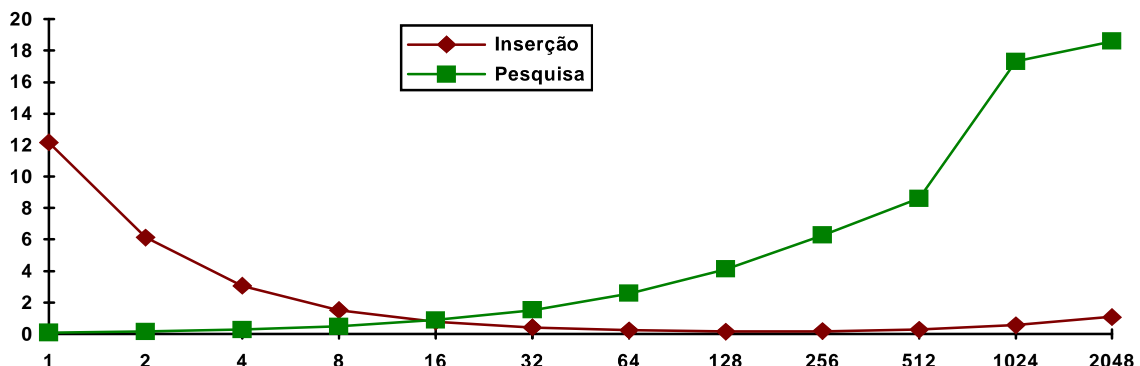

Figura 5 - Tempo medido em segundos para 2048 inserções e 2048 pesquisas na implementação proposta, em função de K.

5. Conclusões

Neste artigo foi analisada a implementação de tabelas em sistemas de tempo real. Especificamente, foi considerada a implementação de listas contíguas alocadas estaticamente. As operações consideradas neste artigo são a inserção e a pesquisa. Enquanto uma lista desordenada permite uma rápida inserção, o tempo de pesquisa é proporcional ao tamanho da lista. Ao mesmo tempo, uma lista ordenada permite uma rápida pesquisa, mas a inserção pode exigir o deslocamento de todos os elementos da lista.

O artigo propõe uma implementação com características mistas. A tabela é implementada através de vários segmentos, cada segmento correspondendo a uma lista ordenada. O resultado é um desempenho intermediário, onde tanto a inserção como a pesquisa são feitas em tempos razoáveis. Além disto, o número de segmentos é um fator de projeto que permite ao programador tornar uma ou outra operação mais rápida, conforme os requisitos da aplicação. O gasto adicional com memória na solução proposta é de apenas um inteiro por segmento, para conter o número de entradas ocupadas no segmento. Em geral, o valor de **K** tende a ser bem menor que **N**, o que significa gasto adicional pequeno em relação ao tamanho da tabela.

Foram feitas experiências com a estrutura de dados proposta, onde os tempos de execução da solução proposta, assim como das soluções tradicionais, foram medidos e comparados. Os resultados numéricos das experiências, apresentados na seção 4, comprovaram os resultados analíticos apresentados na seção 3.

Em sistemas de tempo real, o tempo de execução dos algoritmos é importante fator de projeto. Algoritmos simples são muitas vezes preferidos, buscando-se com isto confiabilidade e economia de memória. Neste sistemas, a escolha entre as duas formas tradicionais de listas se mostra problemática em função da enorme diferença entre os tempos de execução de duas operações básicas. A estrutura de dados apresentada neste artigo é simples, gasta pouca memória, mas oferece flexibilidade ao programador com respeito ao tempo de execução das operações. A solução proposta não só permite que as duas operações sejam executadas rapidamente, como também permite ao programador ajustar suas velocidades relativas. É possível que uma aplicação inviável no seu aspecto temporal torne-se viável com a substituição de uma implementação tradicional de tabela pela implementação proposta neste trabalho.

Uma questão em aberto é o estudo de implementações de tabelas que não utilizam alocação contígua. Principalmente árvores binárias e tabelas de dispersão (*hash tables*). Uma questão importante é determinar o comportamento de pior caso para estas estruturas de dados. Por exemplo, a tabela de dispersão apresenta um excelente tempo médio de pesquisa. Entretanto, dependendo de como as colisões são tratadas, seu comportamento de pior caso pode degradar para o de uma pesquisa sequencial.

Referências

[BEN 85] J. L. BENTLEY, C. C. McGEOCH. Amortized Analyses of Self-Organizing Sequential Search Heuristics. Communications of the ACM, vol. 28, no. 4, april 1985.

[COR 90] T. H. CORMEN, C. E. LEISERSON and R. L. RIVEST. Introduction to Algorithms. The MIT Press, 1990.

[HOR 87] E. HOROWITZ and S. SAHNI. Fundamentals of Data Structures in Pascal. Computer Science Press, 1987.

[SED 83] R. SEDGEWICK. Algorithms. Addison-Wesley Publishing Co., 1983.

[SHA 90] L. SHA, R. RAJKUMAR, J. P. LEHOCZKY. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Transactions on Computers, Vol. 39, No. 9, pp. 1175-1185, september 1990.

[STA 93] J. STANKOVIC, K. RAMAMRITHAM. Advances in Real-Time Systems. IEEE Computer Society Press, 1993.