Análise Experimental Comparativa de Algoritmos de Alocação de Memória de Código Aberto

Rivalino Matias Jr.[1], Autran Macêdo[1], Taís Borges Ferreira[2]

[1]Faculdade de Computação – Universidade Federal de Uberlândia

[2]Curso de Bacharelado em Ciência da Computação - Faculdade de Computação – Universidade Federal de Uberlândia

{rivalino,autran}@facom.ufu.br, taisbferreira@comp.ufu.br

Abstract. *Memory allocations are one of the most ubiquitous operations in computer programs. The performance of the allocators that implement these operations is very important for the overall performance although it is very often negligenced. This paper presents a comparative study of five general purpose and open source memory allocators. Unlike other related works, based on benchmark tests that are difficult to generalize to real-world applications, this work evaluates the selected allocators' performance combined with the MySQL software. The results demonstrate that MySQL shows the best performance when using the jemalloc allocator, especially when it is compared with the standard glibc allocator.*

Resumo. *Operações de alocação de memória estão entre as mais ubíquas em programas de computador. O desempenho dos alocadores que implementam essas operações é de suma importância para o desempenho global da aplicação, embora muitas vezes seja negligenciado. Esse trabalho apresenta um estudo comparativo entre cinco alocadores de memória de propósito geral e de código aberto. Diferente de outros trabalhos na área, baseados em testes de benchmark com difícil generalização para aplicações reais, esse trabalho avalia o desempenho dos alocadores no software MySQL. Os resultados mostram que o MySQL apresentou melhor desempenho em conjunto com o jemalloc, especialmente quando comparado com o alocador padrão da glibc.*

1. INTRODUÇÃO

Durante a execução de um programa de computador, a eficiência no uso da memória principal tem impacto significativo no desempenho do programa. De forma geral, aplicações do mundo real necessitam alocar e desalocar porções de memória, de tamanhos variados, por inúmeras vezes durante suas execuções. Essas operações de alocação e desalocação são comumente realizadas com grande frequência em programas mais sofisticados, o que faz com que o desempenho dessas operações influencie significativamente no desempenho global do sistema.

As operações de alocação de memória existem em dois níveis. No nível do núcleo do sistema operacional (*kernel level*) e no espaço do usuário (*user level*). No *kernel level* elas são implementadas por um alocador de memória chamado genericamente de KMA (*kernel memory allocator*) [Vahalia 1995], o qual tem como função o gerenciamento da memória para atender as necessidades dos subsistemas do próprio sistema operacional. No *user level* essas ope-

rações são implementadas por um alocador que fica armazenado em uma biblioteca padrão que é ligada ao programa durante a geração do seu código executável. No Linux, a *glibc*[1] é a biblioteca que armazena as rotinas e estruturas de dados do alocador de memória padrão em nível de usuário, genericamente chamado de UMA (*user-level memory allocator*). Os alocadores investigados nesse trabalho são do tipo UMA e serão chamados, desse ponto em diante, de alocador de memória ou simplesmente alocador.

Um alocador de memória tem como função primária o gerenciamento do *heap* [Vahalia 1995]. *Heap* é uma porção de memória localizada na região de dados do processo, a qual é usada para atender requisições de alocações dinâmicas de memória. Essas requisições são usualmente realizadas por meio das funções *malloc*(), *realloc*() e *free*(), que são implementadas como parte do alocador. Nos casos onde a requisição ultrapassa o tamanho disponível no *heap* do processo, o alocador pode solicitar mais memória para o sistema operacional a fim de atender a requisição. Essa porção de memória adicional é incorporada ao conjunto de páginas de memória do processo que é gerenciado pelo alocador. Desse modo, as rotinas e estruturas de dados do alocador são partes do processo e, portanto, podem ser substituídas pelo programador da aplicação.

Diversos sistemas não usam o alocador de memória disponível na biblioteca padrão, trazendo em seu código uma implementação própria de alocador. Isso ocorre porque usualmente a biblioteca padrão implementa um alocador de propósito geral, não sendo esse algoritmo otimizado para contemplar características específicas de cada aplicação que o utiliza. Dependendo da aplicação, as necessidades de alocação dinâmica de memória são bastante específicas e o alocador padrão não oferecerá um bom desempenho. Aspectos como o uso de múltiplos processadores e múltiplas *threads* de controle são exemplos de características que têm impacto significativo no desempenho das operações realizadas pelo alocador de memória [Berger *et al*. 2000]. Aplicações sofisticadas tais como o servidor web Apache, gerenciador de banco de dados PostgreSQL, e o navegador Firefox, são exemplos de programas que trazem seu próprio alocador em alternativa ao disponível na biblioteca padrão.

Atualmente, existem diversos algoritmos e implementações de código aberto para alocadores de memória, os quais podem ser usados como alternativa ao alocador da biblioteca padrão. A escolha de qual alocador oferece os melhores resultados para uma determinada aplicação deve ter com base uma avaliação experimental. Vários trabalhos (ex. [Zorn e Grunwald 1994], [Attardi e Nadgir 2003] [Masmano, Ripoll e Crespo 2006]) têm apresentado estudos sobre o desempenho de algoritmos de alocação dinâmica de memória. Contudo, a maioria dos trabalhos nessa área avalia os alocadores com base em programas de teste (ex. mtmalloctest [Attardi e Nadgir 2003]) que realizam diferentes operações de alocação de memória a fim de exercer, muitas vezes de forma aleatória, as rotinas e estruturas de dados do alocador avaliado. O problema com essa abordagem é que os resultados obtidos nesses testes dificilmente podem ser generalizados para uso em aplicações do mundo real. Nesse contexto, esse trabalho apresenta um estudo que compara, experimentalmente, cinco alocadores de memória de código aberto. Ao contrário dos trabalhos citados, nesse estudo foi utilizada uma aplicação real para avaliar o desempenho dos alocadores. A aplicação escolhida foi o gerenciador de banco de dados MySQL. A motivação para essa escolha é que gerenciadores de banco de dados são aplicações com exigentes requisitos de gerenciamento de memória dinâmica e também pelo fato do MySQL ser atualmente muito usado em aplicações web que exigem elevado desempenho.

O restante do artigo está organizado como descrito a seguir. A Seção 2 apresenta os alocadores selecionados para o estudo. Na Seção 3 são apresentados os detalhes envolvidos

[1] *www.gnu.org/software/libc/*

na etapa de experimentação, enfatizando a metodologia e os planos experimentais adotados. A Seção 4 apresenta uma análise dos principais resultados obtidos na etapa experimental. Finalmente, na Seção 5 são apresentadas as conclusões desse trabalho.

2. ALOCADORES INVESTIGADOS

Este trabalho compara o desempenho de cinco implementações de alocadores de memória de propósito geral: Hoard (versão 3.8), ptmalloc (versão 2), TCMalloc (versão 1.5), jemalloc (versão linux_20080827) e nedmalloc (versão 1.05). Esses alocadores foram selecionados devido à disponibilidade de código fonte, além de alguns deles terem sido usados em trabalhos correlatos que demonstram seu melhor desempenho frente a diversos outros alocadores, que por esse motivo foram desconsiderados nesse estudo.

O Hoard [Berger *et al*. 2000] foi desenvolvido com o principal objetivo de se ter alocação com alto desempenho em programas com múltiplas *threads* executando em computadores com múltiplos processadores. Esse algoritmo implementa uma área de *heap* global para todas as *threads*, além de uma área privativa para cada uma delas. Essas duas áreas possibilitam ao Hoard minimizar os efeitos de contenção de *heap* (múltiplas threads concorrendo pela mesma área de memória), além de um tipo específico de fragmentação de memória denominado *blowup* [Berger *et al*. 2000]. O Hoard também é reconhecido pela redução nos efeitos do falso compartilhamento de memória (isto é, quando *threads* executando em diferentes processadores compartilham equivocadamente a mesma linha de *cache*).

O ptmalloc [Gogler 2006] é um algoritmo baseado no DLMalloc [Lea 1996]. O ptmalloc incorpora recursos voltados para multiprocessadores executando programas com múltiplas *threads* de controle, os quais não eram amplamente utilizados quando seu antecessor DLMalloc foi desenvolvido. Tal qual o Hoard, o ptmalloc implementa múltiplas áreas de *heap* para diminuir a contenção de memória em programas *multithreaded*. Diferente do Hoard, esse não trata o problema de falso compartilhamento de memória. Atualmente, existem três versões do ptmalloc. A versão 2, que é o padrão na *glibc* do Linux, é o alocador considerado nesse trabalho.

O TCMalloc [Ghemawat e Menage 2010], diferente dos antecessores, busca minimizar a contenção de acesso ao *heap* usando uma abordagem híbrida. Para objetos pequenos (_ 32K), sua abordagem é similar aos anteriores, onde cada *thread* possui sua própria porção do *heap*. Contudo, para objetos grandes (_ 32K) o TCMalloc utiliza um *heap* global, minimizando a contenção entre as múltiplas *threads* por meio de *spin-locks* de fina granularidade. Segundo Ghemawat e Menage (2010), algoritmos baseados apenas em *heaps* locais por *threads* sofrem com excessiva utilização de memória e fragmentação.

O jemalloc [Evans 2006] foi desenvolvido para ser o alocador padrão no FreeBSD em substituição ao seu antecessor PHKalloc [Kamp 1998], que como o DLMalloc foi criado quando sistemas multiprocessados e *multithreading* não eram populares. Um dos principais objetivos desse alocador é reduzir a contenção de bloqueio em aplicações *multithreaded* executando em múltiplos processadores. Similar aos demais algoritmos, esse usa múltiplas porções do *heap* (*arenas*), por padrão quatro vezes o número de processadores. Cada *thread* é associada a um conjunto de *arenas*, onde uma *arena* é escolhida seguindo uma política *round-robin*, reduzindo assim a probabilidade de contenção. Os metadados de alocações muito grandes (_ 1M) são armazenados em uma única estrutura de árvore rubro-negra (*red-black tree*), e a suposição adotada pelos autores desse algoritmo é que por não ser freqüente esse tamanho de alocação, a manutenção de uma única estrutura para todas as *threads* não seria um problema para a escalabilidade do algoritmo.

O alocador nedmalloc [Douglas 2010], assim como o ptmalloc2, é baseado no DLMalloc. A diferença é que o ptmalloc2 é baseado na versão dlmalloc 2.7.0 enquanto o nedalloc é baseado na dlmalloc 2.8.3. O nedalloc estende o dlmalloc basicamente encapsulando um *cache* por *thread* (*per-thread lookaside cache*) para melhorar os acessos concorrentes ao *heap*.

3. ESTUDO EXPERIMENTAL

O plano experimental utilizado nesse estudo considera a execução de 30 replicações por teste, onde cada replicação tem duração de 10 minutos. Para eliminar os efeitos dos erros experimentais [Montgomery 2005], utilizou-se a média aritmética das replicações.

O instrumental usado contou com uma bancada de teste com a seguinte configuração: 3 computadores, onde um executa o MySQL versão 5.0.41 e os outros dois o software db_Stress [Kravtchuk 2007], o qual foi utilizado para gerar cargas de trabalho no servidor. O db_Stress foi desenvolvido para auxiliar na avaliação de desempenho de gerenciadores de banco de dados por meio da realização de diferentes tipos de transações em intervalos de tempo pré-determinados. As transações geradas pelo db_Stress são submetidas a um banco de dados constituído de cinco tabelas que simulam uma aplicação convencional. Essas tabelas possuem relacionamento entre si, índices e uma população de milhões de linhas. As transações utilizadas consistem em operações *select, insert*, *delete* e *update.* No total, são cinco transações que são executadas ininterruptamente por um período definido pelo usuário no momento da sua execução. Maiores detalhes sobre o modelo de dados e as transações utilizadas pelo db_Stess podem ser obtidos em [Kravtchuk 2007].

Ao final do período definido para o teste, o db_Stress reporta a quantidade de transações realizadas, e o tempo demandado por cada transação. O computador do servidor de banco de dados (Intel Core 2 Duo 2.40 GHz; 2 GB RAM) e os computadores clientes (Pentium 4 3.4 GHz; 1 GB RAM) executando o db_Stress foram interconectados por uma rede 100 Mbps. Tanto nos clientes quanto no servidor adotou-se o sistema operacional Linux Debian 5, kernel 2.6.26-1-686, executando em *runlevel 3*. A versão da *glibc* usada foi 2.7-18lenny2.

O MySQL foi testado com cada um dos cinco alocadores selecionados (ver Seção 2). Para isso, em cada teste do MySQL o ligador dinâmico (*dynamic linker*) do Linux foi instruído a carregar um dos alocadores avaliados. Isso é realizado usando a variável de ambiente LD_PRELOAD (ex. export LD_PRELOAD= libjemalloc_mt.so.0). Como resultado, o MySQL passa a usar as funções (malloc(), free(), etc.) de alocação da biblioteca compartilhada e não da *glibc*. Alternativamente, pode-se ligar (*linking*) estaticamente o alocador de interesse ao código do MySQL. Em ambos os casos, o desempenho do MySQL é influenciado pelo algoritmo de alocação ligado a ele no momento dos testes.

4. ANÁLISE DOS RESULTADOS

As Tabelas 1 a 4 apresentam a classificação do desempenho dos alocadores com respeito aos principais testes realizados. Os alocadores estão ordenados com o melhor desempenho no topo. Verifica-se que em todos os casos avaliados, a ordem de classificação foi a mesma. Como descrito na Seção 3, essa análise considera os valores médios das 30 replicações, onde cada replicação (teste) teve duração de 10 minutos.

Tabela 1. Transações para segundo	Tabela 2. Total de Transações	Tabela 3. Total de Leituras/Escritas	Tabela 4. Tempo Médio de select
Alocador Total	**Alocador Total**	**Alocador Total**	**Alocador Total**
jemalloc 2615,13	jemalloc 1569416,00	jemalloc 784708,23	jemalloc 1187,23
Hoard 2605,73	Hoard 1563745,00	Hoard 781872,50	Hoard 1193,67
TCMalloc 2602,07	TCMalloc 1561580,87	TCMalloc 780790,43	TCMalloc 1196,67
nedmalloc 2600,33	nedmalloc 1560463,60	nedmalloc 780231,80	nedmalloc 1196,77
ptmalloc2 2600,13	ptmalloc2 1560408,20	ptmalloc2 780204,10	ptmalloc2 1198,13

A Tabela 1 refere-se ao número de transações executadas por segundo durante um teste. Nesse teste são realizadas todas as operações (*select*, *insert*, *delete* e *update*). Na Tabela 2 tem-se o número total de transações executadas por teste, considerando as quatro operações citadas anteriormente. A Tabela 3 mostra a quantidade de operações de leitura (*select*) realizadas por teste. O db_Stress reportou os mesmos valores para operações de escrita (*insert*, *delete* e *update*). A Tabela 4 lista os tempos médios para operações do tipo *select*. Para as demais operações, as diferenças nos tempos das operações individuais não foram significativas, sendo o melhor algoritmo (jemalloc), em média, 2,5 microssegundos mais rápido do que o pior resultado (ptmalloc2).

Dentre os cinco alocadores avaliados, o jemalloc apresentou os melhores resultados em praticamente todos os testes. Em especial, sua maior influência no desempenho do MySQL se deu no número total de transações realizadas. Já o pior desempenho ficou com o ptmalloc2, o atual alocador padrão na *glibc*. Considerando que a maioria dos usuários utilizam o MySQL com o alocador padrão da *glibc*, esses resultados sugerem que uma melhor alternativa seria a utilização do jemalloc. O jemalloc realizou 15 transações por segundo a mais do que o ptmalloc2 (ver Tabela 1). Essa diferença em um período de operação, por exemplo, de 24 horas, seria de aproximadamente um milhão e trezentas mil transações. Considerando uma aplicação real, trata-se de uma diferença importante. Esse maior desempenho do jemalloc sobre o número de transações também explica os resultados das Tabelas 2 e 3. Com relação a operações individuais, vale destacar o resultado obtido para as operações de consulta (*select*). O jemalloc sobressaiu com uma diferença para o ptmalloc2 de aproximadamente 11 microsegundos por consulta (ver Tabela 4). Isso significa que em um segundo, o jemalloc consegue realizar 7,6 transações de consulta a mais do que o ptmalloc2. Ao final de 24 horas, por exemplo, essa diferença permitiria ao MySQL (com o jemalloc), executar aproximadamente 662,000 transações a mais do que com o ptmalloc2.

5. CONCLUSÃO

O desempenho de operações de alocação de memória tem significante influência no desempenho global da maioria das aplicações computacionais. Nesse sentido, a seleção de um alocador de memória é um importante requisito no projeto de sistemas mais sofisticados, sendo muitas vezes negligenciada pelos projetistas de software. A forte correlação entre o perfil de uso dinâmico da memória com o desempenho do alocador exige que a seleção do alocador ocorra por meio de um estudo experimental.

Nesse trabalho foi apresentado um estudo experimental comparativo entre cinco dos principais alocadores de memória, de propósito geral e de código aberto, disponíveis atualmente. Diferente de inúmeros trabalhos na área, os quais são baseados em testes de *benchmark*

com difícil generalização para aplicações do mundo real, esse trabalho comparou o desempenho dos alocadores avaliados com o popular gerenciador de banco de dados MySQL, muito usado em inúmeras aplicações reais. Verificou-se que a influência dos alocadores tanto na capacidade total de processamento de transações quando no tempo de operações individuais (ex. *select*) foi significativa, com destaque para o jemalloc, o qual ofereceu um aumento na capacidade de transações por segundo (TPS) na ordem de 15 TPS em relação ao ptmalloc2, utilizado atualmente pela *glibc* como alocador padrão no Linux.

REFERÊNCIAS

Attardi, J. and Nadgir, N. (2003) “A Comparison of Memory Allocators in Multiprocessors”, http://developers.sun.com/solaris/articles/multiproc/multiproc.html

Berger, E. D., McKinley, K.S., Blumofe, R.D. and Wilson, P.R. (2000) “Hoard: a scalable memory allocator for multithreaded applications”, 9th Int’l conf. on architectural support for programming languages and operating systems ACM SIGARCH Computer Architecture News, v.28:5, p.117-128

Douglas, N. (2010) “nedmalloc”, www.nedprod.com/programs/portable/nedmalloc.

Evans, J. (2006) “A Scalable Concurrent malloc() Implementation for FreeBSD”.

Ghemawat, S. and Menage, P. (2010) “TCMalloc: Thread-Caching Malloc”, http://goog-perftools.sourceforge.net/doc/tcmalloc.html.

Gloger, W (2006) “ptmalloc”, http://www.malloc.de/en/

Kamp PH (1998) “Malloc(3) revisited”. USENIX and FREENIX, p. 193–198.

Kravtchuk, D. (2007) “db_Stress Benchmark”, http://dimitrik.free.fr/db_STRESS.html

Lea, D. (1996) “A Memory Allocator,” http://gee.cs.oswego.edu/dl/html/malloc.html

Lever, C. and Boreham, D. (2000) “malloc() Performance in a Multithreaded Linux Environment”, In: Usenix 2000, p.301-311.

Masmano, M., Ripoll, I. and Crespo, A. (2006) “A comparison of memory allocators for real-time applications”, Proc. of 4th Int’l workshop on Java technologies for real-time and embedded systems, ACM Int’l Conference Proceeding Series, vol. 177, p.68-76

Montgomery, D.C. (2005) Design and Analysis of Experiments, 6 ed., J. Wiley & Sons.

Vahalia, U. (1995) UNIX Internals: The New Frontiers, Prentice Hall.

Zorn, B. and Grunwald, D. (1994) “Evaluating models of memory allocation”, ACM Transactions on Modeling and Computer Simulation (TOMACS), vol.4:1,p.107-131