Análise Experimental de Alocadores de Memória em Nível de Kernel

Taís B. Ferreira, Rivalino Matias, Autran Macêdo, Bruno Evangelista
Faculdade de Computação
Universidade Federal de Uberlândia
Uberlândia-MG, Brasil
taisborgesferreira@gmail.com, rivalino@fc.ufu.br, autran@fc.ufu.br, bruno_evangelista@si.ufu.br

***Resumo* — O gerenciamento da memória principal é uma das tarefas mais importantes de um sistema operacional, pois tem impacto direto no desempenho e disponibilidade dos sistemas computacionais. O elemento responsável por esse gerenciamento é o alocador de memória do *kernel*. Problemas inerentes ao funcionamento de alocadores de memória em nível de *kernel* são a fragmentação e o sobreuso (*overhead*) de alocação, entre outros. Esses problemas são intrínsecos ao processo de alocação dinâmica de memória e se fazem notar, preponderantemente, em sistemas que realizam longas execuções ininterruptas. Minimizar ou contornar esses problemas é um desafio para os alocadores de memória em nível de *kernel*. Este trabalho apresenta um estudo experimental comparativo de quatro alocadores de memória em nível de *kernel* (SLAB, SLOB, SLUB, e SLQB). Essa comparação considerou o tempo de execução, consumo de memória e o nível de fragmentação da memória de cada alocador, submetido a uma carga de trabalho padrão. Considerando os achados obtidos neste estudo experimental, conclui-se que os alocadores que apresentaram melhor desempenho geral foram o SLOB e SLAB, com vantagem para o primeiro.**

Palavras-chave* — *alocador de memória; kernel, sistema operacional

I. INTRODUÇÃO

O gerenciamento da memória principal tem impacto significativo no desempenho e disponibilidade dos sistemas computacionais. Este gerenciamento é realizado em dois níveis do ponto de vista do sistema operacional (SO), tanto no *user level* quanto no *kernel level*. Em ambos os níveis o conjunto de rotinas responsável por esta tarefa é denominado alocador de memória.

O alocador de memória do espaço do usuário é chamado de UMA (*user level memory allocator*) [1] e tem por objetivo atender as necessidades de alocação dinâmica de memória dentro do processo de aplicação. Para isso, o UMA gerencia o espaço de endereçamento na *heap* do processo. Cada processo possui o seu UMA que, usualmente, é parte da biblioteca padrão do sistema (ex. *libc*); portanto, este é ligado ao código do programa durante a geração do seu executável (*linking stage*). Alguns programas implementam seu próprio UMA.

Já um alocador de memória do *kernel*, denominado KMA (*kernel level memory allocator*) [1], é a parte do núcleo do SO que atende as requisições de alocação dinâmica dos seus subsistemas (ex. *device drivers* e *system calls*), bem como provê espaço de endereçamento para os processos de aplicação (regiões de dados, código, pilha e *heap*). Diferente do UMA, que pode variar por processo, o KMA é único para todo o sistema operacional, atendendo tanto as demandas do próprio *kernel* quanto dos processos de aplicação, neste último caso fornecendo espaço de memória para o UMA de cada processo. A Fig. 1 ilustra a interação de ambos os tipos de alocadores.

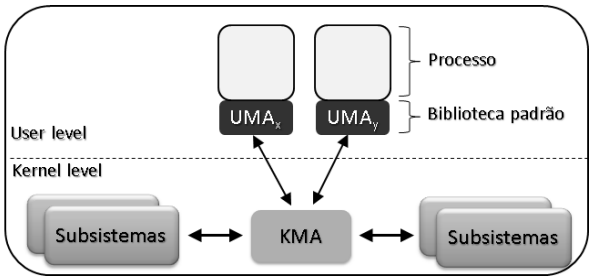

Fig. 1. Interação KMA e UMAs.

Em [2], avaliou-se experimentalmente o desempenho de sete implementações de alocadores do tipo UMA, considerando o tempo de resposta, consumo de memória e taxa de fragmentação da memória. Esta avaliação foi realizada utilizando uma aplicação *middleware* usada na indústria. Em [3], foi realizada a análise teórica dos algoritmos de cada UMA avaliado em [2], considerando suas complexidades de espaço e tempo. A fim de complementar ambos os trabalhos citados anteriormente, em [4] foi apresentado um estudo teórico-experimental sobre os sete alocadores estudados previamente, usando um gerador de carga sintética que permitiu avaliá-los sob diferentes cenários de uso, considerando fatores como número de alocações, tamanho das alocações, número de *threads* e número de processadores. Dando continuidade nas pesquisas nesta área, o presente trabalho é voltado para alocadores do tipo KMA. Até onde foi possível averiguar, não foi encontrado trabalho similar ao aqui apresentado.

Este artigo apresenta um estudo experimental comparativo de quatro alocadores do tipo KMA. Os alocadores investigados foram: SLAB, SLOB, SLUB e SLQB. Estes são KMAs disponíveis atualmente no *kernel* Linux e são baseados no *slab allocator* proposto em [5]. As demais seções deste trabalho

estão organizadas como segue. Na Seção II são apresentados a metodologia e instrumental adotados. Na Seção III são descritos os principais aspectos internos de cada KMA analisado. A Seção IV apresenta os resultados do estudo experimental e a Seção V as conclusões do trabalho.

II. METODOLOGIA

Inicialmente, os quatro alocadores investigados neste trabalho foram estudados quanto às suas estruturas de dados e rotinas de gerenciamento de memória. Esse estudo foi apoiado, principalmente, na análise do código fonte dos alocadores, pois é bastante limitada a literatura nesta área, em especial abordando detalhes de implementação. Como resultado, uma síntese das principais características de cada alocador é apresentada na Seção III. Posteriormente, foi analisado o comportamento dinâmico dos alocadores, tendo como base o estudo experimental apresentado na Seção IV. Nesta etapa, foi utilizado o *SysBench*, um *benchmark* voltado para avaliar o desempenho do SO, para cargas de trabalho voltadas para I/O de arquivos, transferência de dados em memória, entre outras. Seu uso objetivou exercitar o subsistema de gerenciamento de memória do sistema operacional, que foi instrumentado com cada um dos quatro KMAs analisados. O SO usado neste estudo foi o *kernel* Linux (versão 2.6.39), cujo atual KMA padrão é o SLUB. Dado que um KMA é único para o SO, foram geradas quatro versões diferentes do *kernel* Linux, uma para cada KMA, a fim de avaliar o desempenho de cada alocador sob a mesma carga de trabalho gerada pelo *SysBench* (ver Seção II.A). Os critérios de desempenho considerados foram o tempo de execução das operações do *SysBench*, o consumo de memória dentro do *kernel* e o nível de fragmentação da memória principal. O tempo de execução foi obtido diretamente do *SysBench*. O consumo de memória do SO foi medido com um *shell script* que monitorou o valor das variáveis *LowTotal* e *LowFree*, no arquivo /proc/meminfo, a cada dez segundos, durante a execução *SysBench*. A fragmentação foi medida com o *SystemTap* (ver Seção II.B), o qual contou os eventos de fragmentação da memória principal durante a execução do *SysBench*. Os experimentos foram realizados em um computador com processador Dual-Quad Core, 24 GB RAM.

A. SysBench

SysBench [6] é um programa multiplataforma e *multithreaded* desenvolvido para avaliar parâmetros do sistema operacional, em especial com foco para sistemas servidores. Dentre as cargas de trabalho implementadas pelo *SysBench*, neste estudo foram usadas cargas correspondentes aos modos *memory* e *fileio*. O modo *memory* executa a transferência de um conteúdo da memória de uma posição para outra, onde o conteúdo transferido corresponde a valores do tipo inteiro. O destino da transferência é um vetor alocado antes do lançamento das *threads* que realizam a transferência, de forma concomitante, ou seja, compartilhando o vetor. A quantidade de *threads* é um parâmetro do *SysBench*. Cada *thread* executa um *loop*, dentro do qual faz 128 destas transferências de memória, até que a quantidade de memória transferida atinja 100 *gigabytes*. O segundo modo de carga de trabalho usado foi o *fileio*, o qual é executado em três etapas distintas: *prepare*, *run* e *cleanup*. Na etapa *prepare* o *SysBench* cria 128 arquivos de dezesseis *megabytes*. Na etapa *run* este aloca um vetor de *file descriptors* com 128 posições e então abre cada um dos arquivos criados na etapa *prepare*, atribuindo seus respectivos *file descriptors* a uma posição do vetor. Posteriormente lança as *threads*, onde cada uma executa um *loop* dentro do qual escolhe um dos arquivos abertos de forma aleatória e escreve dezesseis *kilobytes* em alguma posição do arquivo até que o número máximo de operações seja atingido. Tanto o número de *threads* quanto o máximo de operações são parâmetros informados na execução da etapa *run*. A última etapa, *cleanup*, apenas exclui os arquivos criados na etapa *prepare*. Ambas as cargas usadas neste trabalho (*memory* e *fileio*) foram previamente identificadas [7] como propensas à produção de eventos de fragmentação de memória, o que é de interesse para este estudo.

B. SystemTap

O *SystemTap* [8] é uma ferramenta de rastreamento de eventos, a qual permite inserir diferentes pontos de monitoramento dentro do *kernel* Linux sem que haja alteração e recompilação do código fonte, ou seja, isso é feito em tempo de execução. Esta ferramenta oferece, por meio de uma linguagem de *script* própria, a infraestrutura necessária para monitorar as atividades de gerenciamento de memória dentro do *kernel*, durante sua execução. Por este motivo o *SystemTap* é usado não apenas como ferramenta para depuração do código do *kernel*, mas também para realizar variadas medições de desempenho. Neste trabalho, o uso do *SystemTap* foi dirigido à coleta de eventos de fragmentação dentro do *kernel*, observados durante a execução dos experimentos com as cargas de trabalho do *SysBench*. Este procedimento é descrito em detalhes em [7].

III. ALOCADORES INVESTIGADOS

Nesta seção são apresentados os detalhes de implementação dos KMAs analisados neste trabalho. Todos os quatro alocadores possuem características comuns, as quais são descritas na Seção III.A. As características específicas de cada alocador são apresentadas nas demais seções (III.B até III.E), respectivamente.

A. Características Comuns dos Alocadores Analisados

Quando um subsistema do *kernel* Linux necessita alocar memória, dinamicamente, para um dado objeto, como um *inode* ou um *dentry*, ou precisa liberar um objeto que não está mais em uso, tal subsistema faz uso das rotinas *kmalloc*() e *kmem_cache_alloc*(), *kfree*() e *kmem_cache_free*() [9]. O KMA é o código responsável por implementar essas rotinas. Para esta implementação, os alocadores estudados neste trabalho têm como característica comum o uso de uma estrutura de dados conhecida como *slab*. Um *slab* é uma porção de memória que tipicamente tem o tamanho de uma página de memória (ex. 4096 bytes), ou múltiplo de uma página, dependendo da implementação do KMA. Assim, KMAs baseados nessa estrutura são denominados *slab allocators*. Considerando a arquitetura do *kernel* Linux, o KMA se situa sobre o alocador primário de páginas (*page-level allocator* – PLA), sendo parte integrante do subsistema de

gerenciamento de memória virtual. A Fig. 2 ilustra esse posicionamento. O sentido das setas indica o destinatário do resultado da respectiva chamada. Por exemplo, um subsistema do *kernel* (ex. *device driver*) recebe um *slab* via *kmalloc*(), o qual é retornado pelo KMA. Para a implementação do seu conjunto de *slabs*, o KMA solicita páginas de memória junto ao PLA, via *alloc_pages*(). Na medida em que os *slabs* tornam-se livres, ou seja, deixam de ser usados, eles são liberados via *kfree*() e suas respectivas páginas podem retornar ao PLA, neste caso via *free_pages*().

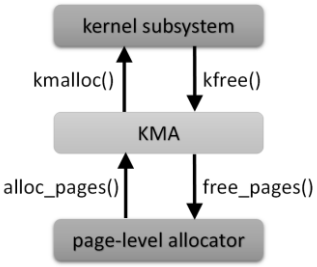

Fig. 2. KMA e demais subsistemas no *kernel* Linux.

De forma geral, outro ponto comum entre os KMAs é a existência de uma política específica para tratar o acesso à memória em máquinas NUMA (*Non-Uniform Memory Access*). Nesse tipo de computador, a memória principal está dividida em bancos de memória, em geral conhecidos como nós, que podem estar associados a um ou mais processadores/núcleos. O tempo de acesso varia de acordo com a distância que o banco de memória está de cada processador.

Nas próximas seções são apresentadas as características específicas de cada KMA analisado. Os alocadores são SLAB, SLOB, SLUB e SLQB. Os três primeiros são parte integrante do *kernel* padrão (*mainstream*) do Linux e podem ser selecionados para uso durante a compilação do *kernel*. O quarto alocador encontra-se disponível como um *patch* a partir da versão 2.6.30 do *kernel* Linux.

B. SLAB

O SLAB [5] foi o KMA padrão do *kernel* Linux entre as versões 2.2 e 2.6.23. Esse alocador é fortemente baseado no alocador originalmente proposto para o sistema operacional SunOS, descrito em [5], o qual se baseia em diferentes *caches* de *slabs*. Cada *slab* consiste de uma ou mais páginas de memória para acomodar objetos (blocos de memória) dinamicamente alocados. Cada *cache* armazena objetos de mesmo tamanho. Em cada *cache*, os *slabs* são organizados em três listas: *full*, *partial* e *free* (ver Fig. 3). A lista *full* armazena os *slabs* que não possuem um único objeto livre e, portanto, não podem ser utilizados para atender uma requisição de memória. A lista *partial* mantém *slabs* que possuem pelo menos um objeto alocado e no mínimo um objeto livre; é essa lista que preferencialmente é utilizada para satisfazer requisições de memória. A lista *free* armazena qualquer *slab* que tenha se tornado totalmente vazio, ou seja, não possui objetos. Essa lista é utilizada para satisfazer uma requisição de memória somente quando a lista *partial* está vazia. Uma *cache*, tal como descrita anteriormente, é denominada de *cache* de uso geral e armazena diferentes tipos de objetos. Esse tipo de *cache* é utilizada por subsistemas do *kernel* que requisitam uma porção de memória via *kmalloc*(). O parâmetro dessa função é o tamanho do objeto a ser alocado. Assim, o alocador utiliza o tamanho para decidir qual de suas *caches* de uso geral irá retornar a porção de memória solicitada. As *caches* de uso geral são criadas automaticamente pelo SLAB. Por outro lado, é possível criar uma *cache* para um tipo específico de objeto (ex. *dentry*). Essa *cache* específica é criada explicitamente por meio da rotina *kmem_cache_create*(). É necessário, entretanto, especificar um construtor e destrutor a ser usado com tal tipo de objeto. O construtor indicado na criação destas *caches* é utilizado apenas quando o alocador busca mais páginas de memória para a *cache* e divide estas páginas em novos objetos. O destrutor será chamado apenas quando o alocador decidir devolver páginas de memória para o PLA. Desta forma, não haverá necessidade de fazer chamadas do construtor de um objeto enquanto houver memória na *cache*, nem chamadas ao destrutor deste objeto, o que melhora o tempo de resposta das operações de alocação e desalocação de memória. A criação, manutenção e definição das políticas de acesso a uma *cache* específica são de responsabilidade do subsistema que a criou. Por exemplo, no subsistema de gerenciamento de arquivos, como há necessidade de alocar, frequentemente, algumas estruturas específicas, este subsistema se responsabiliza por criar as *caches* que necessita, tais como as *caches dentry* e *inode_cache*. Um subsistema que necessita requisitar um objeto a uma *cache* específica deve fazer uso da função *kmem_cache_alloc*(), passando como parâmetro o endereço da *cache* de interesse. A política do SLAB para máquinas NUMA é criar as três listas de *slabs* (*full*, *partial* e *free*) para cada nó encontrado. Assim, quando o SLAB percebe que a requisição que chegou a uma dada *cache* é proveniente de um determinado nó, ele a direciona para a lista *partial* deste nó.

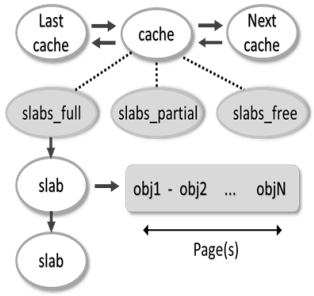

Fig. 3. Organização da *cache* no SLAB.

C. SLOB

O alocador SLOB [10] foi desenvolvido para sistemas com limitação de memória. Esse alocador foi introduzido na versão 2.6.14 do *kernel* Linux. Este é o alocador recomendado quando se está compilando o *kernel* para sistemas embarcados. Apesar de implementar as interfaces de alocação do SLAB, tais como *kmalloc*() e *kmem_cache_alloc*(), ele apenas simula o uso das *caches*. O SLOB mantém listas encadeadas de páginas que são utilizadas para atender requisições por objetos menores do que uma página (4096 bytes). As listas são organizadas de acordo com o tamanho dos objetos, como ilustrado na Fig. 4. A lista *small* mantém objetos menores do que 256 bytes; a lista *medium* mantém objetos com tamanho entre 256 e 1023 bytes;

e a lista *large* mantém objetos com tamanho entre 1024 e 4095 bytes. A organização em apenas três listas tem como objetivo reduzir o consumo de memória provocado pelas *caches* e listas do alocador SLAB. No entanto, armazenar diferentes tamanhos de objetos em uma mesma página, dependendo do comportamento da aplicação, pode acarretar fragmentação interna, o que resulta no aumento de consumo de memória. Se a porção de memória requisitada para um objeto for maior ou igual a uma página, o SLOB requisita um conjunto de páginas suficiente para conter o tamanho requisitado, diretamente do PLA. A liberação de tais objetos provoca o retorno das respectivas páginas diretamente ao PLA. Por fim, a política do SLOB para máquinas NUMA é simplesmente entregar um bloco de memória que pertença a uma página proveniente do nó especificado.

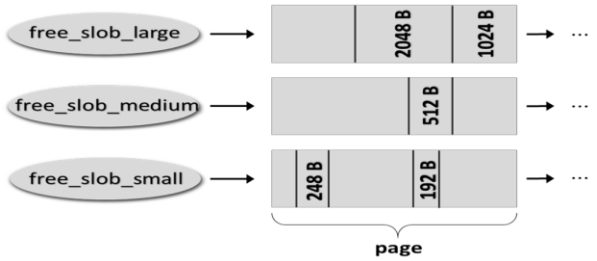

Fig. 4. Listas de páginas do SLOB.

D. SLUB

O alocador SLUB [11] tornou-se disponível a partir da versão 2.6.22 do *kernel* Linux [12]. Esse KMA foi proposto para corrigir problemas observados no alocador SLAB, tais como de escalabilidade, complexidade do código fonte e sobreuso (*overhead*) de memória, causados por metadados de gerenciamento de memória. Assim como o SLAB, o SLUB é organizado em *caches*, onde cada *cache* mantém vários *slabs*; cada *slab* tem o tamanho de uma página e reúne objetos do mesmo tamanho. O SLUB também classifica seus *slabs* como *full*, *partial* e *free*. No entanto, apenas a lista *partial* é realmente mantida. Quando um *slab* se torna *full*, esse *slab* é removido da lista de *slabs* e ignorado pelo alocador até que um de seus objetos seja liberado. Quando um *slab* se torna *free*, sua página é imediatamente devolvida ao PLA. Com esta medida o SLUB reduz a quantidade de páginas que mantém, reduzindo assim o consumo de memória. Focando ainda no baixo consumo de memória, o SLUB não implementa estruturas de controle por *slab* e nem metadados sobre objetos. A página que compõe um *slab* possui um ponteiro para o primeiro objeto da lista de objetos livres dentro do *slab* e um contador que indica o número de objetos do *slab* que já foram alocados, conforme ilustrado na Fig. 5.

Quando ocorre uma alocação, o contador *inuse* é incrementado, o primeiro objeto da lista de livres é retornado e o ponteiro *freelist* é atualizado. Na liberação de um objeto, essas operações são realizadas na ordem inversa. Finalmente, a política do SLUB para máquinas NUMA é manter uma lista de *partial slabs* por nó. Além disso, cada processador/core mantém um *slab* exclusivo (*active slab*), eliminando a necessidade de mecanismos de exclusão mútua. Este é o primeiro *slab* consultado pelo alocador quando ocorre uma requisição (alocação ou liberação).

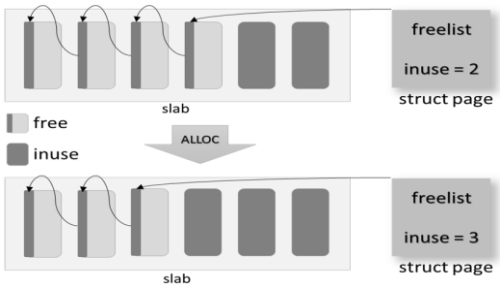

Fig. 5. Alocação de um objeto no SLUB.

E. SLQB

O alocador SLQB [13] utiliza várias ideias implementadas nos alocadores SLAB e SLUB, mas possui uma estrutura diferente para tratar seus *slabs*. O SLQB implementa uma *cache* (abstrata) que mantém objetos de tamanho fixo. Objetos são requisitados via *kmem_cache_alloc*() ou *kmalloc*(), tal que esta última é apenas um redirecionamento para a primeira. A estrutura da *cache* é ilustrada na Fig. 6. A estrutura *kmem_cache* mantém vários parâmetros globais, tais como o tamanho dos objetos (*size*), o nome da *cache* (*name*) e a ordem de alocação das páginas (*order*). Cada processador/*core* tem a sua própria *kmem_cache_cpu*. Requisições por objetos são atendidas, inicialmente, pelos objetos disponíveis na lista *freelist*. Essa lista é gerenciada como uma pilha (novas requisições usam objetos recém-liberados) para otimizar o uso da memória *cache*. Quando ocorre uma requisição e não há objetos livres na *freelist*, o SLQB requisita uma nova página do PLA. A nova página é adicionada à lista *partial* e o objeto, para atender à requisição, é retirado dessa página. Outros objetos podem ser retirados dessa página, mas somente quando a *freelist* estiver vazia. Eventualmente, os objetos retornam, via *kmem_cache_free*() ou *kfree*(), à *freelist*. Quando o tamanho dessa lista alcança um limiar (definido por *hiwater*), os objetos são devolvidos para suas páginas na lista *partial*. Essas páginas por sua vez são devolvidas ao PLA, quando completamente preenchidas com objetos livres. Contudo, um objeto da *freelist* pode ter sido alocado originalmente em outro *core*, nesse caso, então, o objeto é movido para a lista *rlist*. Quando o tamanho dessa lista alcança um limiar (definido por *freebatch*), o SLQB percorre a lista *rlist* de cada *core*, movendo cada objeto para a lista *remote_free* do *core* no qual o objeto foi alocado originalmente. Os objetos permanecem na *remote_free* até que um limiar (definido por *remote_free_check*) seja atingido. Neste caso, objetos dessa lista são transferidos para a *freelist* ou para as páginas de origem (na lista *partial*).

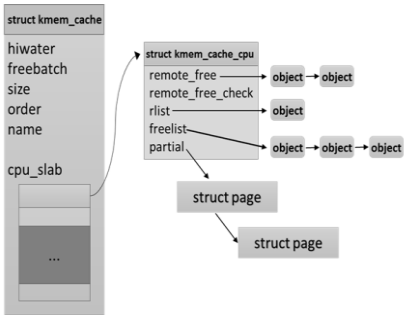

Fig. 6. Estruturas do SLQB.

IV. AVALIAÇÃO EXPERIMENTAL

Como informado na Seção II, foram criadas quatro instâncias do *kernel* Linux, cada uma sendo o resultado da compilação da versão 2.6.39 configurada com um dos quatro KMAs apresentados na Seção III. Um experimento consistiu na execução do *SysBench*, nos modos de operação *fileio* e *memory*, com cada instância compilada do *kernel* e variando o número de processadores (*cores*) de 1 até 8. Cada experimento foi repetido cinco vezes e seus resultados foram analisados com base na média das repetições. Portanto, foram realizadas 160 execuções considerando todas as combinações. Salienta-se que o *SysBench* foi parametrizado para executar 64 *threads* e realizar 100 mil operações de escrita. Ao final de cada conjunto de cinco repetições do mesmo experimento, o computador foi reiniciado a fim de configurar o novo cenário de teste.

A. Resultados do SysBench no Modo "fileio"

A Fig. 7 mostra o tempo médio de execução do *SysBench*, em modo *fileio*, sob cada KMA. O menor tempo de execução foi obtido com o SLOB em um *core*. Nenhum dos alocadores foi predominantemente melhor ou pior. Os achados experimentais sugerem que o SLQB melhora seu desempenho a partir de quatro *cores*. Já o SLUB foi o alocador que apresentou maior variabilidade nos resultados entre os diferentes números de *core*, ao contrário do SLOB que apresentou a menor variabilidade, sendo este também o alocador com menor tempo médio (369,88s), seguido do SLAB (378,88s), SLUB(382,13s) e SLQB (384,50s).

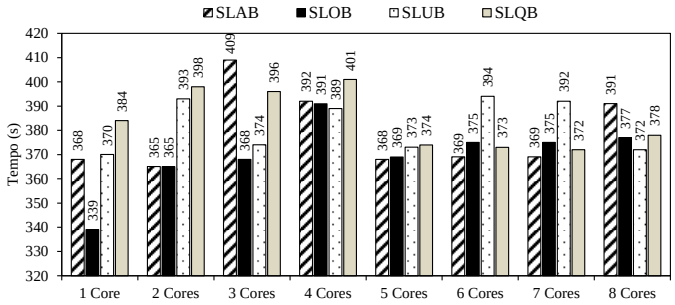

Fig. 7. Tempo médio de execução do *SysBench* no modo *fileio*.

Ressalta-se que o tempo de execução reportado pelo *SysBench* corresponde ao tempo da etapa *run*. Desse modo, as etapas *prepare* e *cleanup* não foram considerados. No modo *fileio*, a criação dos arquivos se dá na etapa *prepare* e a exclusão na etapa *cleanup*, portanto, os valores apresentados não contemplam essas duas operações.

Os resultados para consumo e fragmentação de memória foram obtidos considerando as três etapas do modo *fileio*. O consumo de memória do *kernel* é apresentado na Fig. 8. O eixo *y* representa o consumo da memória no *kernel* Linux, em *megabytes*, após a execução do *SysBench*. Nota-se que o *kernel* compilado com o SLQB demandou significativamente mais memória do que os demais, independente da quantidade de *cores*. A diferença foi superior a 100 *megabytes*. Similar ao tempo de execução, o menor valor de consumo de memória foi obtido com o SLOB em 1 *core*. Observa-se que, em média, o consumo de memória do SLOB (186,75 MB) e SLUB (188,25 MB) foi muito próximo, com ligeira vantagem para o SLOB. Convém ressaltar que em nível de *kernel* uma diferença de alguns *megabytes* é significativa, principalmente em se tratando de sistemas com restrições de memória, tal como sistemas embarcados. Neste caso, a diferença entre SLOB e SLUB foi de 1,5 MB. O SLUB é seguido pelo SLAB (193,13 MB) e SLQB (291,50 MB).

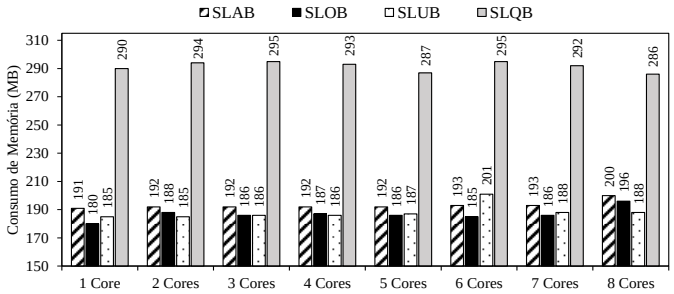

Fig. 8. Consumo de memória do *kernel* após o *SysBench* no modo *memory*.

A Fig. 9 apresenta o número médio de eventos de fragmentação de memória capturados com o *SystemTap* durante a execução do *SysBench*. Diferente dos resultados anteriores, o pior desempenho foi do alocador SLOB, independentemente do número de núcleos. Este foi seguido pelo SLQB com o segundo pior desempenho, principalmente com 1 até 4 *cores*. No geral, o melhor desempenho foi obtido com o SLAB, apresentando baixo nível de fragmentação; em algumas execuções este alocador não gerou fragmentação. O segundo melhor resultado foi observado com o SLUB.

A carga de trabalho exercida pelo modo *fileio* tem como principal característica maior número e variedade de operações em arquivos, sendo típica de sistemas servidores de banco de dados, arquivos, web, entre outros correlatos. Os achados experimentais reportados nesta seção sugerem que o SLQB não é adequado para este tipo de carga de trabalho, apresentando o maior tempo de resposta, maior consumo de memória e ocorrências de fragmentação de memória. Já o SLOB demonstrou os melhores resultados para tempo de execução e consumo de memória, porém tendo sido o alocador com maior número de eventos de fragmentação. Para sistemas que não executam de forma ininterrupta por longos períodos de tempo, onde a fragmentação deixa de ser uma preocupação, o SLOB seria a opção recomendada. Para os demais casos, então o SLAB e o SLUB se mostram as melhores alternativas, cuja escolha dependeria das necessidades em termos de tempo de resposta, consumo de memória e número de processadores.

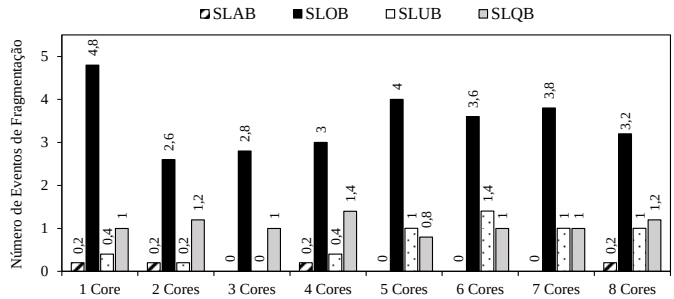

Fig. 9. Média do número de eventos de fragmentação.

B. Resultados do Sysbench no Modo "memory"

O *SysBench* no modo *memory* realizou a transferência de 100 *gigabytes* entre posições de memória. Como a transferência praticamente não gera requisições de alocação de memória, não houve evento de fragmentação em nenhuma das instâncias, independente do número de *cores*. Quanto ao tempo de transferência, observou-se que esse sofre redução na medida em que o número de *cores* varia de um até quatro. A partir de cinco *cores* há aumento pouco significativo nesse tempo, como pode ser observado na Tabela I. Cada linha dessa tabela refere-se ao tempo (em segundos) de transferência demandado por cada KMA analisado. As colunas referem-se à quantidade de *cores*. De dois a cinco *cores* o melhor tempo, nesta ordem, foi obtido por SLAB, SLOB, SLUB e SLQB. A partir de sete *cores* esta ordem é invertida. Os resultados de tempo médio de execução dos testes no modo *memory* com cada um dos alocadores foram, em geral, bem próximos uns dos outros. Com relação ao consumo de memória (em megabytes) pelo *kernel*, a diferença entre os KMAs é mais destacada (ver Tabela II). Observa-se que, para cada KMA, a quantidade de memória consumida se manteve praticamente constante, independente da quantidade de *cores* (colunas da Tabela II). O SLQB foi o KMA que mais consumiu memória. Os alocadores SLAB e SLUB apresentaram resultados de consumo de memória semelhantes, uma vez que implementam estruturas de dados muito parecidas. O alocador SLOB, dentre os analisados, é o que apresentou o melhor resultado de consumo de memória de *kernel* para esse teste.

TABELA I. TEMPO MÉDIO DE EXECUÇÃO (*SYSBENCH / MEMORY*)

	1	2	3	4	5	6	7	8
SLAB	178,6	95,9	67,4	50,9	55,8	62,2	67,3	67,4
SLOB	162,1	99,6	70,0	53,0	56,7	61,1	66,8	67,0
SLUB	191,7	102,3	72,1	54,6	57,0	61,5	66,3	66,8
SLQB	204,5	105,1	73,7	55,9	57,3	61,6	64,8	65,7

TABELA II. CONSUMO DE MEMÓRIA NO KERNEL (*SYSBENCH / MEMORY*)

	1	2	3	4	5	6	7	8
SLAB	138	139	139	139	140	141	141	142
SLOB	103	88	89	98	90	125	125	126
SLUB	135	136	136	137	137	138	138	140
SLQB	260	257	260	259	259	261	262	264

V. CONCLUSÕES

Este trabalho apresentou uma comparação de quatro KMAs (SLAB, SLOB, SLUB, e SLQB). Os experimentos adotaram cargas de trabalho constituídas de operações de escrita de dados em disco e transferências de dados entre posições de memória. Os KMAs foram analisados e comparados quanto ao tempo de execução, fragmentação e consumo de memória. Os achados experimentais sugerem que, de forma geral, o SLOB foi o KMA com melhor desempenho entre todos os alocadores avaliados, não obtendo o melhor desempenho apenas no critério de fragmentação de memória. Esse alocador foi seguido do SLAB, SLUB e SLQB. Em termos de tempo de execução o SLAB foi o segundo melhor. Já em consumo de memória o SLUB foi o segundo melhor alocador. Em número de eventos de fragmentação, o SLAB foi o alocador com melhor desempenho, seguido do SLUB. A Tabela III resume estes resultados.

TABELA III. SUMÁRIO DOS RESULTADOS (RANKING GERAL)

	Tempo (I/O)	Memória (I/O)	Eventos de Frag. (I/O)	Tempo (Transf.)	Memória (Transf.)
SLAB	2º	3º	1º	2º	3º
SLOB	1º	1º	4º	1º	1º
SLUB	3º	2º	2º	3º	2º
SLQB	4º	4º	3º	4º	4º

REFERÊNCIAS

[1] U. Vahalia, UNIX internals: the new frontiers, Prentice Hall Press, Upper Saddle River, NJ, 1995.

[2] T. B. Ferreira, R. Matias, A. Macêdo, L. B. Araujo, "A comparison of memory allocators for multicore and multithread applications: a quantitative approach," in Proc. of Brazilian Symposium on Computing Systems Engineering, Florianopolis, Brazil, 2011. SBESC., in press of IEEE Computer Society, Washington, DC, pp. 200-205, November 2011.

[3] T. B. Ferreira, M. A. Fernantes, R. Matias, "A comphensive complexity analysis of user-level memory allocator algorithms," in Proc. of Brazilian Symposium on Computing System Engineering, Natal, Brazil, pp. 5-7, 2012. PDCAT., in press of IEEE Computer Society, Washington, DC, pp. 99-104, November 2012.

[4] D. Elias, M. A. Fernandes, R. Matias, L. Borges, "Experimental and theoretical analyses of memory allocation algorithms," in Proc. of the 29th Annual ACM Symposium on Applied Computing, Gyeongju, Korea, 2014. SAC., in press of ACM New York, NY, USA, pp. 1545-1546, March 2014.

[5] J. Bonwick, "The slab allocator: an object-caching kernel memory allocator," in Proc. of USENIX Summer, Boston, USA, 1994. USTC., in press of USENIX Association, Berkeley, CA, pp. 87-98, June 1994.

[6] A. Kopytov, "SysBench manual", 2004. https://launchpad.net/sysbench/

[7] R. Matias, I. Beicker, B. Leitão, P. Maciel, "Measuring software aging effects through OS kernel instrumentation," in Proc. of Workshop of Software Aging and Rejuvenation, San Jose, USA, 2010. WoSAR., in press of IEEE Computer Society, Washington, DC, pp. 1-6, November 2010.

[8] J. Bart, P. Larson, B. Leitão e A. M. S. da Silva, SystemTap: instrumenting the Linux kernel for analyzing performance and functional problems., IBM Redbook, 2008.

[9] M. Gorman, Understanding the Linux Virtual Memory Manager, Prentice Hall PTR, Upper Saddle River, NJ, 2004.

[10] M. Mackall, "slob: introduce the SLOB allocator", LWN.net, 2005. http://lwn.net/Articles/157944/

[11] J. Corbet, "The SLUB allocator", LWN.net, 2007. http://lwn.net/Articles/229984/

[12] O. M. Perla Enrico, A Guide to Kernel Exploitation: Attacking the Core, Syngress Publishing, 2010.

[13] J. Corbet, "SLQB - and then there were four", LWN.net, 2008. http://lwn.net/Articles/311502/