PEDRO DEMARCHI GOMES

GERENCIAMENTO DE MEMÓRIA VIRTUAL COM HIPER PÁGINAS

Trabalho de Graduação apresentado ao Curso de Graduação em Ciência da Computação, Setor de Ciências Exatas,Universidade Federal do Paraná, como requisito parcial à obtenção do título de Bacharel em Ciência da Computação..

Área de concentração: *Ciência da Computação*.

Orientador: Roberto André Hexsel.

CURITIBA PR

2019

RESUMO

O gerenciamento de memória através da paginação é usado desde a década de 1970, e foi desenvolvida para os computadores da época, onde a quantidade de memória disponível era muito pequena. Nos computadores com grandes quantidades de memória, essa técnica passa a trazer diversos problemas, como tabelas de páginas muito grandes e uma grande quantidade de *TLB misses*, impactando negativamente no desempenho dos programas. Para solucionar esses problemas é criado nesse trabalho uma técnica de paginação com páginas de tamanhos variados, chamadas de hiper páginas. Através de um simulador o desempenho dessa técnica é testado e comparado com a segmentação e a paginação comum.

Palavras-chave: Memória virtual. Paginação. TLB.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O gerenciamento de memória é uma importante função dos sistemas operacionais, ele controla o acesso e alocação de memória dos processos, separando seus espaços de endereçamento. Isso facilita o desenvolvimento de programas, dado que cria uma abstração para o programador lidar com a memória, e adiciona mais segurança ao SO, pois isola a memória utilizada por cada processo, sendo que um processo não pode acessar a memória utilizada por outro, a não ser que ele o permita. Existem diversas técnicas para fazer tal gerenciamento, sendo a mais utilizada atualmente a paginação. Quando ela foi desenvolvida nas décadas de 1960 e 1970, o problema principal das máquinas da época era a escassez de memória RAM. Sendo assim, foi priorizada a facilidade de utilização de memória em detrimento do desempenho.

Em 2005 a largura do barramento de memória era de 40 bits. Atualmente temos computadores com barramentos de memória de largura de 50 bits, número que tende a crescer, e que, segundo [LH16, Lau17, LH18], se mantida essa tendencia de crescimento, na década de 2020 chegaremos a barramentos de 64 bits. Tanto nesse trabalho como em [Lau17], memórias de 2^{64} bytes são chamadas de “memórias infinitas".

Logo, a paginação passa a se tornar um problema, dado que o cenário em que este mecanismo foi concebido se alterou. A memória RAM não é mais um recurso escasso, fazendo com que seja necessário, após meio século, repensar a forma com que é feito o gerenciamento de memória.

[Lau17] mostrou que a segmentação, técnica que precedeu a paginação, passa a ser vantajosa em máquinas com "memória infinita". Essa vantagem se deve ao fato de que, para o gerenciamento de grandes quantias de memória, problemas como: (1) armazenamento da tabela de páginas (TP), que tem seu tamanho aumentado em consequência do aumento da memória disponível; e (2) aumento no número de *TLB misses*, causado por processos que utilizam uma grande quantidade de memória principal, tais como programas de análise de dados.

Nesse trabalho busca-se encontrar um solução intermediária, dando mais flexibilidade ao desenvolvedor do sistema operacional sem comprometer o desempenho. Foi desenvolvido um simulador, que utiliza como entrada *traces* obtidos em [Lau17], de um sistema de paginação com páginas de tamanhos variados, sendo estes potências de 2.

Em [Lau17] foram implementados dois simuladores de gerenciamento de memória virtual, um utilizando a técnica de paginação e o outro de segmentação. Seus resultados são comparados com os resultados obtidos nesse trabalho, para que se possa comparar a paginação, segmentação e paginação com páginas de tamanhos variados, chamada de hiper páginas.

2 MEMÓRIA VIRTUAL

Com o passar do tempo, notou-se a necessidade de executar programas grandes o bastante para não se encaixarem inteiramente na memória. Uma solução adotada para esse problema na década de 1960 foi a divisão do programa em módulos [TB14], chamada de sobreposição (*overlay*). Essa técnica consistem em dividir o programa em módulos, que teriam que ser independente entre si. Esses módulos deveriam ser carregados um por um, sobrepondo uns aos outros caso não houvesse mais espaço em memória. O carregamento dos módulos era feito pelo sistema operacional, porém a divisão de um programa em módulos deveria ser feita manualmente, tarefa lenta e propensa a erros.

Devido aos problemas do *overlay*, foi criada a memória virtual, para automatizar a tarefa de trazer dados da memória secundária para a memória principal, sem que o programador tenha que estar ciente. A memória virtual é uma camada de abstração que provê ao programador a impressão de que a memória instalada no computador é, para todos efeitos práticos, infinita [TB14]. Isso se obtém com a tradução de endereços virtuais para endereços físicos. Os *endereços virtuais* são gerados pelo carregador, eles são usados para fazer o endereçamento dentro de um determinado programa. Cada programa tem seu *espaço de endereçamento virtual*, que consiste no seu conjunto de endereços virtuais. Já o *espaço de endereçamento físico* se refere ao conjunto de endereços da *memória principal* (memória RAM). Os *endereços físicos* são utilizados para endereçar a memória principal.

Em um processador de 32-bits, com barramento de endereços de 32-bits, o espaço de endereçamento virtual é de 4GiB, porém a quantidade de memória principal da máquina pode ser inferior ao que seria possível acessar através do espaço de endereçamento virtual. O sistema de memória virtual faz com que o programador tenha a ilusão de que tem 4GiB disponíveis, e traduz os endereços virtuais para endereços físicos, podendo referenciar objetos que podem estar na memória principal ou na secundária.

3 HUGE PAGES

Para mitigar os problemas da paginação comum foram criadas as *Huges Pages*, que são um conjunto de páginas de tamanhos maiores, como 2MB e 1GB. Sendo assim quando for alocada uma grande quantidade ele poderá paginá-la com páginas dos tamanhos 2MB e 1GB, diminuindo o tamanho da TP, e consequentemente o número de *TLB misses*.

A distribuição Linux *Red Hat Enterprise Linux 6* (RHEL 6) implementa as *Huge Pages* de tamanhos 2MB e 1GB. Como o gerenciamento manual dessas páginas seria muito complexo, foi optado por fazer uma camada de abstração que torna transparente para o programador o gerenciamento das *Huge Pages*, chamanda de *Transparent Huge Pages* (THP). Essa camada tem como objetivo aumentar o desempenho do sistema, utilizando o tamanho de página mais adequado para cada requisição. Para isso a equipe de desenvolvimento do RHEL 6 testou e otimizou suas configurações *default* para diversas cargas de trabalho e aplicações.

A THP tenta sempre alocar um *Huge Page*, caso não consiga devido a falta de espaço na memória, são alocadas páginas comuns de tamanho 4KB. Para aumentar o uso das *Huge Pages* é criada uma kernel thread chamada de *khugepaged* que tenta substituir o uso de páginas comuns por uma *Huge Page*.

Em [PPG18] é mostrado que o uso de THP pode não ser benéfico para sistemas que executam durante um longo período de tempo, devido a fragmentação causada, que acaba fazendo com que a kernel thread *khugepaged* ocupe grande parte do tempo da CPU. Devido a isso diversas aplicações recomendam que a THP seja desabilitada, como o Hadoop, que em seu guia de *performance tunning* afirma que a utilização de CPU pelo kernel aumenta 66% utilizando THP no *benchmark* TeraSort.

4 ASSOCIATIVIDADE DE CACHES

Há três formas principais de se fazer o mapeamento de endereços da memória principal para a memória *cache*, (i) o mapeamento direto, (ii) o mapeamento totalmente associativo, e (iii) o mapeamento híbrido ou associativo por conjunto. Essas três formas de mapeamento são explicadas nas seções seguintes.

4.1 MAPEAMENTO DIRETO

No mapeamento direto, cada bloco de memória só pode ser mapeado para uma única linha da *cache*. O mapeamento usado é uma função de *hashing* simples, que usa m bits do endereço para escolher uma dentre as 2^m posições da cache. Estes m bits são chamados de *bits de índice*.

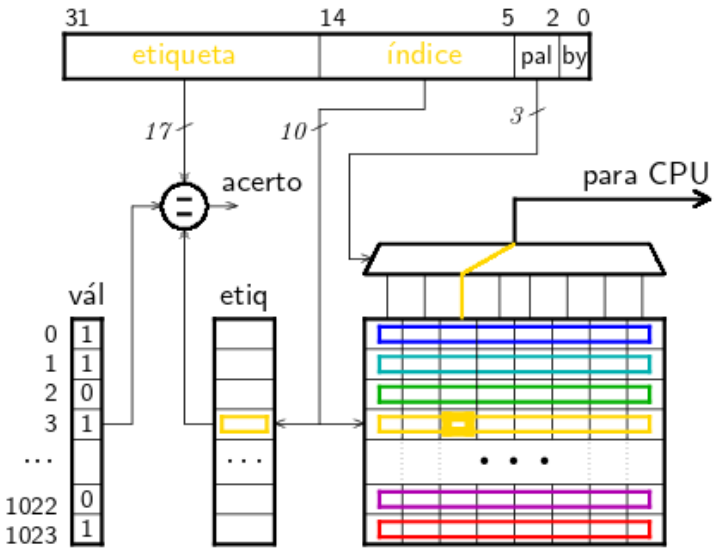

Figura 4.1: Mapeamento direto [Hex16]

A figura 4.1 mostra o acesso a uma *cache* com mapeamento direto de um processador com endereços de 32 bits. Nesse tipo de mapeamento o endereço deve ser separado em etiqueta, índice e deslocamento, sendo na figura *pal* e *by* o deslocamento. O índice é utilizado para endereçar a *cache*, sendo que seu tamanho é definido pelo tamanho da *cache*. A etiqueta é usada para resolver os conflitos de mapeamento, ao verificar se o bloco endereçado pelo índice corresponde ao endereço buscado, pela comparação de suas etiquetas. Caso as etiquetas não sejam iguais (*cache miss*), o bloco que está na cache deve ser substituído pelo do endereço buscado.

Esse mapeamento utiliza hardware simples. Caso haja muitas colisões entre os endereços acessados, a *cache* pode acabar sendo mal utilizada por causa dos frequentes conflitos de mapeamento – *hashing* com muitas colisões [Hex16].

4.2 MAPEAMENTO TOTALMENTE ASSOCIATIVO

No mapeamento totalmente associativo, cada bloco de memória pode ser mapeado para qualquer linha da *cache*. Esse é o mapeamento ideal e, infelizmente, o mais caro.

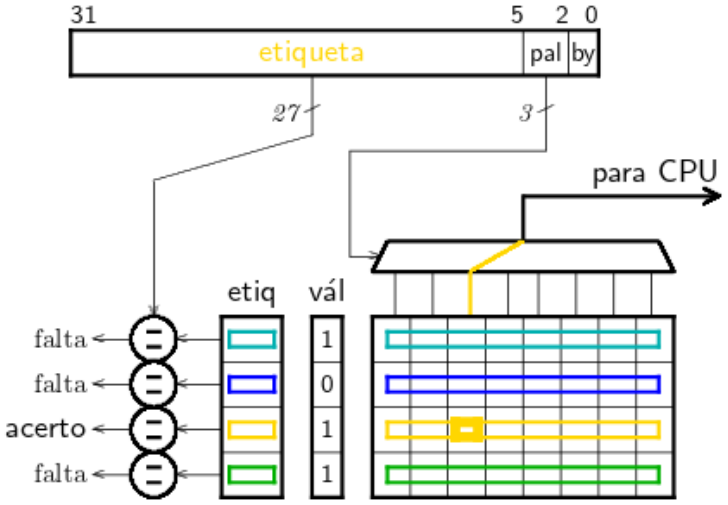

Figura 4.2: Mapeamento totalmente associativo [Hex16]

A Figura 4.2 mostra o acesso a uma *cache* com mapeamento totalmente associativo de uma memória com endereços de 32 bits. Nesse mapeamento o campo de índice não existe, deixando o campo de etiqueta ocupar o seu espaço. Como não há índice, para se fazer um acesso é necessário comparar a etiqueta do endereço que se deseja acessar com a etiqueta de todos os blocos da *cache*. Caso nenhuma das etiquetas sejam iguais a do endereço buscado (*cache miss*), uma das linhas da *cache* precisa ser substituída com o bloco do endereço faltante. Essa substituição é feita de acordo com a política de substituição adotada.

Para fazer a comparação das etiquetas é necessário um comparador para cada linha da *cache*, o que adiciona um custo de hardware se comparada com o mapeamento direto, mas são potencialmente evitados (todos) os conflitos de mapeamento [Hex16].

4.3 MAPEAMENTO HÍBRIDO OU ASSOCIATIVO POR CONJUNTO

O mapeamento híbrido combina as duas formas de mapeamento descritas nas seções acima. Isso busca melhorar o desempenho da *cache*, como no mapeamento totalmente associativo, porém sem aumentar demais a complexidade do hardware e, consequentemente, seu custo, como no mapeamento direto.

O mapeamento híbrido também pode ser chamado de mapeamento associativo de n vias, sendo que se $n = 1$ o mapeamento é direto e se $n = log(tamanho\ da\ cache)$ o mapeamento é totalmente associativo.

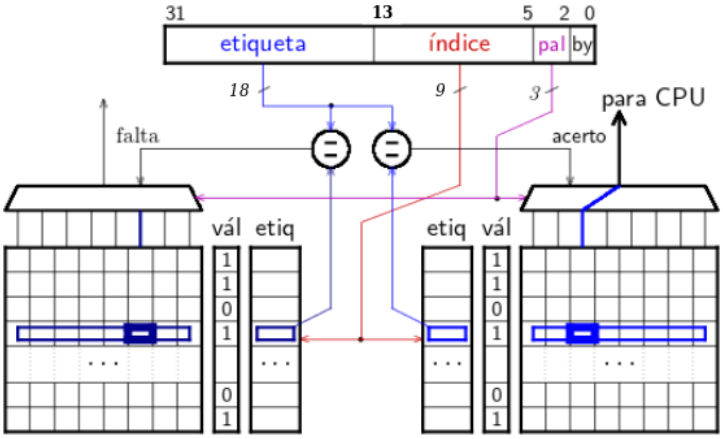

Figura 4.3: Mapeamento associativo de duas vias [Hex16]

A Figura 4.3 mostra o acesso a uma *cache* com mapeamento associativo de 2 vias de uma memória com endereços de 32 bits. Como no mapeamento direto o endereço é separado em etiqueta, índice e deslocamento (*pal* e *by*). O índice aqui tem a capacidade de endereçar apenas metade da *cache*.

Em um acesso a memória, o índice é usado para endereçar a *cache*, porém diferentemente do mapeamento direto aqui ele retorna duas de suas linhas. Para saber qual bloco corresponde ao endereço buscado são comparadas as etiquetas das linhas com a etiqueta do endereço buscado. Caso nenhuma das etiquetas sejam iguais a do endereço buscado (*cache miss*), uma das linhas precisa ser substituída com o bloco do endereço. Como na *cache* totalmente associativa, essa substituição é feita de acordo com a política de substituição definida.

As *caches* podem ter 2^n vias, sendo que quanto mais vias melhor o desempenho, porém maior o custo. Logo é preciso chegar a um equilíbrio entre esses fatores, de forma que se chegue a uma CPU com desempenho satisfatório sem aumentar muito seu custo.

4.4 POLÍTICA DE REPOSIÇÃO LRU

Programas exibem dois tipos de comportamento que são explorados na construção de hierarquias de memória (registradores $\subset$ cache $\subset$ memória principal $\subseteq$ memória secundária), que são os dois modos de *localidade de referência*, conhecidos como as propriedades de localidade espacial e localidade temporal.

A propriedade de *localidade espacial de referência* indica que, se uma posição de memória foi referenciada, então existe uma alta probabilidade de que localidades próximas serão referenciadas num futuro próximo. As referências aos elementos contíguos de um vetor exibem localidade espacial.

A propriedade de *localidade temporal de referência* indica que, se uma posição de memória foi referenciada, então existe uma alta probabilidade de que esta posição seja referenciada novamente num futuro próximo. As referências à variável que indexa um vetor no corpo de um laço exibem localidade temporal.

O "conjunto de trabalho"de um programa (*working set*) [Den70] é o conjunto de posições de memória que foram referenciadas durante o último intervalo de duração τ. Se $C_1 = \{a, b, c, d\}$ posições distintas foram referenciadas frequentemente durante τ_1, então o conjunto de trabalho do programa em τ_1 é C_1. No intervalo τ_n, o conjunto de trabalho é $C_n = \{p, q, \ldots x, y\}$. Ao longo do tempo, o conjunto de trabalho de um processo é o conjunto de páginas que foram ativamente referenciadas numa janela de tempo de duração τ.

A política *Least Recently Used* (LRU) é um algoritmo usado na substituição das páginas nas caches de tradução de endereço. Essa política se aproveita dos princípios de localidade temporal e do *working set*, e tem como objetivo minimizar o número de trocas de página nessas *caches*

A seguir é descrito o algoritmo LRU-perfeito, que na prática só é empregado em simulações devido à complexidade e tempo de execução. O que é empregado em paginação/segmentação/caches são aproximações do LRU que podem ser implementadas com custos aceitáveis.

É preciso manter uma pilha com as páginas acessadas. Essa pilha deve estar ordenada, com as páginas mais acessadas no topo. Para isso, cada vez que uma página é acessada ela é retirada de sua posição na pilha da LRU e inserida no topo. A pilha do LRU pode ser de diferentes tamanhos, limitada pelo tamanho da TLB. Quanto maior o tamanho da TLB maior sua eficiência, porque contém um histórico mais longo da execução do processo.

A Figura 4.4 mostra uma sequência de referências a uma memória de tamanho 4. A sequência inicia com referências às posições 3, 4, 2 e 6 (na primeira coluna), seguidas pelas referências acima da linha horizontal. Cada coluna mostra, do topo para baixo, a página acessada e disposição da fila da LRU após o acesso. Itens em negrito são referências encontradas na memória (*hits*).

		4	7	1	3	2	6	3	5	1	2	3
topo	6	**4**	7	1	3	2	6	**3**	5	1	2	**3**
	2	6	4	7	1	3	2	6	3	5	1	2
	4	2	6	4	7	1	3	2	6	3	5	1
base	3	3	2	6	4	7	1	1	2	6	3	5

Figura 4.4: LRU em memória de tamanho 4 [Hex19]

5 PAGINAÇÃO

A *paginação* é uma técnica utilizada na gerência da memória virtual. Essa técnica consiste em dividir a memória em partes do mesmo tamanho, geralmente 4KiB. Essas partes são chamadas de *páginas*, e são usadas na tradução de endereços virtuais para endereços físicos. Para isso é necessário armazenar uma *tabela de páginas*, que conterá um endereço físico correspondente a cada endereço virtual.

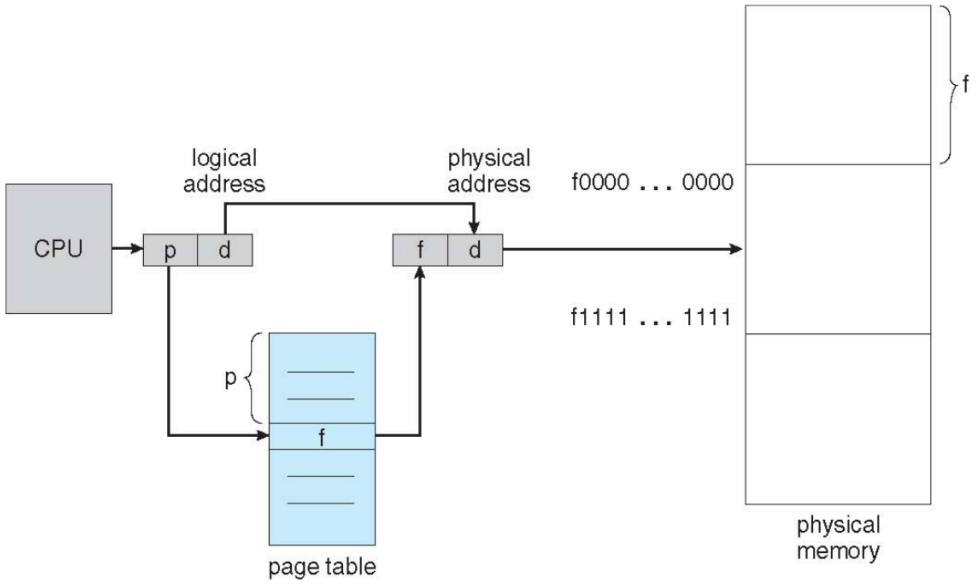

Figura 5.1: Tradução de um endereço virtual para um endereço físico com uma tabela de páginas [O9]

A Figura 5.1 mostra a tradução de um endereço virtual em um de endereço físico. Um *endereço virtual* é dividido em dois campos, o *número de página virtual* (p) e um *deslocamento* (d) a partir do início da página. Um *endereço físico* também é dividido em dois campos, um *número de página física* (f) e um *deslocamento* (d). O deslocamento tem o mesmo número de bits nos dois endereços. A tradução nada mais é do que um acesso a uma *tabela de páginas* (TP) na posição p, de onde se obtém f, e a concatenação do resultado com d, gerando o endereço físico $\langle f, d \rangle$.

A TP também armazena as permissões das páginas, como leitura, escrita e execução. As permissões tornam a paginação também um mecanismo de segurança.

Na maior parte dos casos, o endereço físico obtido na tradução é utilizado para acessar a memória principal, quando o conteúdo do endereço é retornado para a CPU. Neste caso, quando a tradução EF-EV é encontrada na TP, ocorre um *acerto* (*hit*) na TP.

Num dado instante, a demanda de espaço em memória por todos os processos, pode não ser atendida somente pela memória física instalada. Uma região do disco magnético é reservada para acomodar 'pedaços' de processos que não estejam ativos. Quando a demanda aumenta, o SO

escolhe um ou mais ‘pedaços’ de processos e os copia para o disco, liberando espaço em RAM. Se os pedaços voltam a ser necessários, estes são copiados do disco para a RAM. Esta região do disco é chamada de *área de troca* (*swap area*), e é gerenciada pelo SO de forma transparente ao programador. Isso faz com que o programador possa utilizar mais memória em seus programas do que a quantidade de memória principal disponível.

Existe a possibilidade de o endereço físico obtido pela tradução estar na memória secundária, como páginas armazenadas na *área de troca*. O tempo de acesso a memória secundária pode ser 10^5 vezes maior do que o tempo da memória principal, como no caso de um disco rígido, tornando a computação de processos que acessam páginas em área de troca mais lenta [LH16].

No caso de uma CPU de 32 bits com páginas de 4KiB, p e f teriam 20 bits, e d teria 12 bits. Nesse caso a TP teria $2^{32}/2^{12} = 2^{20}$ elementos. Com cada elemento contendo 4 bytes, o tamanho da TP pode chegar a 4MiB. Dado que cada processo deve ter sua própria TP, pode ser necessário que ela seja armazenada na memória secundária. Como usar 4MiB de memória para a TP de cada processo é inviável, a TP é implementada de maneira hierárquica [Hex19].

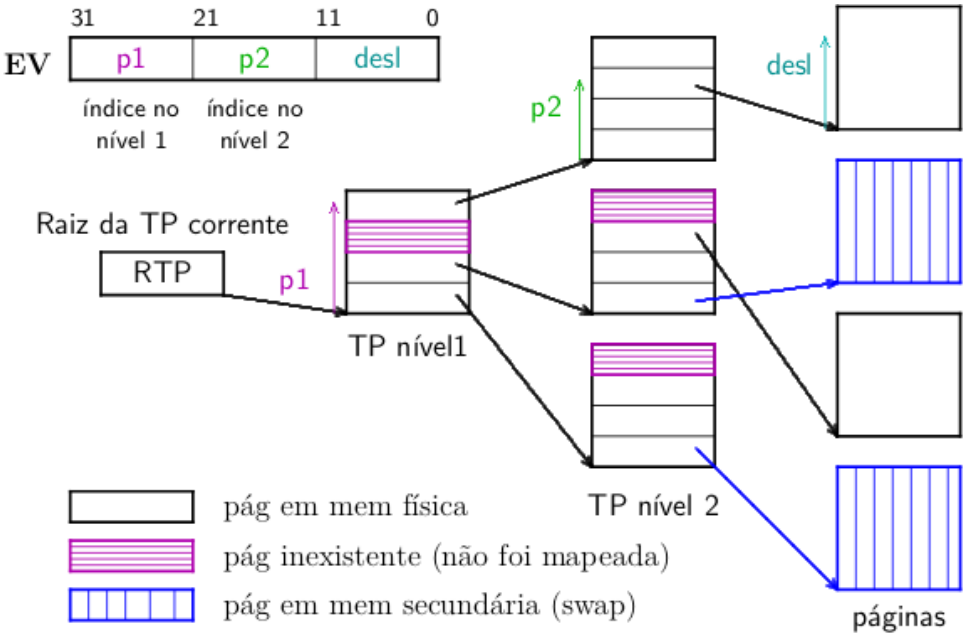

Figura 5.2: Tradução de um endereço virtual para um endereço físico com uma tabela de páginas hierárquica [Hex19]

A Figura 5.2 mostra uma TP hierárquica. Para fazer um acesso nessa TP o endereço virtual é divido em 3 partes, duas de índice de nível ($p1$ e $p2$), e outra de *offset*. O índice $p1$ indexa o primeiro nível da TP, e $p2$ o segundo nível. O valor de *desl* indexa um byte dentro da página. Como estão sendo usados endereços de 32 bits e páginas de 4KiB, e tendo cada linha das TPs o tamanho de 4 bytes, cada TP do nível 1 e 2 caberia em exatamente uma página. Este é o modelo de TP hierárquica usado nos processadores da família x86.

Para acelerar a tradução de endereços, os processadores empregam um *Translation Lookaside Buffer* (TLB). A TLB é uma *cache*, que armazena a tradução dos endereços virtuais

acessados mais recentemente. Uma *memória cache* é geralmente implementada com memória RAM de alta velocidade e de pequena capacidade e é usada para manter dados usados num passado recente, na esperança de que estes sejam usados novamente num futuro próximo. Memórias cache são geralmente instaladas junto ao circuito que 'consome' os dados armazenados nela.

A decisão de quais endereços devem ficar na TLB é baseada em uma política de reposição, sendo a mais popular delas a política *Least Recently Used* (LRU). Quando um endereço virtual é traduzido, primeiramente ele é buscado na TLB, se encontrado então é retornada sua tradução para a CPU (quando ocorre um *TLB-hit*). No caso de não ser encontrado na TLB, ocorre um *TLB miss* e esse endereço é buscado na TP, e inserido na TLB de acordo com a política de substituição da TLB.

Um exemplo de uso comum dessas permissões para a segurança é que, normalmente, páginas com permissão de escrita não podem ser executadas. Isso é usado em páginas do segmento de dados de processos, evitando que um atacante possa desviar a execução de um programa para um código malicioso inserido nesse segmento. No caso de alguma tentativa de violar as permissões das páginas, como uma tentativa de escrever em uma página de somente leitura, o processo é interrompido pelo SO.

A paginação evita a *fragmentação externa* da memória, pois a memória é sempre alocada em quantias múltiplas do tamanho da página. Porém é introduzido o problema de *fragmentação interna*: cada processo desperdiça, em média, meia página para cada segmento lógico do programa. Logo a fragmentação interna é pequena.

5.1 ÁREA DE TROCA OU *SWAPPING*

A *área de troca*, ou área de *swap*, nada mais é do que uma área da memória secundária, normalmente disco ou SSD, para onde páginas podem ser mapeadas. Normalmente os SOs modernos utilizam a *swap* para armazenar páginas inativas, liberando mais espaço da memória principal [Hex19]. Quando uma página que se encontra na área de *swapping* é referenciada, é preciso que essa página seja copiada para memória principal. A TP do processo deve ser alterada para refletir a mudança ocorrida no mapeamento da página, que antes estava na área de *swapping* e foi movida para a memória principal. É chamado de *swapping out* a operação de transferência de página da memória principal para a secundária, e de *swapping in* a movimentação no sentido contrário.

Quando o *working set* [Den70] de um processo não cabe inteiramente na memória principal o mecanismo de *swap* é usado intensivamente. Isso causa um sério problema de desempenho, pois como a memória secundária é mais lenta que a memória principal, na ordem de $10^6\times$ para discos e $10^3\times$ para SSDs, e as operações de *swapping in* e *swapping out* são executadas constantemente, o programa acaba tendo seu desempenho diminuído drasticamente [LH18].

5.2 ARQUIVOS MAPEADOS EM MEMÓRIA

Nos sistemas operacionais modernos, é possível que um processo solicite que arquivos sejam mapeados em memória. Ao invés de o arquivo, ou um pedaço dele, ser copiado do disco para a memória principal, para então ser usado pelo processo, nos arquivos mapeados em memória um mapeamento é criado na TP para o arquivo no disco [Hex19]. Se o arquivo mapeado em memória for usado somente para leitura, nenhum providencia adicional precisa ser tomada, além de as páginas usadas no mapeamento terem permissão somente de leitura. Isso é muito usado para compartilhar informações entre processos. No caso de o arquivo mapeado em memória ser usado para escrita, suas páginas devem ser marcadas com permissão de escrita, e ao serem 'des-mapeadas' as alterações dever ser propagadas até a cópia em disco [Hex19].

6 SEGMENTAÇÃO

Diferentemente da paginação, em sistemas com *segmentação* a memória não é dividida em partes iguais (páginas), podendo os processos alocarem blocos contínuos de memória de tamanhos diferentes, chamados de *segmentos*.

Um processo no SO é divido em segmentos lógicos, como *text*, *data*, *bss*, *heap* e *stack*. A segmentação divide naturalmente os processos de acordo com seus segmentos lógicos [Maz19].

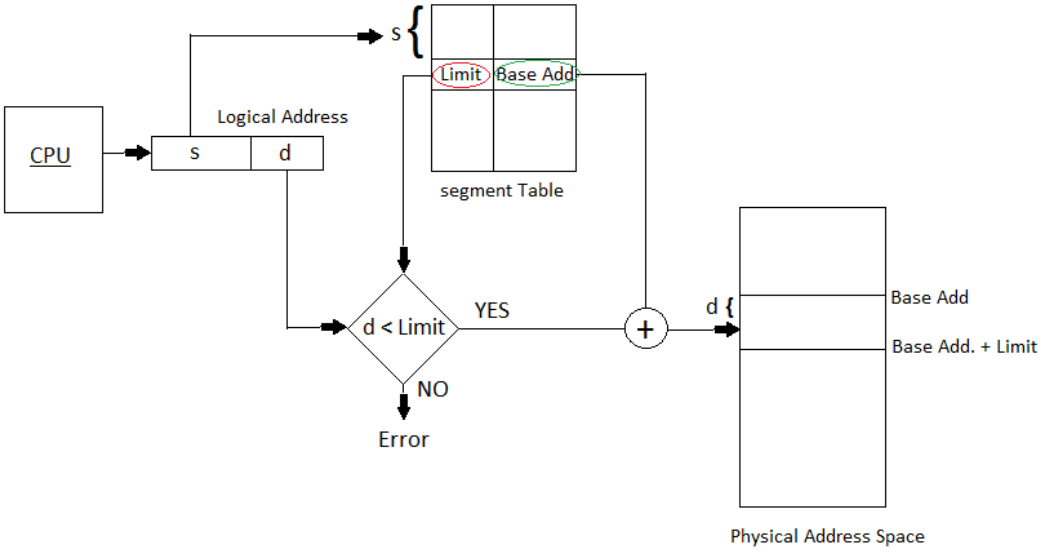

Figura 6.1: Tradução de um endereço virtual para um endereço físico com uma tabela de segmentos [gw19]

Como os segmentos têm tamanhos arbitrários, é preciso salvar além do endereço inicial o endereço final dos segmentos. Dessa forma, o tamanho dos segmentos é obtido subtraindo o endereço final pelo inicial.

Em sistemas segmentados, um endereço virtual é dividido em dois campos: o campo *s* é o *número do segmento* e indexa a tabela de segmentos, o campo *d* é o *deslocamento* (*offset*) do byte endereçado dentro do segmento. A tabela de segmentos contém, em cada linha, um segmento diferente, e para cada segmento o seu endereço inicial, endereço limite e permissões de acesso (semelhante a paginação).

Como mostra a Figura 6.1, o campo *s* é usado para indexar a tabela de segmentos, acessando o endereço base e o limite do segmento. Com isso basta comparar *d* com o limite para saber se o endereço virtual não ultrapassa o tamanho do segmento, e caso o resultado da comparação seja falso, somar *d* com o endereço base do segmento.

Num sistema segmentado, pode-se empregar uma cache para conter as traduções dos endereços virtuais "mais acessados", com a reposição também baseada em uma política, como a LRU.

A segmentação, diferentemente da paginação, sofre com o problema de *fragmentação externa*. Conforme a memória é usada ao longo do tempo, são alocados e liberados diversos segmentos de tamanhos diferentes. Ao se criar um novo segmento de tamanho *m*, é possível que a quantia total de memória livre seja maior do que *m*, mas não exista uma região contínua de memória livre com esse tamanho. Para resolver esse problema, os segmentos teriam que ser deslocados para desfragmentar a memória, o que pode fazer com que a criação de um novo segmento se torne uma operação muito custosa e lenta.

6.1 PAGINAÇÃO COM SEGMENTAÇÃO NO MULTICS

O sistema operacional MULTICS (Multiplexed Information and Computing Service), desenvolvido para o mainframe GE-645 pelo Massachusetts Institute of Technology (MIT), General Electric (GE) e Bell Labs em 1965, avançou muito na questão de gerenciamento de memória da época, utilizando memória virtual com segmentos paginados [mw19].

O uso da segmentação foi motivado pela maneira mais simples que ela oferece para compartilhar informações entre os processos. Já a paginação foi usada para diminuir a fragmentação externa causada pela segmentação, que implicava na subutilização da memória disponível, além de otimizar o tráfego entre as memórias primária e secundária.

Para utilizar a segmentação com paginação, cada segmento deve ter sua tabela de páginas. Dessa forma as entradas da tabela de segmentos eram os endereços base para as tabelas de páginas dos segmentos.

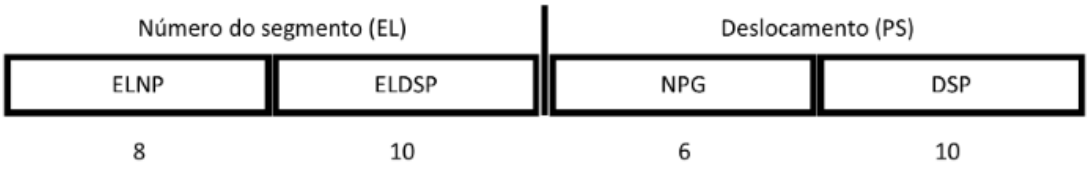

Figura 6.2: Endereço de memória do MULTICS [Lau17]

A Figura 6.2 mostra como é o endereçamento no MULTICS. O número do segmentos tem 18 bits (EL), e é divido em ELNP e ELDSP, pois 18 bits demandavam uma tabela de segmentos muito grande para a época, logo optou-se por fazer paginação na tabela de segmentos. O ELNP é o índice na tabela de páginas da tabela de segmentos, e ELDSP é o deslocamento numa página. Dessa forma é encontrado o segmento do endereço. NPG é o índice da tabela de páginas do segmento e DSP o deslocamento dentro dessa página. O processo para traduzir os endereços é mostrado na Figura 6.3.

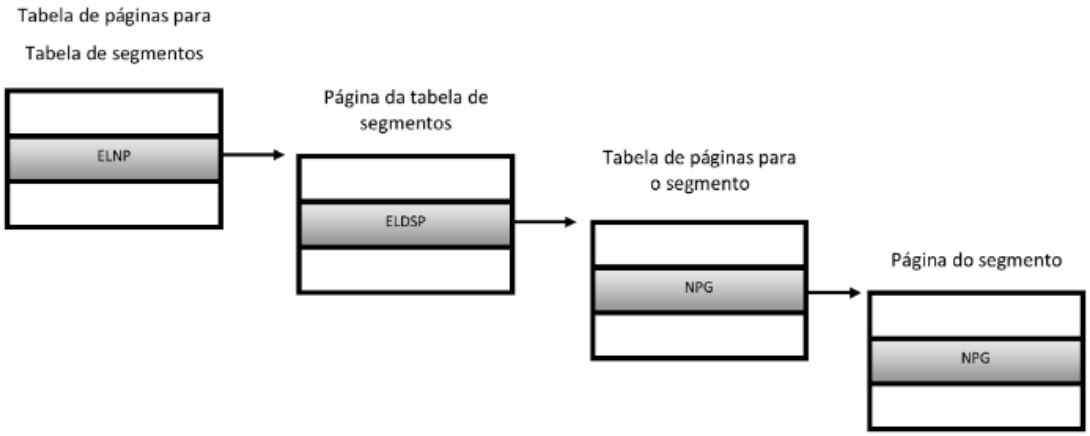

Figura 6.3: Tradução de endereço no MULTICS [Lau17]

Apesar de vantagens como: (1) não sofrer com fragmentação externa, (2) facilidade para compartilhar informações entre processos; (3) e menor tamanho das tabelas de páginas, a paginação com segmentação não é mais usada pela sua complexidade, tanto de hardware quanto de software, que não compensa as vantagens oferecidas [Lau17].

7 MEMÓRIA INFINITA

A quantidade de memória principal disponível nos computadores tem crescido com o passar dos anos. Em 1995 um barramento típico de memória tinha 32-36 bits de endereço. Uma década depois o barramento de memória aumentou para 40 bits de endereço, e em 2019 é possível comprar computadores com barramentos de endereço com mais de 50 bits de largura. Segundo [Lau17], se mantida essa tendência de crescimento, na década de 2020 chegaremos a barramentos de 64 bits. Tanto nesse trabalho como em [Lau17], memórias de 2^{64} bytes são chamadas de "memórias infinitas" porque, para fins práticos, 2^{64} bytes é uma capacidade muito grande.

Tendo em vista que os sistemas operacionais, em especial o gerenciamento de memória, foi pensado para solucionar problemas causados principalmente pela escassez de memória RAM e a baixa velocidade dos discos magnéticos, os novos sistemas operacionais deverão mudar significativamente o modo com que esse gerenciamento é feito.

Técnicas utilizadas em sistemas operacionais no passado, como a segmentação, voltam a ser interessantes com "memórias infinitas". Isso se deve a diversos fatores, sendo o principal deles o desempenho de programas que, por utilizarem muitas páginas, acabam gastando boa parte do tempo gerenciando a memória pois diversos dos mapeamentos necessários não estão na TLB porque sua capacidade é relativamente pequena, de 64 a 256 mapeamentos. [HCG+15] demonstra que o impacto negativo no desempenho de programas é da ordem de 10 a 20%, mas dependendo da carga de trabalho, pode ultrapassar os 50%.

Programas que utilizam grandes quantidades de memória e, consequentemente, muitas páginas, acabam gerando uma maior quantidade de *TLB misses*, pois a TLB é incapaz de armazenar os mapeamentos de seu *working set*. Isso faz com que o desempenho desses programas seja prejudicado pelo tráfego entre a TLB e a memória *RAM*, dado que a cada TLB *miss* é necessário acessar a TP, que está na memória *RAM*, para substituir um mapeamento da TLB pelo mapeamento requisitado.

8 *HIPER* PÁGINAS

Os processadores modernos podem ser configurados para utilizarem tamanhos de página diferentes. O processador o Intel x86-64 tem suporte para páginas de 4KiB, 2MiB e 1GiB. Isso pode trazer ganhos de desempenho para alguns programas, dependendo do seu *working set*, localidade de referência espacial e padrão de uso [Lau17]. O ganho varia de aplicação para aplicação, podendo chegar a reduzir em 98% o número de *TLB misses* [GS98]. O aumento do tamanho de página não significa necessariamente aumento de desempenho, haja vista que aplicações que usam diversos mapeamentos pequenos causam muita fragmentação interna.

Neste trabalho é avaliada uma técnica de paginação que utiliza páginas de tamanhos variados, para que uma aplicação possa alocar páginas de diversos tamanhos, sendo que os tamanhos devem ser potências de 2. Essa técnica, chamada de *hiper páginas*, permite maior controle do gerenciamento de memória. Com hiper páginas, os tamanhos de página podem ser alocados de acordo com a necessidade. Caso fosse preciso alocar áreas grandes de memória que seriam utilizadas de maneira sequencial, seria alocada uma página grande, ao invés de diversas páginas pequenas como no caso da paginação comum. Dessa forma o número de páginas utilizadas diminui significativamente, dado que não é necessário mapear diversas páginas pequenas, sendo possível mapear apenas uma (hiper)página grande. Isso também diminui o número de TLB *misses* sem aumentar a fragmentação interna, haja vista que não se adota um conjunto pequeno de tamanhos de página.

Em um sistema com hiper páginas, o compilador é responsável por avisar o SO dos tamanhos dos segmentos, para que a cada segmento seja alocada uma página do menor tamanho possível e que contenha o segmento inteiro.

Uma TP simplificada de um sistema com hiper páginas contém três campos por linha: *máscara da etiqueta* (mTAG), *etiqueta* (TAG) e *endreço base* (EB). A mTAG é a máscara que define quantos bits da TAG são usados realmente como TAG, ajustando assim o tamanho da página nas comparações. O campo TAG contém os bits de etiqueta página, com seu tamanho definido pela mTAG. O EB é o endereço da base da página, que é o endereço do “primeiro" byte da página.

A Figura 8.1 mostra uma TLB que suporta hiper páginas e que contém os mesmos campos da TP descrita acima.

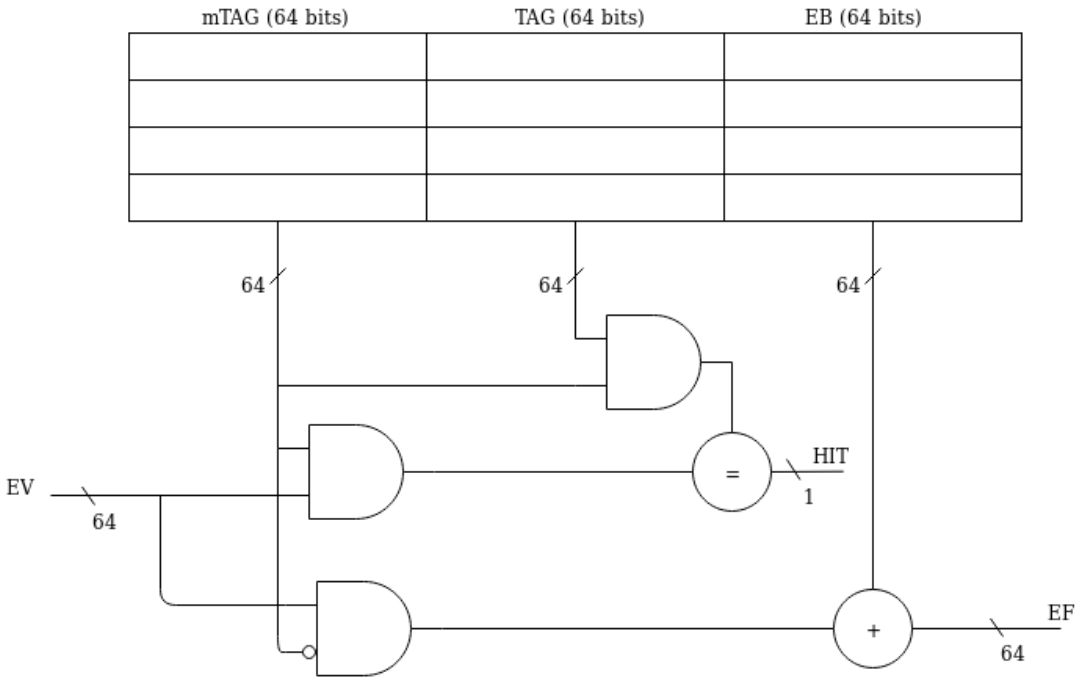

Figura 8.1: TLB com hiper páginas.

A tradução de um endereço virtual (EV) para endereço físico (EF) funciona da seguinte maneira: é feito um AND entre o EV e a mTAG, e o resultado é comparado com TAG. Caso sejam iguais, ocorre um *hit* na TLB, indicando que a tradução para o EV foi encontrada na TLB. A linha da TLB na qual ocorreu o *hit* contém o EB da página. Logo, para obter o EF é feito um AND entre a mTAG negada e o EV, e somado o resultado ao EB da linha.

A definição do tamanho da página é feita pela mTAG, limitando-os a potências de 2.

Considere que a Figura 8.2 seja uma linha da TLB, e o EV a ser traduzido seja $0x00000000.2180095A$. Primeiramente é preciso fazer o AND entre o EV e a mTAG, obtendo $0x00000000.21800000$, que por sua vez é comparado com o AND entre a TAG e a mTAG, que resulta no mesmo valor. Sendo assim, o comparador tem como saída o bit 1, caracterizando um *hit*. Assim é possível saber que essa linha da TLB contém a tradução do EV. Para obter o EF é preciso fazer um AND entre a mTAG negada e o EV, resultando em $0x00000000.0000095A$, que é somado com o EB, obtendo o EF $0x00000FE7.A860095A$.

mTAG (64 bits)	TAG (64 bits)	EB (64 bits)
0xFFFFFFFF.FFF00000	0x00000000.21800000	0x00000FE7.A8600000

Figura 8.2: Exemplo de linha da TLB.

9 SIMULADOR DE MEMÓRIA VIRTUAL COM *HIPER* PÁGINAS

Para simular a paginação com hiper páginas foi construído um simulador em linguagem *Crystal*. Esse simulador recebe como entrada *traces* de memória, que são sequências de endereços referenciados por um programa, e uma TP pré computada a partir de mapas de memória obtidos durante a execução do programa.

Os *mapas de memória* foram obtidos durante a execução dos programas de teste pela observação dos mapeamentos dos segmentos lógicos dos programas criados e removidos durante a execução. Por exemplo, cada biblioteca ligada ao programa acrescenta três segmentos lógicos ao processo: (1) o texto da biblioteca, (2) sua área de dados estáticos, e (3) sua área de *heap*; cada segmento desses com uma ou mais páginas. Os mapas de memória foram obtidos com execuções periódicas de *pmap*.

Os *traces* e os mapas de memória originais utilizados nesse trabalho foram obtidos em [Lau17]. A seguir são descritos os dados gerados da dissertação e como eles foram obtidos. Foi utilizado o programa *Valgrind*, com o módulo *Lackey*, para gerar os *traces* das primeiras 10 bilhões de instruções dos programas de teste. Durante a coleta dessas instruções, a cada 10 milissegundos, é executado um *script* denominado *pmap observer*, que extrai o mapa de memória da execução do processo naquele instante.

Para transformar os mapas de memória numa TP, a sequência de mapas é concatenada, e as repetições são removidas. Caso haja uma sobreposição de dois ou mais segmentos de mesmo nome mas tamanhos distintos, o maior mapeamento é mantido. Isso feito, os tamanhos dos segmentos são alterados para potências de 2 e depois da modificação nas faixas de endereços, alguns segmentos podem se sobrepor e precisam ser 'empurrados' para endereços mais altos. Depois que a TP para hiper páginas é criada, é preciso ajustar os *traces* para que estes reflitam as mudanças nos endereços dos segmentos lógicos: os *traces* são percorridos e os endereços ajustados de acordo com o deslocamento aplicado ao seu novo mapeamento.

A saída da simulação é um arquivo contendo o número de acessos de cada página e o número de *TLB misses* durante a execução simulada.

Um mapa de memória pode ser visto na Figura 9.1. Ele contém as colunas de endereço, tamanho, permissões e processo proprietário do segmento lógico. Cada linha da tabela corresponde a um segmento lógico do processo (texto, dados, pilha, *heap*), ou aos segmentos lógicos de uma biblioteca que seja ligada ao processo (texto, dados e *heap*). Na Figura 9.1, `ld-2.23.so` corresponde ao segmento de texto do ligador dinâmico (com permissão de execução), que é uma biblioteca ligada a todos os processos.

Address	Size	Rights	Owner
0000.0000.0040.0000	72K	r-x--	lsblk
0000.0000.0061.1000	4K	r----	lsblk
0000.0000.0061.2000	4K	rw---	lsblk
0000.0000.0400.0000	152K	r-x--	ld-2.23.so
0000.0000.0402.6000	8K	rw---	[anon]
0000.0000.0402.8000	28K	r--s-	gconv-modules.cache
0000.0000.0404.b000	4K	rw---	[anon]
...	...	...	...
ffff.ffff.ff60.0000	4K	r-x--	[anon]

Figura 9.1: Mapa de memória [Lau17]

Segmentos "anônimos"são segmentos que não estão relacionados a nenhum arquivo em disco, tais como área de *heap* ou dados não inicializados (sessão *bss*) [Maz19].

Nas próximas seções são descritos os passos executados para se fazer a simulação de um programa utilizando memória paginada com hiper páginas.

9.1 TABELA DE *HIPER* PÁGINAS

A partir dos mapas de memória do programa a ser simulado, deve ser gerada uma TP que utiliza hiper páginas. Isso é feito concatenando os mapas de memória obtidos no decorrer da execução do programa, e eliminando mapeamentos repetidos para um mesmo segmento lógico, usando o mapeamento de maior tamanho. Os mapeamentos com mesmo dono (mesmo segmento lógico) e mesmas permissões que estiverem em endereços consecutivos são concatenados, e os tamanhos dos mapeamentos alterados para a menor potência de 2 que é maior do que o tamanho original. Isso pode fazer com que o tamanho de alguns mapeamentos aumente a ponto de se sobrepor a outros mapeamentos. Os endereços dos mapeamentos que foram sobrepostos devem ser deslocados para remover a sobreposição. Os deslocamentos de endereços para cada mapeamento são mantidos em uma tabela para a próxima etapa, quando os deslocamentos são aplicados aos *traces* para não alterar seu comportamento normal.

Após essas alterações o mapa de memória pode ser convertido em uma TP que utiliza hiper páginas, bastando transformar o tamanho de cada mapeamento em uma máscara de etiqueta (mTAG). Para cada mapeamento, a mTAG é um número com a mesma quantidade de bits do endereço, tendo os n bits mais significativos iguais a 0 e os demais iguais a 1, porque o tamanho do segmento é igual a 2^n.

9.2 AJUSTES NOS *TRACES*

Os endereços contidos nos *traces* devem ser ajustados de acordo com os deslocamentos feitos na criação da TP para hiper páginas. Para isso é preciso utilizar a tabela dos mapeamentos com seus deslocamentos, e com essa informação percorrer os *traces* calculando, para cada

endereço, a qual mapeamento ele pertencia e somando o deslocamento necessário a esse mapeamento ao endereço original.

9.3 SIMULAÇÃO DE EXECUÇÃO DOS *TRACES*

A simulação é feita utilizando os *traces* deslocados e a TP com hiper páginas. A TLB simulada é totalmente associativa, e utiliza como política de substituição o LRU-perfeito. São avaliadas na simulação a quantidade de TLB *misses* e o número de acessos em cada página.

O trecho de código abaixo faz parte da classe TLB. O método *translate* faz a tradução dos endereços virtuais, contando também o número de referências para cada página. Primeiramente a tradução para o endereço *virtualAddress* é buscada na TLB, e caso encontrada é atualizada a lista de LRU. Caso o endereço não esteja na TLB, é incrementado o contador de *TLB miss* total da simulação, e a tradução é buscada na TP. Ao ser encontrada a tradução na TP a página é inserida na TLB.

Quando uma página é referenciada, esteja ela na TLB ou não, seu contador de referências é incrementado.

```
1  def translate(virtualAddress : UInt64)
2    # Search address in TLB
3    @tlb.each do |entry|
4      if (entry.valid) &&
5        ((entry.tagMask & entry.tag_offset) == (entry.tagMask & virtualAddress))
6           updateLRU(entry.ptIndex)
7        # Increment count of accesses in pt entry
8           @pt[entry.ptIndex].countAccess += 1
9
10          return (~entry.tagMask & entry.tag_offset) + entry.ppn
11        end
12   end
13   # If address is not in TLB, search in PT
14   @missCounter += 1
15   @pt.each do |entry|
16     if (entry.tagMask & entry.tag_offset) == (entry.tagMask & virtualAddress)
17          # Increment count of accesses in pt entry
18          @pt[entry.ptIndex].countAccess += 1
19
20          insertTlbLRU(entry, entry.ptIndex) # Insert address in TLB using LRU
21          return (~entry.tagMask & entry.tag_offset) + entry.ppn
22     end
23   end
24   return -1;
25 end
```

9.4 SAÍDA DA SIMULAÇÃO

A saída da simulação é o número de TLB *misses* total da simulação e a TP acrescida com o campo de número de referências para cada página.

Abaixo é mostrado um exemplo de saída da simulação. A primeira linha contém a quantidade total de *TLB misses* da simulação. As demais linhas contém informações das páginas, cada linha contendo o número da página, a máscara de TAG, a TAG e o número de referências.

```
Miss counter = 26513962

[0]: tagMask: fffffffffff0, tag: 0, countAccess: 7406220
[1]: tagMask: ffffffc00000, tag: 400000, countAccess: 1880331637
[2]: tagMask: fffffff00000, tag: a00000, countAccess: 72848708
[3]: tagMask: ffffffc0000, tag: b40000, countAccess: 27312440
...
[126]: tagMask: ffffff800000, tag_offset: 1006800000, countAccess: 0
[127]: tagMask: fffff8000000, tag_offset: 1008000000, countAccess: 0
[128]: tagMask: ffffffe0000, tag_offset: 1fff000000, countAccess: 378006275
[129]: tagMask: ffffffc0000, tag_offset: 7ffca2380000, countAccess: 0
[130]: tagMask: fffffffe000, tag_offset: 7ffca23ec000, countAccess: 0
```

10 EXPERIMENTOS

Foram simuladas os *traces* gerados a partir das primeiras 10 bilhões instruções dos seguintes programas: Python, Libre Office, MySQL e Firefox. Esses *traces* foram obtidos em [Lau17]. Essas aplicações foram escolhidas pois devido a limitação do número de instruções capturadas, é preciso que sejam capturadas instruções o bastante para que o programa entre em execução de fato, e não apenas carregue suas bibliotecas. Aumentar o número de instruções não é uma solução viável, dado que os tamanhos dos arquivos gerados chegam a 120 GiB. Sendo assim foram escolhidas aplicações de modo a evitar esse problema.

Foram simulados TLBs totalmente associativas dos tamanhos de 8 e 32, com política de substituição LRU-perfeito.

Para cada simulação são apresentados os seguintes resultados, para simulações com TLB de tamanho 32: (a) o número de acessos por página, (b) um histograma do número de páginas acessadas, classificadas por tamanho, e (3) uma comparação do número de TLB *misses* entre a paginação, segmentação e paginação com hiper páginas.

Os dados da execução com segmentação foram obtidos em [Lau17]. Seu simulador é chamado de *segment buffer* (SB) e simula um dispositivo de hardware semelhante a uma TLB, porém utilizado para traduzir segmentos.

A sessão a seguir discute a acurácia das simulações, e nas sessões seguintes são apresentados os resultados dos experimentos para cada programa.

10.1 ACURÁCIA

As simulações apresentam alguma imprecisão, causada por simplificações utilizadas no processo da simulação. A concatenação dos mapas de memória é um dos causadores dessas imprecisões, pois as leituras dos mapas de memória são obtidos com *pmap* a cada 10 milissegundos, e durante a execução do programa real ocorrem mudanças nos mapeamentos e esses mapeamentos podem conter endereços nos quais já havia um segmento alocada anteriormente. Assim, foi necessário descartar alguns segmentos ao montar a TP utilizada nas simulações. O critério adotado para minimizar as imprecisões foi o de descartar o segmento de menor tamanho.

Ao ajustar os *traces*, os endereços que, por causa dos segmentos removidos, não estavam mapeados na TP foram alterados para o endereço zero, e foi criado na TP um mapeamento para este endereço, com tamanho de 15 Bytes. As referências na página zero mostram a quantidade de endereços que não foram traduzidos.

A Tabela 10.1 mostra o número de endereços não traduzidos na simulação de cada programa. Em todas as simulações, o número de referências à página 0 é menor do que 1.31% de todas as referências.

Programa	Não traduzidos	Fração
Python	191.240.948	1.31%
MySQL	116.383	≈ 0%
Libreoffice	106.164.862	0.78%
Firefox	63.900	≈ 0%

Tabela 10.1: Número endereços não traduzidos por programa

10.2 PYHTON

Foi avaliada a execução do programa *pip3* no ambiente de execução Python em sua versão 3.5.2. O programa *pip3* é um gerenciador de pacotes escrito em Python, utilizado para instalar pacotes do *Python Package Index - PyPI* [Lau17].

Foi executado o comando abaixo, a partir de um ambiente no qual não havia nenhum dos pacotes pré-instalados.

```
valgrind --log-socket=127.0.0.1 --trace-children=no
--tool=lackey --trace-mem=yes
pip3 install --upgrade Prospector PyLint Pep8 Flake8
pydocstyle autopep8 jsonschema requests pip
```

Segundo [Lau17], Python foi escolhido para representar uma linguagem interpretada e também porque a linguagem é bastante utilizada no meio acadêmico para pesquisas científicas.

O histograma do tamanho das páginas acessadas (pelo menos uma vez) é mostrado na Figura 10.1. Os tamanhos de páginas de 8 KiB, 16 KiB e 256 KiB são os mais populares, sendo que são 21 páginas com o primeiro tamanho e 11 com o segundo e com o terceiro.

Na Figura 10.2 é possível ver o número total de acessos nas páginas por tamanho. O tamanho de página mais acessado, que detém 65.6% de todos acessos é 4 MiB. O segundo tamanho mais acessada detém apenas 15.2% dos acessos, e é o de 128 KiB. Isso mostra que grande parte do *working set* do programa está localizado nessas páginas, sendo que os demais receberam ≈ 20% dos acessos, com menos de 7% para cada um dos outros tamanhos.

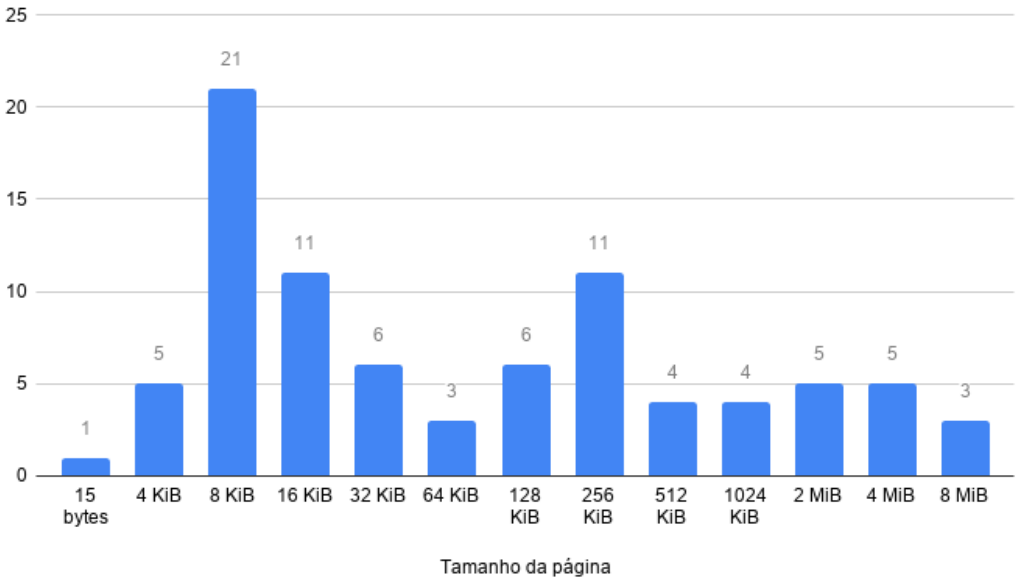

Figura 10.1: Número de páginas de cada por tamanho

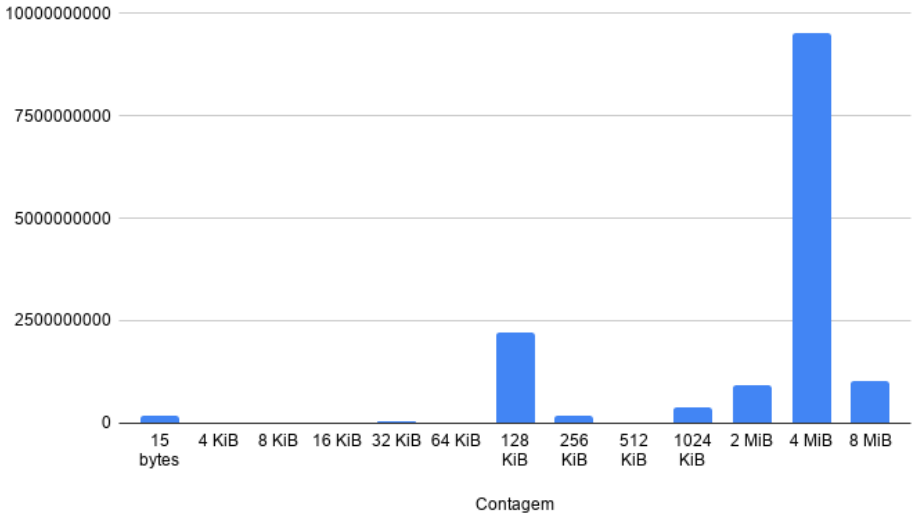

Figura 10.2: Número de acessos por página para cada tamanho

As simulações com os dois tamanhos de TLB de hiper páginas, 8 e 32, apresentaram uma grande diferença na quantidade de *TLB misses* gerados, como pode ser visto na Tabela 10.2. Essa diferença era esperada, haja vista que quanto maior a TLB maior o número de páginas que são mapeadas na *cache*, e não serão buscadas da TP, assim gerando um *TLB miss*.

Capacidade	TLB *misses*
8	113.592.866
32	2.390

Tabela 10.2: Número de *TLB miss* para TLBs de 8 e 32 mapeamentos

A comparação com as simulações de [Lau17], apresentada na Figura 10.3, mostra que a TLB com hiper páginas é 51.450 vezes mais eficiente do que a TLB convencional, e 6 vezes mais eficiente do que a SB.

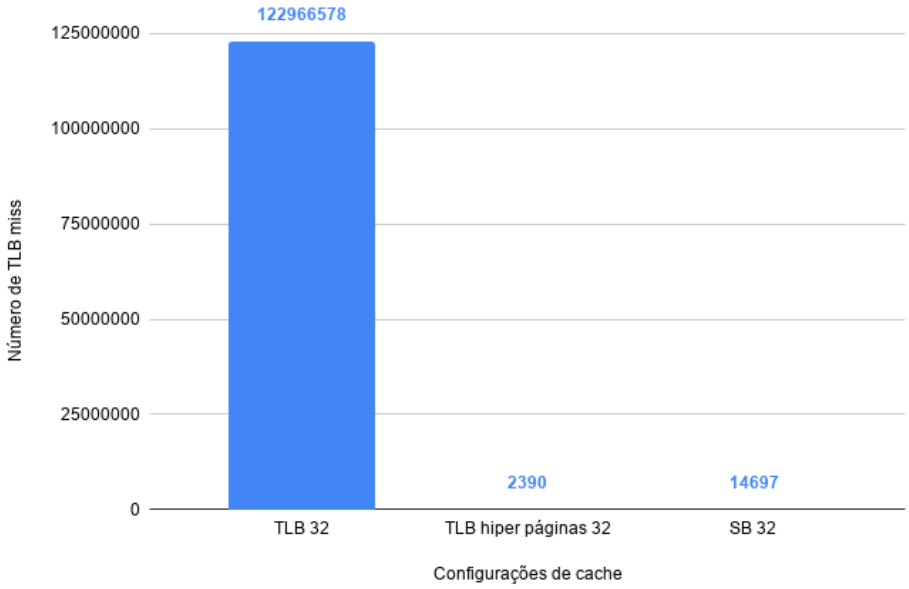

Figura 10.3: Comparação do número de *TLB misses* entre configurações de *cache*

10.3 MYSQL UTILIZANDO SYSBENCH

O MySQL é um SGBD relacional mantido pela *Oracle Corporation*. Seu uso é muito frequente em desenvolvimento web, fazendo parte da arquitetura de desenvolvimento LAMP (Linux,Apache, MySQL, Pearl/PHP/Python) [Lau17]. Segundo [Lau17], ele foi escolhido para representar a classe de aplicações de banco de dados nas simulações.

Os *traces* foram obtidos a partir do utilitário SysBench, que é uma ferramenta de medição de desempenho (*benchmark*) utilizada para avaliar bancos de dados. Segundo [Lau17], ele foi configurado para realizar operações aleatórias em uma tabela com 1.000.000 de linhas utilizando 8 threads simultâneas durante 480 segundos. A tabela utilizada no banco de dados MySQL foi previamente criada utilizando o próprio utilitário SysBench. Foi utilizada a versão 5.7 do MySQL e a versão 0.4.12 do SysBench.

Para a simulação foi executado o comando abaixo.

```
valgrind --log-socket=127.0.0.1 --trace-children=no2 --tool=lackey
--trace-mem=yes3/usr/sbin/mysqld --basedir=/usr
--datadir=/var/lib/mysql4 --plugin-dir=/usr/lib/mysql/plugin5
--log-error=/var/log/mysql/error.log6 --pid-file=/var/run/mysqld/mysqld.pid7
--socket=/var/run/mysqld/mysqld.sock --port=33068--log-syslog=1
--log-syslog-facility=daemon9 --log-syslog-tag
```

O histograma do tamanho das páginas acessadas pelo menos uma vez é mostrado na Figura 10.4. O tamanho de página mais popular é o de 4 KiB, sendo que ele corresponde a 43 páginas. O número de páginas por tamanho é mais equilibrado aqui do que no Python.

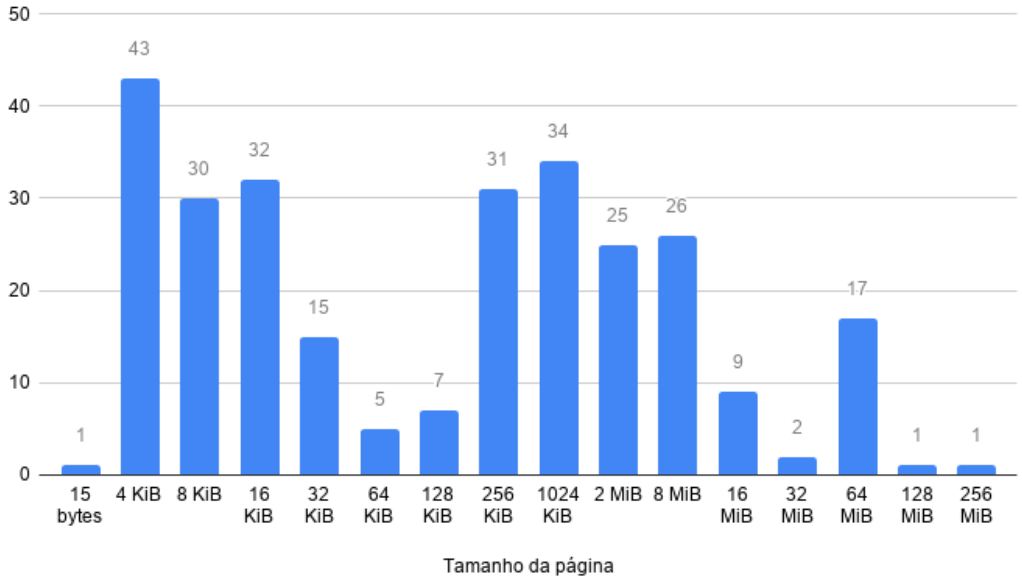

Figura 10.4: Histograma do número de páginas acessadas pelo menos uma vez por tamanho

Na Figura 10.5 é possível ver o número total de acessos em cada tamanho de página. O tamanho de 32 MiB detém 56.8% do total de acessos.

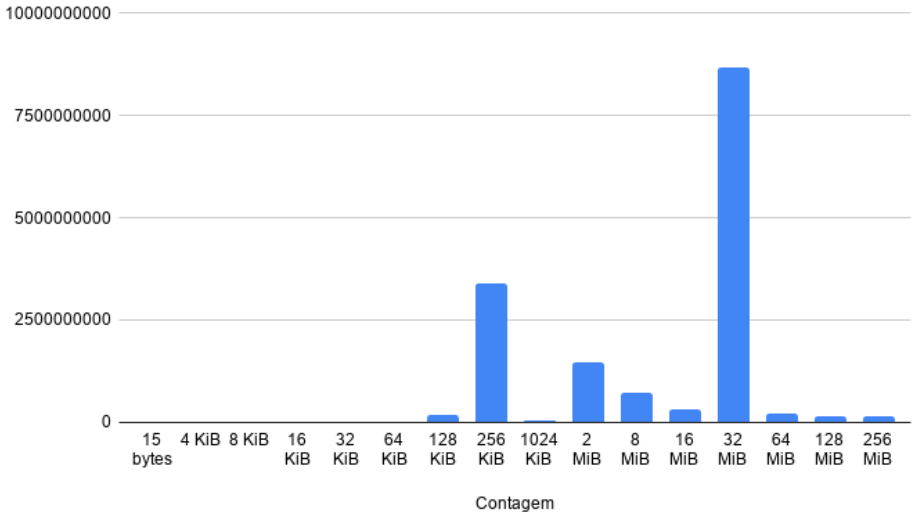

Figura 10.5: Número de acessos por página e seus tamanhos

As simulações com os dois tamanhos de TLB de hiper páginas, com 8 e 32 mapeamentos, apresentaram uma grande diferença na quantidade de *TLB misses* gerados, como pode ser visto na Tabela 10.3.

Capacidade	TLB *misses*
8	36.190.489
32	389.608

Tabela 10.3: Número de *TLB miss* por tamanho de TLB

A comparação com as simulações de [Lau17], apresentada na Figura 10.6, mostra que a TLB com hiper páginas é 279 vezes mais eficiente do que a TLB convencional e 1,29 vezes mais eficiente do que a SB.

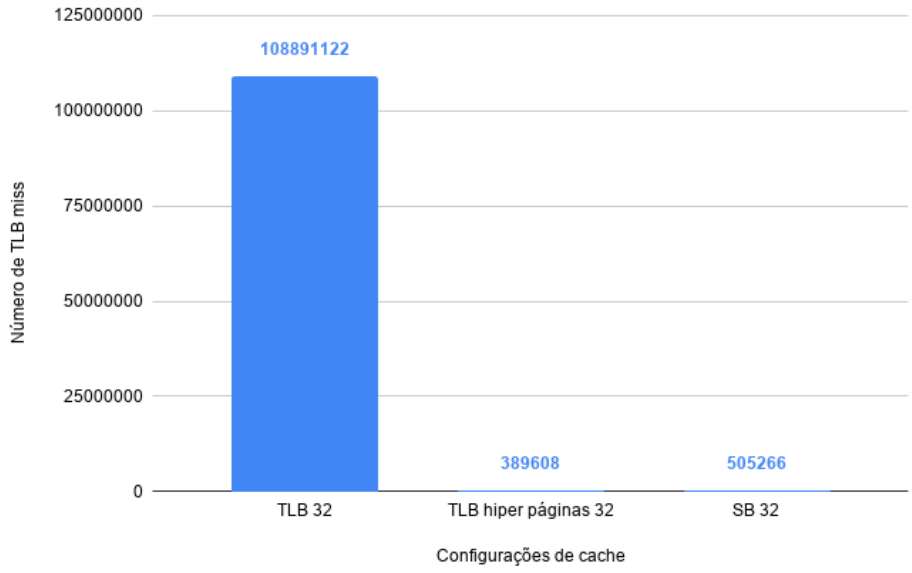

Figura 10.6: Comparação do número de *TLB misses* entre configurações de *cache*

10.4 LIBREOFFICE

O Libreoffice é uma suite de aplicativos em *software* livre para escritório, composta por editor de texto, planilha eletrônica, editor de apresentações, gerenciador de banco de dados, editor de formulas matemáticas e um editor de diagramas. Ele é uma aplicação padrão da maioria das distribuições Linux [Lau17]. Segundo [Lau17], sua escolha se deu pelo fato de ele representar a classe de aplicações de escritório, e também devido a grande quantidade segmentos utilizados.

Os *traces* foram gerados utilizando o editor de texto (*writer*) para abrir uma arquivo no formato Microsoft Word com 314 KiB composto por 13.999 palavras. O comando executado pode ser visto abaixo.

```
1 valgrind --log-socket=127.0.0.1 --trace-children=no2--tool=lackey
2 --trace-mem=yes3/usr/lib/libreoffice/program/soffice.bin --writer test.doc
```

O Libreoffice é um programa que mapeia muito mais páginas do que os anteriores, sendo a página de tamanho 2 MiB a mais popular, com 225 aparições, seguido pelo tamanho 8 KiB, com 160. O histograma com a distribuição dos tamanhos de páginas é mostrado na Figura 10.7.

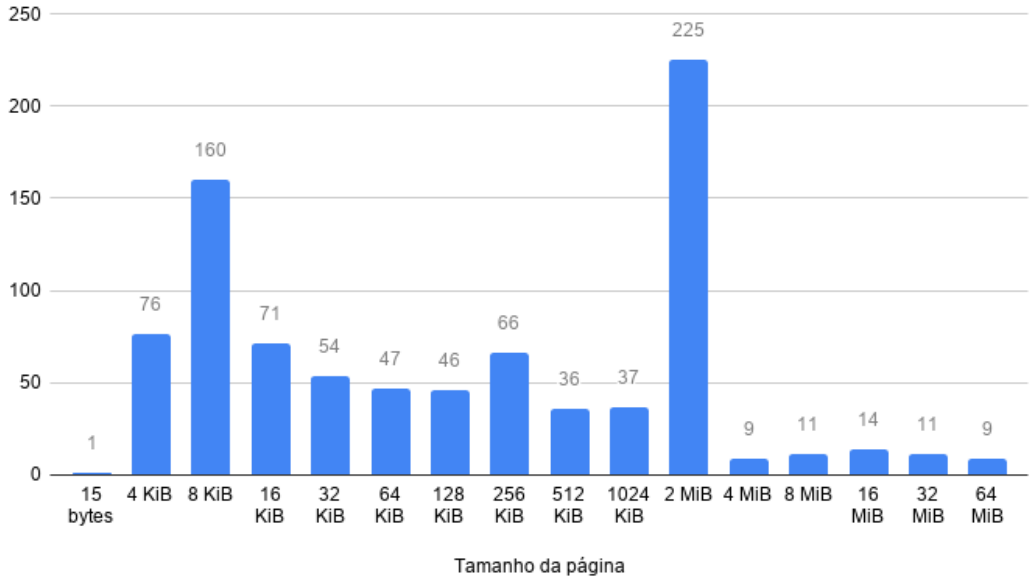

Figura 10.7: Histograma do número de páginas acessadas pelo menos uma vez por tamanho

Na Figura 10.8 é possível ver que a página de tamanho 256 KiB é a mais acessada recebendo 33% dos acessos.

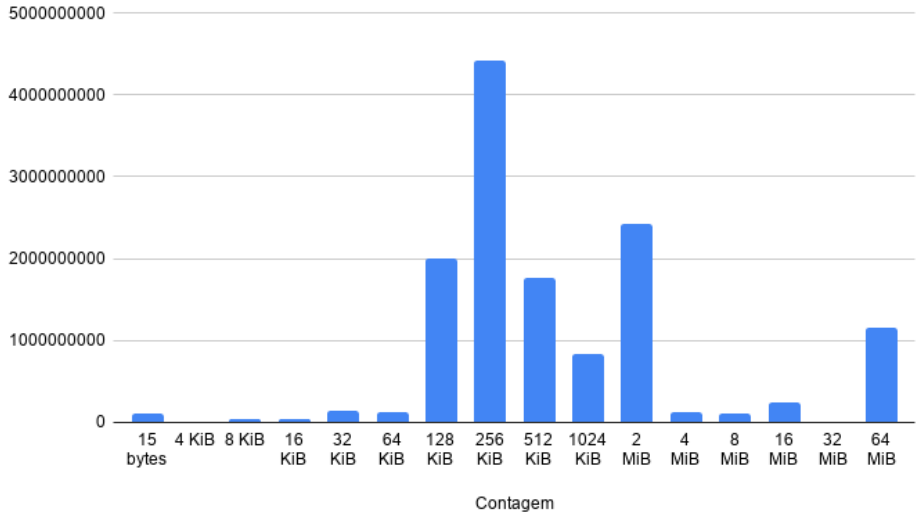

Figura 10.8: Número de acessos por página e seus tamanhos

O número de *TLB misses* das simulações com a TLB de hiper páginas de tamanho 8 e 32 pode ser visto na Tabela 10.4.

Capacidade	TLB *misses*
8	93.848.384
32	6.225.146

Tabela 10.4: Número de *TLB miss* por tamanho de TLB

A comparação com as simulações do [Lau17], apresentada na Figura 10.9, mostra que a TLB com hiper páginas é 4.6 vezes mais eficiente do que a TLB convencional e 1.1 vezes mais eficiente do que a SB.

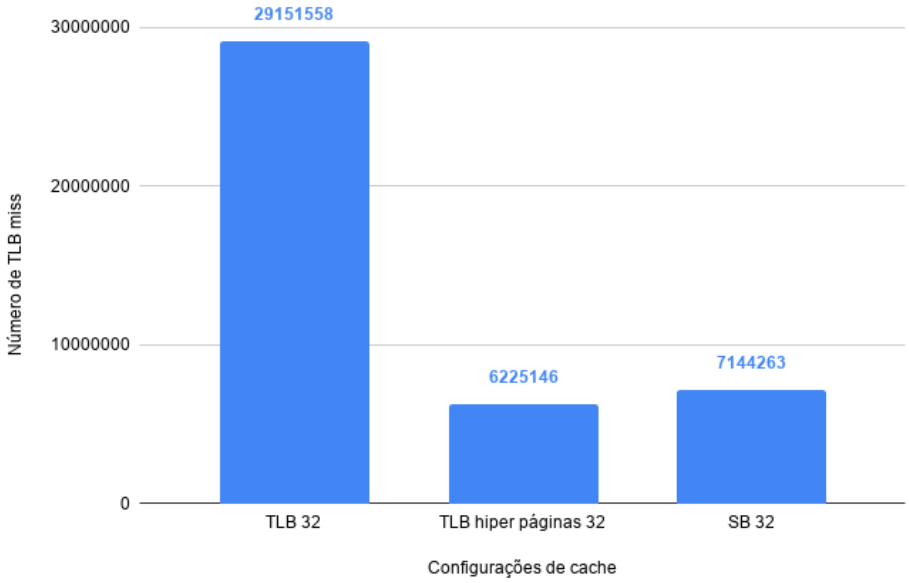

Figura 10.9: Comparação do número de *TLB misses* entre configurações de *cache*

10.5 FIREFOX

O Firefox é um navegador de *software* livre desenvolvido pela Fundação Mozilla. Ele é o navegador padrão da maioria das distribuições Linux, e o segundo navegador para *desktop* mais utilizado do mundo [Lau17].

Os *traces* foram obtidos utilizando o Firefox para abrir a página da Intel no *Youtube*. Segundo [Lau17], essa página foi escolhido pois tem uma carga de trabalho maior do que uma página estática. O comando executado pode ser visto abaixo.

```
1 valgrind --log-socket=127.0.0.1 --trace-children=no
2 --tool=lackey --trace-mem=yes
3 /usr/lib/firefox/firefox http://www.youtube.com/intel
```

No Firefox as páginas de tamanho 8 KiB e 2 MiB são as mais populares, com 151 e 145 páginas respectivamente, como mostra a Figura 10.10.

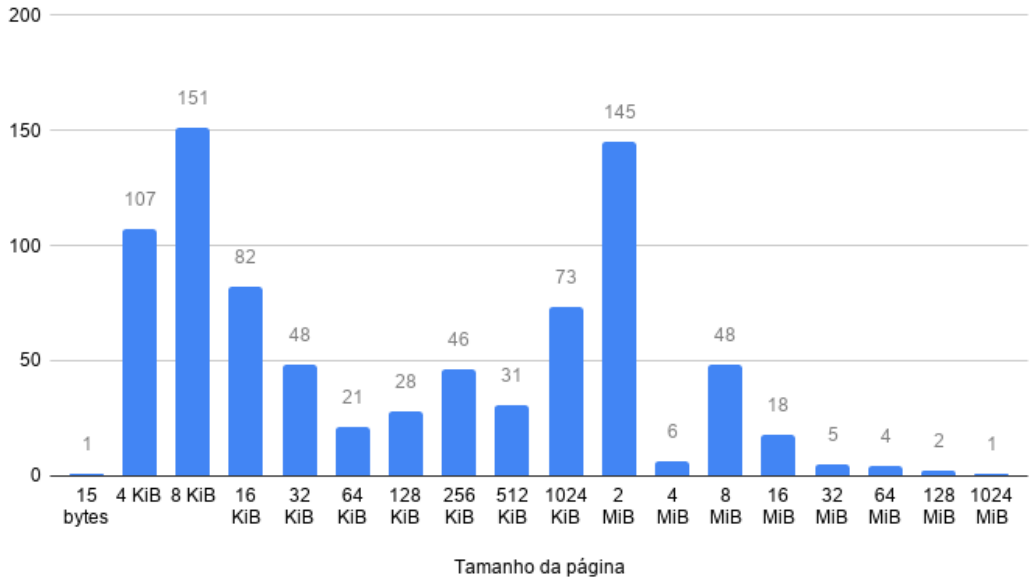

Figura 10.10: Histograma do número de páginas acessadas pelo menos uma vez por tamanho

O tamanho de página com mais acessos é o de 128 MiB, contendo 48.5% do total de acessos, seguida pelo tamanho de 2 MiB com 18.7%, como mostra a Figura 10.11.

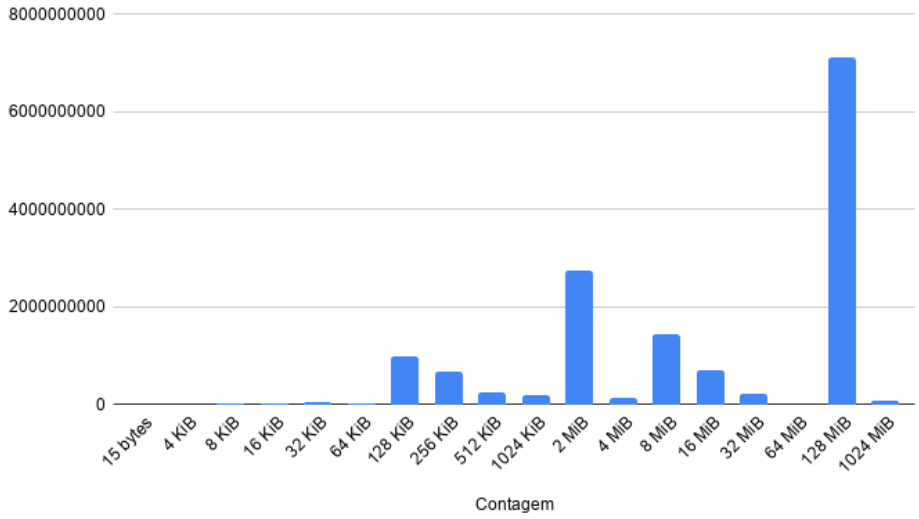

Figura 10.11: Número de acessos por página e seus tamanhos

O número de *TLB misses* das simulações com a TLB de hiper páginas de tamanho 8 e 32 pode ser vista na Tabela 10.5.

Capacidade	TLB *misses*
8	109.492.019
32	3.220.857

Tabela 10.5: Número de *TLB miss* por tamanho de TLB

A comparação com as simulações de [Lau17], apresentada na Figura 10.12, mostra que a TLB com hiper páginas é 16 vezes mais eficiente do que a TLB convencional e 1.1 vezes mais eficiente do que a SB.

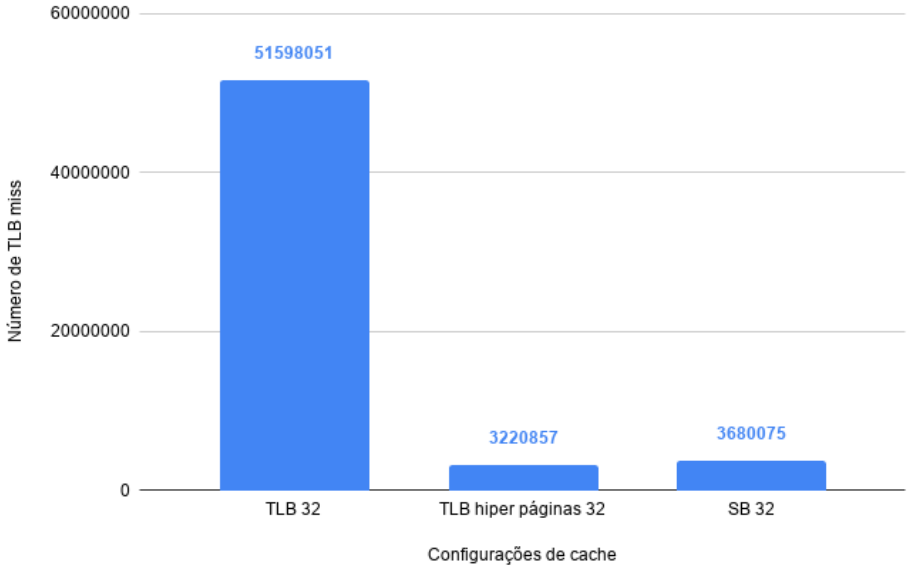

Figura 10.12: Comparação do número de *TLB misses* entre configurações de *cache*

11 DISCUSSÃO DOS RESULTADOS DAS SIMULAÇÕES

Os resultados obtidos sugerem três pontos. O primeiro é que as hiper páginas podem ser uma alternativa à segmentação, dado que os resultados obtidos nos experimentos mostram que seu desempenho é semelhante. O segundo é que como páginas de diversos tamanhos são usadas pelos programas, e não um conjunto pequeno de tamanhos, técnicas como *Huge Pages* não podem substituir as hiper páginas. O terceiro ponto é que o simulador implementado nesse trabalho não têm precisão satisfatória em alguns casos, como as simulações dos programas Pyhton e Libreoffice. Os erros causados pela simulação nesses casos acabam deixando os resultados obtidos inconclusivos, pois o número de endereços não traduzidos supera o número de *TLB misses* da TLB 32. Como os endereços não traduzidos não podem ser contados como *miss* ou como *hit* na TLB, não se pode afirmar nem que as hiper páginas são melhores ou piores nesses casos.

Nos outros programas testados (MySQL e Firefox) o número de endereços não traduzidos é baixo, e os resultados são melhores do que os obtidos com a TLB 32. Quando se compara com a SB 32, os resultados foram similares, sendo a TLB com hiper páginas 1.29 e 1.1 vezes mais eficiente nos testes com MySQL e Firefox, respectivamente. Isso era esperado, pois as hiper páginas são alocadas de forma a assumir o tamanho dos segmentos, sendo, na maioria das vezes, maiores do que os segmentos lógicos pela limitação de seus tamanhos, que devem ser potências de 2.

12 CONCLUSÃO

O crescimento da quantidade de memória física instalada nos computadores, o desenvolvimento de memória secundária mais rápidas (como SSDs) e processos que utilizam grandes quantidades de memória, fazem com que as premissas que justificavam a utilização de memória paginada não sejam mais verdadeiras.

Em um futuro próximo a quantidade de memoria disponível nos computadores chegará a 2^{64} bytes, e técnicas de gerenciamento de memória diferentes da paginação passarão a ser mais vantajosas. A segmentação, que foi utilizada na passado, volta a ser considerada, e em [Lau17] são demonstrados ganhos de desempenho consideráveis dessa técnica com relação à paginação. Além do menor número de *SB misses*, também há a vantagem da SB ser muito menor do que uma TP, pois para cada segmento lógico são mapeadas de uma a muitas páginas.

No decorrer desse trabalho foi implementado um simulador para uma técnica de gerenciamento de memória que é "intermediária"entre a paginação e a segmentação, chamada de *hiper páginas*. Nessa técnica as páginas têm tamanhos variados, sendo esses potências de 2. Esta técnica foi pensada para ser mais fácil de implementar do que a segmentação, pois apesar de mudanças no hardware e dos compiladores terem que ser re-escritos nas duas técnicas, as alterações no SO para suportar a segmentação seriam muito maiores do que para suportar as hiper páginas, pois essa é muito mais próxima da paginação.

O simulador de hiper páginas foi construído tomando como base o simulador implementado em [Lau17]. Os *traces* e mapas de memória são os mesmos, para permitir que fossem comparados os resultados.

Para fazer a simulação, primeiramente são concatenados os mapas de memória, para que seja criada um TP estática que é usada durante a simulação. Então são eliminados mapeamentos repetidos, e caso haja alguma sobreposição o maior mapeamento é mantido. Após isso é necessário transformar esses mapeamentos em uma TP, alterando os tamanhos das páginas para potencias de 2, e ajustando-os caso haja alguma sobreposição, "empurrando"os mapeamentos para endereços mais altos. Esse ajustes fazem com que os *traces* precisem ser alterados, recebendo esses deslocamentos que foram aplicados para ficarem coerentes com a TP. Com a TP e os *traces* ajustados é possível executar a simulação, que referencia os endereços dos *traces* em um TLB com hiper páginas, e devolve o número de *TLB misses* e o número de referências em cada página. Esses números foram comparados com os obtidos na simulação de uma SB em [Lau17].

Foram simulados quatro programas: Python, Libreoffice, MySQL e Firefox. Eles foram escolhidos por terem cargas de trabalhos diferentes, que representam tipos diversos de utilização de uma máquina. As hiper páginas tiveram um desempenho satisfatório nas simulações dos programas MySQL e Firefox, muito próximas a SB se considerada a margem de erro, sendo 1.29 e 1.1 vezes mais eficientes do que a segmentação, respectivamente. Nas simulações dos programas

Python e Libreoffice os resultados são inconclusivos, pois o erro originário das simplificações adotadas, e descritas na Sessão 10.1, foi relativamente grande.

São necessários testes mais conclusivos para que se possa saber o real ganho de desempenho das hiper páginas, e se os custos dessa solução "intermediária" compensam as alterações em *hardware* e *software*. Para isso é preciso que seja implementado um simulador mais robusto de um sistema com hiper páginas, aos moldes do construído em [Lau17], que tenha uma TP dinâmica para que sejam diminuídos os erros de simulação. Pode-se também testar uma implementação em VHDL de uma TLB com hiper páginas, sendo essa a maneira mais precisa de avaliar seu desempenho.

REFERÊNCIAS

[Den70] Peter J. Denning. Virtual memory. *ACM Comput. Surv.*, 2(3), September 1970.

[GS98] Narayanan Ganapathy and Curt Schimmel. General purpose operating system support for multiple page sizes. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '98, pages 8–8, Berkeley, CA, USA, 1998. USENIX Association.

[gw19] geeksforgeeks website. Segmentation in operating system, 2019. `https://www.geeksforgeeks.org/segmentation-in-operating-system/`.

[HCG+15] Peter Hornyack, Luis Ceze, Steve Gribble, Dan R. K. Ports, and Hank Levy. A study of virtual memory usage and implications for large memory. 2015.

[Hex16] Roberto André Hexsel. Projeto de caches, 2016. `http://www.inf.ufpr.br/roberto/ci212/slides/s07hier-2.pdf`.

[Hex19] Roberto André Hexsel. Ci064 – software básico. August 2019.

[Lau17] Lauri Paulo Laux Jr. De volta ao passado: Memória virtual com segmentaçõ p ara máquinas com memória RAM 'infinita'. Dissertação de mestrado, Departamento de Informática, UFPR, December 2017. `<http://www.inf.ufpr.br/roberto/dissLauri.pdf>`.

[LH16] Lauri P Laux Jr and R A Hexsel. Back to the past: Segmentation with infinite and non-volatile memory. pages 278–289, October 2016.

[LH18] Lauri P Laux Jr and R A Hexsel. Back to the past: When segmentation is more efficient than p aging. pages 278–289, August 2018.

[Maz19] Carlos A. Maziero. Conceitos básicos, 2019. `http://wiki.inf.ufpr.br/maziero/lib/exe/fetch.php?media=socm:socm-texto-14.pdf`.

[mw19] multics website. Multics, 2019. `https://multicians.org/`.

[O9] Michael O'Boyle. Paging, 2019. `https://www.inf.ed.ac.uk/teaching/courses/os/slides/10-paging16.pdf`.

[PPG18] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 679–692, New York, NY, USA, 2018. ACM.

[TB14] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 4th edition, 2014.