

ORGANIZAÇÃO E ARQUITETURA DE COMPUTADORES III

UNIDADE: HIERARQUIA DE MEMÓRIA

Conteúdo:

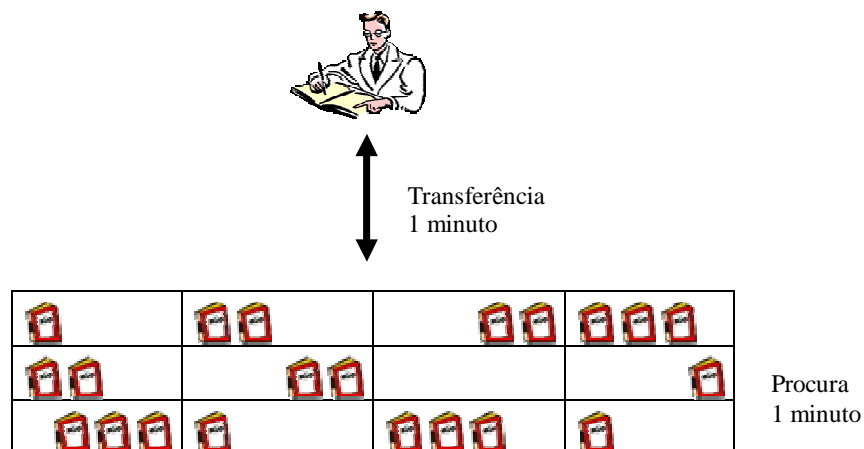
1 INTRODUÇÃO.....	2
1.1 Exemplo da Biblioteca	2
2. HIERARQUIA DE MEMÓRIA	4
3 GERÊNCIA DE MEMÓRIAS <i>CACHE</i>	6
3.1 Mapeamento de endereços em memória <i>cache</i>	6
3.1.1 Mapeamento Direto	6
3.1.2 Mapeamento Associativo	10
3.1.3 Mapeamento Conjunto associativo	12
Exercícios de dimensionamento de memórias <i>cache</i>	16
<i>Cache</i> com mapeamento conjunto associativo (4 conjuntos)	16
<i>Cache</i> com mapeamento totalmente associativo.....	17
<i>Cache</i> com mapeamento direto	17
<i>Cache</i> com mapeamento conjunto associativo (16 conjuntos)	18
<i>Cache</i> com mapeamento totalmente associativo.....	19
<i>Cache</i> com mapeamento direto	19
3.2 Integridade dos dados na <i>cache</i>	20
3.2.1 Write-through (escrevo através).....	21
3.2.2 Write-back (escrevo de volta)	21
3.3 Passos para escrita e leitura na <i>cache</i>	21
3.3.1 Leitura	22
3.3.2 Escrita	22
4 GERÊNCIA DA MEMÓRIA PRINCIPAL	22
4.1 Histórico da gerência de memória	23
4.2 O problema da fragmentação de memória	24
4.3 Endereçamento da memória principal	25
4.3.1 Endereçamento contíguo.....	25
4.3.2 Endereçamento não-contíguo.....	26
4.3.3 Otimização do endereçamento não-contíguo (TLB)	33
4.4 Memória Virtual.....	34
4.5 Estudo de Casos.....	36
4.5.1 UNIX e SOLARIS	36
4.5.2 Linux	36
4.5.2 IBM OS/2 (hardware Intel)	37
4.5.3 Windows 2000	37
5 EXERCÍCIOS.....	38

1 INTRODUÇÃO

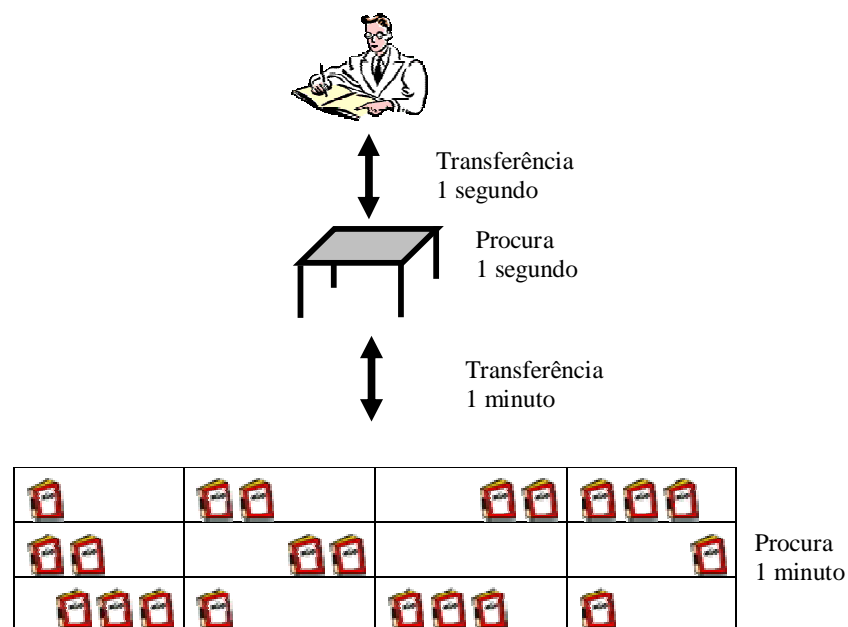
- Nos últimos anos vem se investindo muito no aumento da velocidade dos processadores, que está ocorrendo de forma significativa
- Porém, a velocidade de processamento de um sistema não é determinada somente pela velocidade do processador
- Pouco adianta um processador muito rápido se a alimentação deste processador com dados não conseguir acompanhar, pelo menos aproximadamente, o mesmo ritmo
- Como tanto o fornecimento de dados, como o seu posterior armazenamento após o processamento são efetuados na memória, a velocidade média de acesso a memória é um componente importante no cálculo da velocidade de processamento de um sistema
- Além da velocidade, o tamanho da memória é importante, já que ela funciona como uma área de armazenamento temporário para dados que serão armazenados na memória secundária (discos rígidos, óticos, etc.), que é ainda mais lenta
- Ideal seria:
 - Memória de tamanho ilimitado
 - Memória com um tempo de acesso muito rápido
- **Objetivos são contraditórios**
 - Por problemas tecnológicos, quanto maior a memória mais lento será o seu tempo de acesso o que faria com que o caso ideal, com a atual tecnologia, não possa ser alcançado
- Solução: criar uma ilusão para o processador de forma que a memória pareça ilimitada e muito rápida.

1.1 Exemplo da Biblioteca

- Um estudante recebe a tarefa de redigir um trabalho sobre os Sistemas Operacionais encontrados no mercado e suas principais características
- Ele se dirige a biblioteca, senta em uma cadeira com braço (capacidade um único livro) e inicia sua pesquisa



- O algoritmo de acesso neste caso é o seguinte:
 1. ir até a estante de livros
 2. procurar livro desejado
 3. levar livro até a cadeira
 4. consultar livro
 5. se não terminou ir para 1
- Considerando que o aluno precisa consultar 10 livros e que leva 1 minuto para buscar cada livro e 1 minuto procurando o livro desejado na estante, ele perderia 20 minutos para fazer o trabalho
- Isso se não tiver se esquecido de consultar alguma coisa e necessitar buscar novamente um livro que já consultou, o que custaria **2 minutos** adicionais por livro
- Agora o aluno procura uma mesa vazia

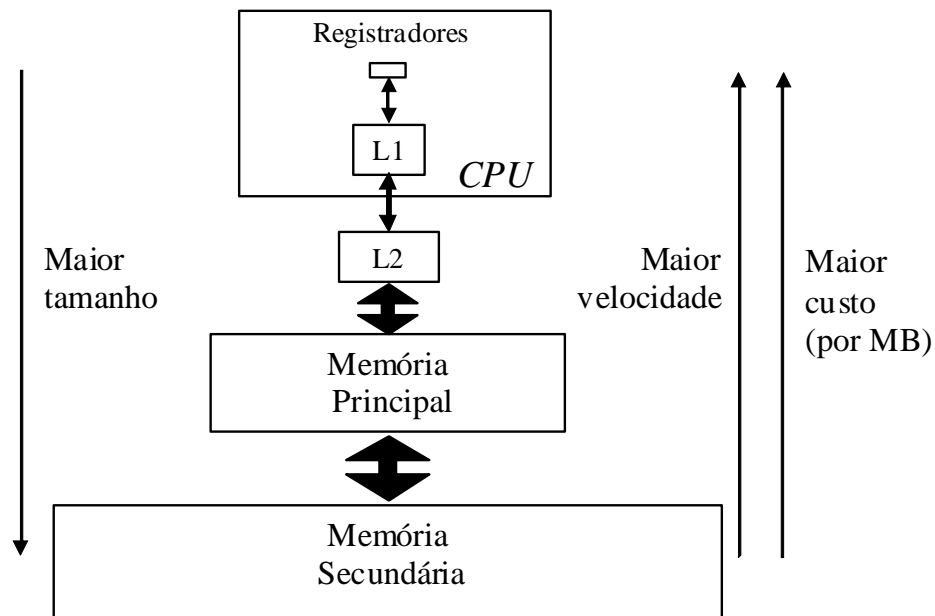


- O algoritmo de acesso neste caso é o seguinte:
 1. ir até a estante de livros
 2. procurar livros desejados
 3. levar livros até a mesa
 4. consultar livros
 5. se não terminou ir para 1
- Assumindo que a mesa tenha espaço suficiente e ele consiga carregar todos os livros que precisa, o aluno pode buscar os 10 livros de uma vez o que custaria 11 minutos (10 para a procura dos livros e 1 para a transferência). Depois são necessários mais 20 segundos para a consulta dos livros na mesa (1 para a consulta e 1 para a transferência de cada livro), totalizando 11 minutos e 20 segundos
- Se tiver que carregar os livros individualmente, a consulta de cada livro passa a custar até mais do que no exemplo anterior, já que a mesa adiciona um custo intermediário de 1 segundo para a procura e um segundo para a transferência (2 m. e 2 s. por livro)

- No entanto, no caso de ter se esquecido de algo, o custo para consultar novamente um livro que ainda está na mesa cai significativamente para apenas **2 segundos!**
- Ou seja, neste caso, o custo da **reutilização** ficou muito menor (se os livros desejados ainda estiverem na mesa)
- Quanto mais reutilizar os livros, mais cai o tempo médio de acesso das informações, pois o alto custo inicial para trazer os livros é compensado pelas várias reutilizações com custo menor
- **Situações complicadoras**
 - todos os livros que quero não cabem na mesa
 - vou tomar um café e chega um colega e pega a mesa (só que o trabalho dele é sobre outro assunto – mesa vai estar cheia de livros diferentes)
- Posso ampliar o exemplo incluindo mais um nível intermediário: uma mesinha entre a mesa maior e o aluno. Na mesinha são colocados os três livros que interessam mais para o trabalho. O acesso a mesinha é mais rápido que a mesa, mas a capacidade é menor
- **Porque o tempo de acesso melhora na média? ⇒ Localidade**
 - trabalho restrito a um grupo de livros. Se o trabalho fosse catalogar todos os livros da estante, sempre estaria tendo que consultar livros que não estariam na mesa e teria que buscá-los na estante (**Localidade Espacial**)
 - de tempos em tempos noto que esqueci de algum dado de um determinado sistema e volto a consultar um livro que já tinha consultado antes (**Localidade Temporal**)

2. HIERARQUIA DE MEMÓRIA

- Ilusão de uma memória ilimitada e rápida obtida através da utilização de diversos níveis de acesso (mesmo princípio do exemplo da biblioteca - Localidade)
- **Funciona porque programas que executam na CPU também possuem localidade espacial e temporal**
 - se um endereço foi referenciado, existe grande probabilidade do endereço seguinte ser referenciado em pouco tempo, Ex: Execução seqüencial de código e vetores de dados (**Localidade Espacial**)
 - se um endereço foi referenciado, existe grande probabilidade de ser referenciado novamente em pouco tempo, Ex: Laços de código e variáveis globais (**Localidade Temporal**)
- Níveis intermediários usados para amortizar a diferença de velocidade entre processador e memória
- Memória já aplica há décadas o mesmo princípio para acelerar o acesso ao disco rígido (memória secundária)
- Os dados são transferidos para níveis mais altos à medida que são usados (por demanda), como no exemplo da biblioteca
- Transferência entre níveis feita com grupos de palavras (bloco, página), pois o custo relativo de transferir um grupo de dados é menor do que para uma única palavra, além de já antecipar acessos (explorando a existência de localidade espacial)



- Comparação entre os diferentes tipos de memória

Tipo	Tempo de Acesso	Tamanho	Custo (por MB)
Registradores	Ciclos CPU	32-64 bits	---
L1	Ciclos CPU	32-64 KBytes	---
L2	8-35 ns	512KBytes -2 MBytes	50 Us\$
L3	35-50 ns	2 MBytes – 8 MBytes	30 Us\$
principal	50-120 ns	512 MBytes – 4 GBytes	1 Us\$
secundária	5 ms	40 GBytes-4 TBytes	0.02 Us\$

- Para movimentação de dados entre níveis necessita **Mecanismos** e para algumas decisões estratégicas que tem que ser tomadas os mecanismos usam **Políticas**
 - Ex: preciso mover dados para um nível superior que já está cheio. Quem retirar?
 - Decisão errada (ou ruim) pode afetar desempenho do sistema como um todo.
- Se os mecanismos conseguirem manter a os dados usados pelo processador nos níveis mais altos na maior parte dos acessos, o tempo médio de acesso a memória como um todo será próximo do tempo de acesso a estes níveis
- Algumas definições importantes

Hit – dado encontrado no nível procurado

Miss – dado não encontrado no nível procurado

Hit-rate (ratio) – percentual de hits no nível, Ex: 70%

Miss-rate (ratio) – percentual de misses no nível, Ex: 30% (complementar ao Hit-rate)

Hit-time – tempo de acesso ao nível incluindo tempo de ver se é hit ou miss

Miss-penalty – tempo médio gasto para que o dado não encontrado no nível seja transferido dos níveis mais baixos (inclui tempo de substituição caso necessário)

- Exercício: calcule o tempo médio efetivo de acesso a uma memória *cache* considerando Hit-ratio = 80%, Hit-time = 2 μ s e Miss-penalty = 10 μ s

$$T_{me} = Hit-time + (1 - Hit-rate) * Miss-penalty$$

$$T_{me} = 2 + (1 - 0.8) * 10 = 2 + 0.2 * 10 = 2 + 2 = 4 \mu s$$

- Vamos analisar os níveis de *cache* e de memória principal com mais detalhes

3 GERÊNCIA DE MEMÓRIAS CACHE

- Significado da palavra *cache* – Lugar seguro para guardar (esconder) coisas (francês)
- Considerando que a *cache* só pode ter parte dos dados do nível mais abaixo, por causa do menor tamanho, temos dois problemas:
 - Como identificar se o dado procurado está na *cache* e
 - Se ele estiver na *cache*, como acessá-lo de forma rápida
- O processador não sabe que está sendo enganado e gera um endereço para um espaço de endereçamento que não necessariamente existe fisicamente
- Endereçar a *cache* com este endereço não faz nenhum sentido (na maioria dos casos a *cache* nem teria essa posição gerada por ser muito menor)
- Uma varredura seqüencial também não é uma solução aceitável, pelo tempo que levaria (não esquecer que o objetivo é acelerar o acesso)

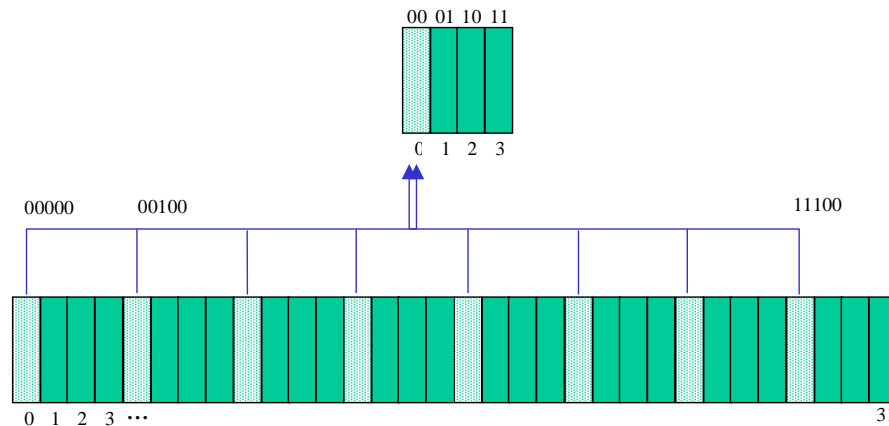
3.1 Mapeamento de endereços em memória *cache*

- O termo **mapeamento** é usado para indicar o relacionamento dos dados do nível inferior com as posições da memória *cache* (como o endereço fornecido pelo processador tem que ser transformado para poder ser usado no acesso a uma memória *cache*)
- Serão vistas três formas de mapeamento de memórias *cache*
 - Direto
 - Associativo
 - Conjunto associativo

3.1.1 Mapeamento Direto

- Forma mais simples de mapeamento
- Posição na *cache* depende do endereço da palavra
- Cada palavra possui uma posição fixa na *cache*
- Como tenho menos espaço na *cache* do que no nível inferior, um grupo de palavras é mapeado na mesma posição da *cache*

- Este mapeamento é dado diretamente através de uma operação no endereço que se está procurando
 - Exemplo de operação: endereço \bmod número de posições da *cache* (módulo é o resto da divisão inteira)
 - Para uma *cache* de 4 posições e uma memória com 32 endereços teríamos (palavra de 8 bits):

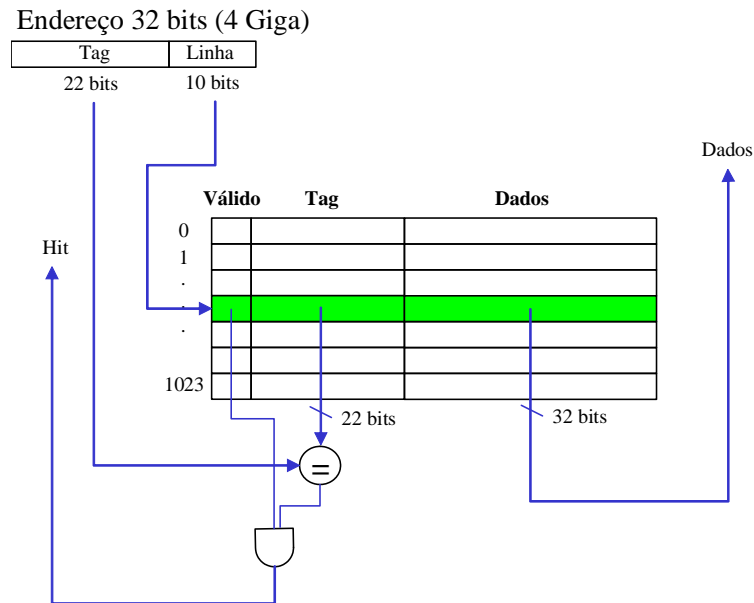


- Cada posição da *cache* pode ter 8 posições da memória
 - Pergunta: como podemos obter rapidamente o mapeamento? Utilizando os dois bits menos significativos do endereço
- Pergunta: Como saber qual das possíveis palavras está realmente na *cache*? É necessário um **TAG** (rótulo, etiqueta) de identificação para cada posição da *cache*
- No exemplo poderia usar os bits mais significativos que sobraram
- Pergunta: Só isto já basta? Não, ainda é necessário um **bit de validade** que indique se a posição da *cache* está ocupada ou se contém lixo
- Dessa forma a *cache* de mapeamento direto do exemplo tem várias **linhas** com a seguinte estrutura

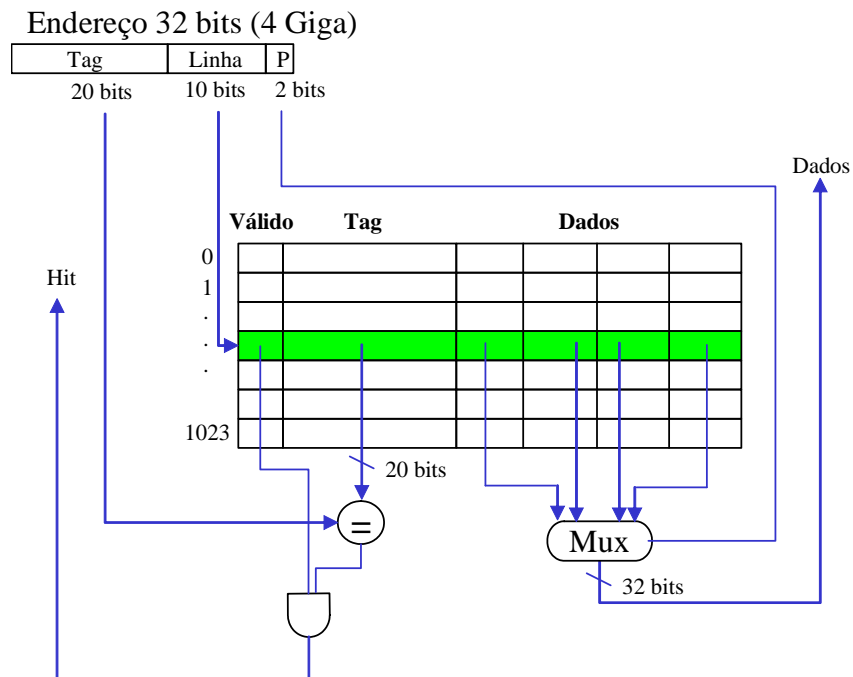
Bit de Validade	<u>Tag</u>	<u>Dado</u>
1	001	00110110
0		
0		
1	000	11100011

- Passos para um acesso
 1. Calculo o módulo do endereço que procuro pelo número de posições/linhas da *cache* (ou uso os bits menos significativos do endereço)
 2. Verifico o bit de validade da posição da *cache* correspondente e se for inválido acuso *miss* (vou para 4), senão verifico o Tag
 3. Se Tag diferente do endereço procurado acuso *miss* (vou para 4), senão tenho *hit* e leio a posição/linha (fim)
 4. Busco o dado no nível inferior coloco na posição/linha e efetuo a leitura (fim)

- Divisão de bits no registrador de endereçamento
 - Exemplo de uma *cache* com 1024 linhas (2^{10}) com palavra de 32 bits



- Neste caso o espaço de endereçamento tem 4 Gigabytes (2^{32})
- Pergunta: Já vimos que na realidade são transferidos blocos entre níveis. Como ficaria a divisão de bits neste caso?
- Exemplo de uma *cache* com 1024 linhas (bloco com 4 palavras de 32 bits)



- Pergunta: Considere agora um espaço de endereçamento de 1 Giga. Como ficaria a divisão de bits para uma *cache* de 2048 linhas que trabalhe com blocos de 8 palavras?

Endereço de 30 bits (1 Giga)

16 (Tag)	11 (Linha)	3 (Palavra)
----------	------------	-------------

- Pergunta: Quanto se tem efetivamente de dados nessa *cache*? (Bit de validade + Tag + Dados) * linhas = (1+16+(8*32)) = 273 bits dos quais 8*32 = 256 são dados. $256 * 100 / 273 = 93.77\%$ (regra de três para uma linha)

Linha da *Cache*

1 (validade)	16 (tag)	8*32 (bloco de dados)
--------------	----------	-----------------------

- Exercício de mapeamento – os parênteses indicam o conteúdo da posição

Cache Mapeamento Direto					
MP	H	Conteúdo da <i>cache</i>			
	M	0	1	2	3
154	M			(154)	
68	M	(68)		"	
34	M	"		(34)	
67	M	"		"	(67)
154	M	"		(154)	"
100	M	(100)		"	"
67	H	"		"	(67)
68	M	(68)		"	"
69	M	"	(69)	"	"
70	M	"	"	(70)	"
68	H	(68)	"	"	"
34	M	(68)	(69)	(34)	(67)
	2				

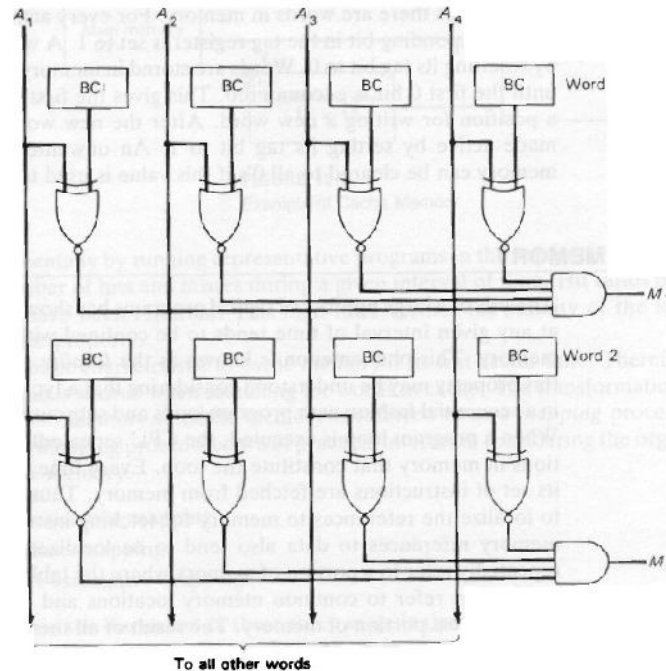
- Vantagens/Desvantagens dessa técnica de mapeamento

↑	Barato (hardware)	↓	Posso ter um mau aproveitamento das linhas da <i>cache</i> (dependendo dos endereços gerados)
	Procura é simples (posição fixa calculada)		
	Escolha da vítima não existe (é dada pelo módulo)		Uso parte da área da <i>cache</i> para controle
	Simplicidade / Velocidade		

- Como poderia melhorar o mapeamento apresentado? Como retirar a dependência entre o endereço na memória e a posição na *cache* sem comprometer o desempenho da procura? Uso **hardware** para acelerar procura.

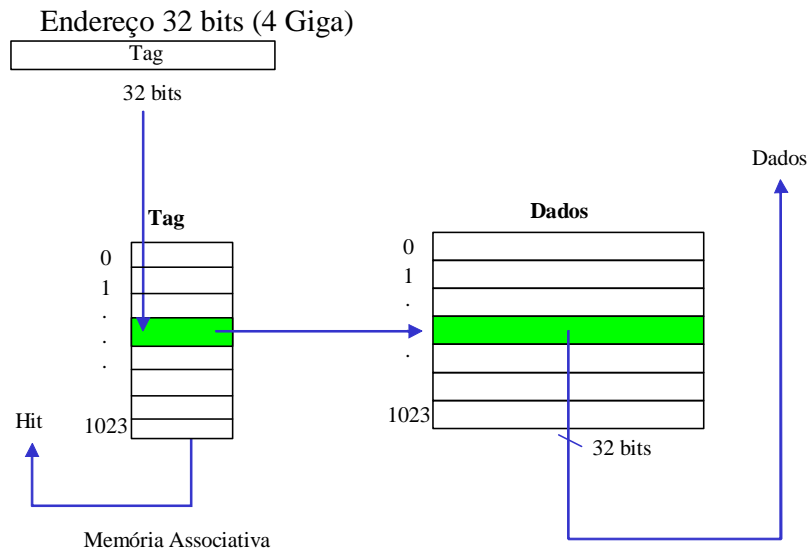
3.1.2 Mapeamento Associativo

- Endereço da memória em qualquer endereço da *cache* (100% de aproveitamento).
Conseqüências:
 - Tenho que fazer procura
 - Preciso política de substituição (Quando tenho *miss* e busco no nível mais abaixo, caso a *cache* já esteja cheia quem tirar para abrir lugar?)
- Solução para a procura: procura em paralelo. Uso memória associativa
 - Memória cara e de tamanho limitado

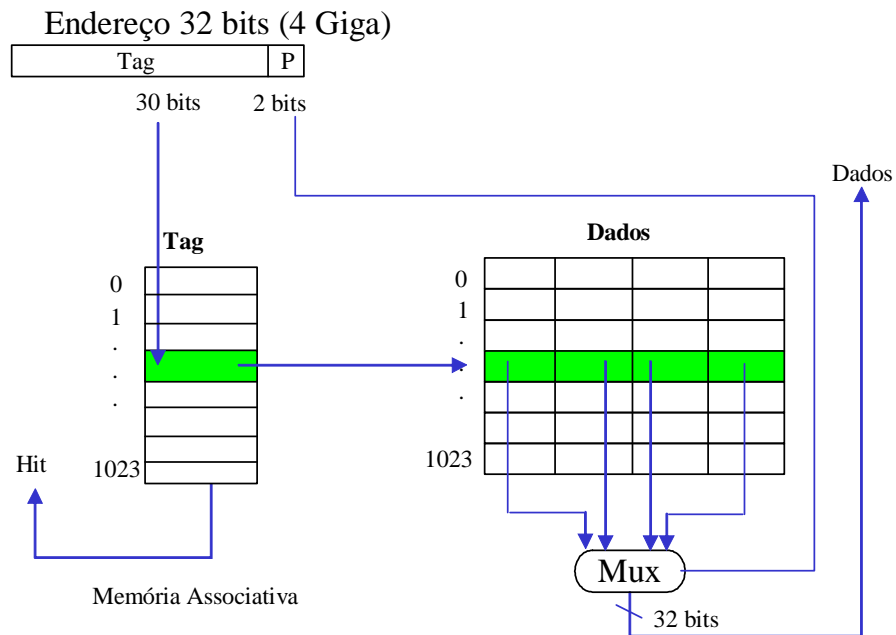


- Solução para a substituição: uso política
- Possibilidades
 - Randômica: escolha aleatoriamente uma posição a ser substituída
 - LFU (*Least Frequent Used*) - a posição da *cache* que foi usada menos vezes será substituída (menos frequentemente usada) – preciso incrementar um contador a cada acesso e comparação para escolha
 - LRU (*Least Recent Used*) - a posição da *cache* que foi usada a mais tempo será substituída (menos recentemente usada) – preciso incrementar um contador a cada acesso e comparação para escolha
- Quero aproximação do melhor caso sem perder tempo ...
- Passos para um acesso
 1. Alimento a memória associativa com o Tag procurado
 2. Se o Tag procurado não está na memória associativa tenho *miss* (vou para 4)
 3. Senão tenho *hit* e acesso a memória *cache* com o índice fornecido pela memória associativa e efetuo a leitura (fim)
 4. Se não existir posição livre na *cache* escolho um endereço para substituir (LRU)

5. Busco o endereço procurado no nível mais baixo e coloco em uma posição livre (ou escolhida) da *cache* cadastrando essa posição e Tag na memória associativa e efetuo a leitura (fim)
- Divisão de bits no registrador de endereçamento
 - Exemplo de uma *cache* com 1024 linhas (2^{10}) com palavra de 32 bits
 - Neste caso o espaço de endereçamento tem 4 Gigabytes (2^{32})



- Pergunta: Já vimos que na realidade são transferidos blocos entre níveis. Como ficaria o Hardware neste caso?
- Exemplo de uma *cache* com 1024 linhas (bloco com 4 palavras de 32 bits)



- Pergunta: Considere agora um espaço de endereçamento de 256 Mega. Como ficaria a divisão de bits do endereço para uma *cache* de 2048 linhas que trabalhe com blocos de 8 palavras ?

Endereço de 28 bits (256 Mega)

25 (Tag)	3 (Palavra)
----------	-------------

- Pergunta: Quanto se tem efetivamente de dados nessa *cache*? 100%
- Pergunta: Qual o tamanho da memória associativa? $2048 * 25 \text{ (tag)} = 51200 \text{ bits} / 8 = 6400 \text{ bytes} / 1024 = 6,25 \text{ Kbytes}$
- Exercício de mapeamento – os parênteses indicam o conteúdo da posição

Cache Mapeamento Associativo

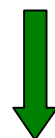
MP	H	Conteúdo da <i>cache</i>			
		0	1	2	3
154	M	(154)			
68	M	"	(68)		
34	M	"	"	(34)	
67	M	"	"	"	(67)
154	H	(154)	"	"	"
100	M	"	(100)	"	"
67	H	"	"	"	(67)
68	M	"	"	(68)	"
69	M	(69)	"	"	"
70	M	"	(70)	"	"
68	H	"	"	(68)	"
34	M	(69)	(70)	(68)	(34)
	3				

- Vantagens/Desvantagens dessa técnica de mapeamento



Melhor aproveitamento das linhas da *cache*, depois de cheia, 100% de aproveitamento

Dados de controle não ocupam espaço da *cache* (estão em área separada)



Memória associativa tem alto custo e tamanho limitado

Limita o número de linhas da *cache*

Necessito política de substituição

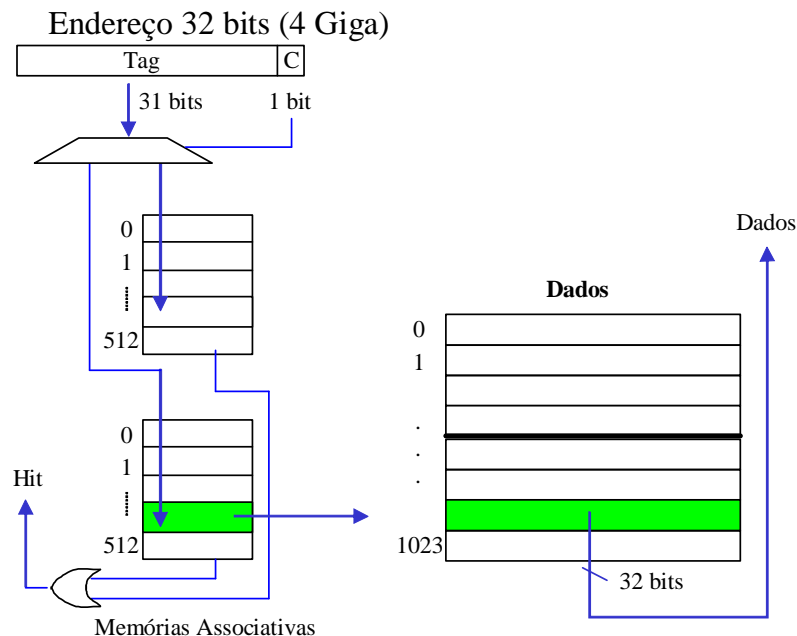
- Custa tempo
- Pode escolher mal

- Limite de tamanho da *cache* por causa da memória associativa é uma restrição muito forte, já que uma das tendências hoje é exatamente o aumento do tamanho da *cache*
- Qual seria outra possibilidade? Como poderia juntar as vantagens dos dois métodos vistos até agora?

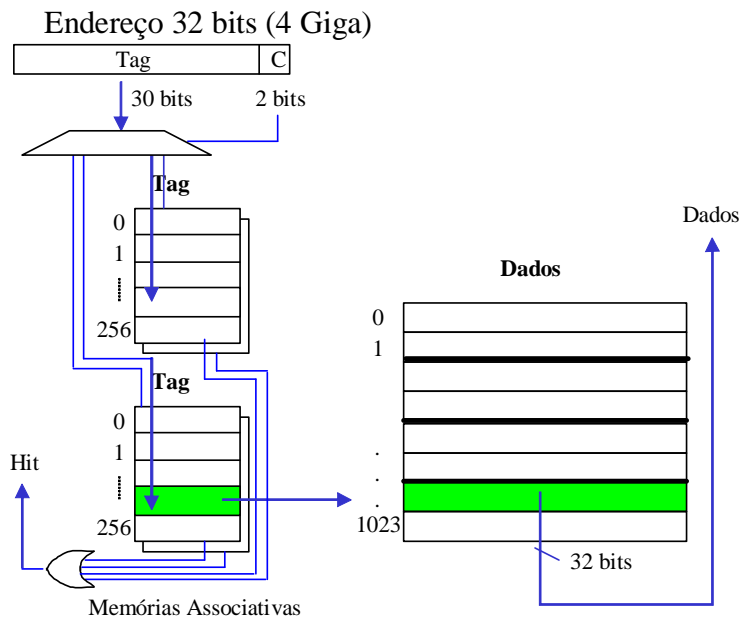
3.1.3 Mapeamento Conjunto associativo

- Compromisso entre mapeamento direto e totalmente associativo
- *Cache* é dividida em **S** conjuntos de **N** blocos/palavras
- Se **S** = 1 tenho mapeamento associativo e se **S** = número de blocos/palavras da *cache* tenho o mapeamento direto

- O endereço i da memória principal pode mapear para qualquer endereço no conjunto $(i \bmod S)$ da *cache*
 - Tenho que fazer procura dentro do conjunto
 - Preciso política de substituição (Quando tenho *miss* e busco no nível mais abaixo, caso o conjunto já esteja cheio quem tirar para abrir lugar?)
- Passos para um acesso
 1. Calculo o módulo do endereço que procuro pelo número de conjuntos S da *cache* (ou uso os bits menos significativos do endereço)
 2. Alimento a memória associativa deste conjunto com o Tag procurado
 3. Se o Tag procurado não está na memória associativa tenho *miss* (vou para 5)
 4. Senão tenho *hit* e acesso à memória *cache* com o índice fornecido pela memória associativa e efetuo a leitura (fim)
 5. Se não existir posição livre no conjunto escolho um endereço para substituir (LRU)
 6. Busco o endereço procurado no nível mais baixo e coloco em uma posição livre (ou escolhida) da *cache* cadastrando essa posição e Tag na memória associativa do conjunto e efetuo a leitura (fim)
- Divisão de bits no registrador de endereçamento
 - Exemplo de uma *cache* com 1024 linhas (2^{10}) com palavra de 32 bits e 2 conjuntos ($S=2$)



- Neste caso o espaço de endereçamento tem 4 Gigabytes (2^{32})
- Pergunta: Quanto se tem efetivamente de dados nessa *cache*? 100%
- Pergunta: Qual o tamanho das memórias associativas? **Duas MAs** de $512 * 31$ (tag) = 15872 bits / 8 = 1984 bytes / 1024 = 1,93 Kbytes
- Como ficaria a mesma *cache* com 4 conjuntos ($S=4$)?



- Pergunta: Qual o tamanho das memórias associativas? **Quatro MAs** de $256 * 30$ (tag) = 7680 bits / 8 = 960 bytes / 1024 = 0,94 Kbytes
- Pergunta: Já vimos que na realidade são transferidos blocos entre níveis. Como ficaria a divisão do endereço de uma *cache* conjunto associativa de 64 conjuntos com 2048 linhas (bloco com 16 palavras de 32 bits)

Endereço de 32 bits (4 Giga)

22 (Tag)	6 (Conjunto)	4 (Palavra)
----------	--------------	-------------

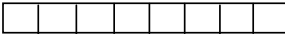
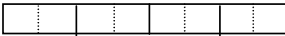
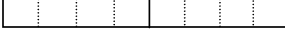
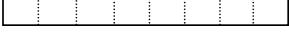
- Exercício de mapeamento – os parênteses indicam o conteúdo da posição

Cache Mapeamento Conjunto Associativo – 2 Conjuntos

MP	H M	Conteúdo da <i>cache</i>			
		Conjunto 1		Conjunto 2	
		0	1	2	3
154	M	(154)			
68	M	"	(68)		
34	M	(34)	"		
67	M	"	"	(67)	
154	M	"	(154)	"	
100	M	(100)	"	"	
67	H	"	"	(67)	
68	M	"	(68)	"	
69	M	"	"	"	(69)
70	M	(70)	"	"	"
68	H	"	(68)	"	"
34	M	(34)	(68)	(67)	(69)
	2				

- Quantidade de associatividade de uma *cache* é dada pelo número de Vias (Ways)

- Exemplo: uma *cache* de 8 posições pode ter de 1 a 8 vias (*ways*)

Vias / Ways	Mapeamento	Desenho
1	Direto	
2	Conjunto-associativo (4 conjuntos)	
4	Conjunto-associativo (2 conjuntos)	
8	Associativo	

- Exercício: Quantos conjuntos possui a *cache* L1 4-Way de 64 KBytes de um processador Ultra Sparc III (assumir bloco de 32 palavras de 64 bits)?
 - A família Ultra Sparc é utilizada nas estações da empresa SUN (A versão III possui um pipeline com 14 estágios – Superpipeline)

Resposta: tamanho do bloco = $32 * 64 \text{ bits} = 2048 \text{ bits} = 256 \text{ bytes} = 0,25 \text{ KBytes}$
 $64 \text{ Kbytes} / 0,25 \text{ Kbytes} = 256 \text{ blocos} / 4 \text{ (4-Way} = \text{quatro blocos por conjunto)}$
 $= 64 \text{ conjuntos} \rightarrow \text{cache conjunto-associativa com 64 conjuntos}$

Divisão de bits:

Assumindo um registrador de endereços de 64 bits

53 (Tag)	6 (Conjunto)	5 (Palavra)
----------	--------------	-------------

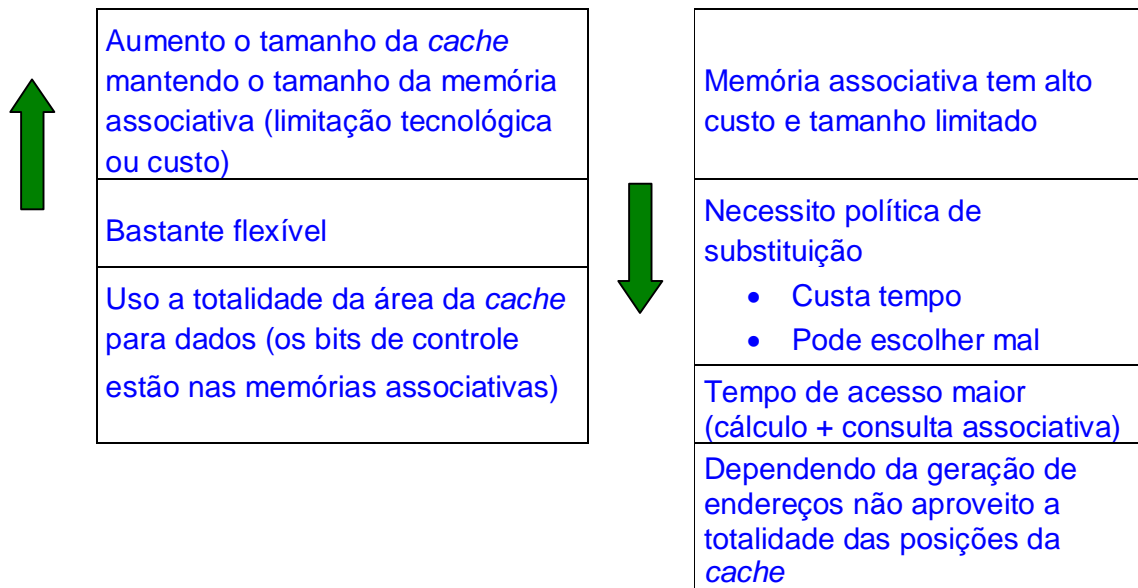
Como não existe controle dentro da cache (100% dados), cada linha da cache contém um bloco, ou seja, **a cache possui 256 linhas** (64 conjuntos, cada um com 4 blocos/linhas)!

Quantidade e tamanho das MAs utilizadas:

Para 64 conjuntos precisamos 64 MAs (uma para cada conjunto)

Como cada MA controla 4 linhas de cache (4-way) e cada linha da MA contém uma etiqueta (tag), o tamanho da linha fica: $4 * 53 \text{ bits} = 212 \text{ bits} / 8 = 26,5 \text{ bytes} = \mathbf{27 \text{ bytes}}$

- Vantagens/Desvantagens dessa técnica de mapeamento



Exercícios de dimensionamento de memórias *cache*

- 1) A área de memória disponível para implementação de uma *cache* L2 é 512 Kbytes. Considerando que a memória a ser endereçada possui 64 Mbytes (2^{26}) e a *cache* deve trabalhar com blocos de 16 palavras de 32 bits calcule para a técnica direta, totalmente associativa e conjunto associativa (4 conjuntos):
 - Divisão de bits do endereço
 - Aproveitamento efetivo da área da *cache* (relação entre dados e controle)
 - Número de linhas da *cache*
 - Quantidade e tamanho em Kbytes das memórias associativas (quando necessário)

Cache com mapeamento conjunto associativo (4 conjuntos)

- Divisão de bits do endereço:

Endereço de 26 bits (64 Mbytes)

20 (Tag)	2 (Conjunto)	4 (Palavra)
----------	--------------	-------------

- Aproveitamento efetivo:
 - 100% (só dados na *cache*, controle fica nas MAs)
- Número de linhas da *cache*:
 - Tamanho da linha?
 - Cada linha tem bloco de 16 palavras de 32 bits = $16 * 32 = 512$ bits / 8 = 64 bytes
 - Quantas linhas cabem na *cache*?
 - Cache tem 512 Kbytes = $512 * 1024 = 524288$ bytes / 64 = **8192 linhas**
- Tamanho das memórias associativas:
 - Quantas?
 - Uma para cada conjunto, ou seja 4

Tamanho de cada uma?

Cada linha da MA tem tamanho do Tag = 20 bits

O número de linhas da MA é igual ao número de linhas da *cache* que ela endereça. Como a *cache* tem 8192 e são 4 MAs, cada MA endereça $8192 / 4 = 2048$ linhas

Uma MA tem então $2048 \text{ (linhas)} * 20 \text{ (tag)} = 40960 \text{ bits} / 8 = 5120 \text{ bytes} / 1024 =$
5 Kbytes

Cache com mapeamento totalmente associativo

– Divisão de bits do endereço:

Endereço de 26 bits (64 Mbytes)

22 (Tag)	4 (Palavra)
----------	-------------

– Aproveitamento efetivo:

100% (só dados na *cache*, controle fica nas MAs)

– Número de linhas da *cache*:

Tamanho da linha?

Cada linha tem bloco de 16 palavras de 32 bits = $16 * 32 = 512 \text{ bits} / 8 = 64 \text{ bytes}$

Quantas linhas cabem na *cache*?

Cache tem 512 Kbytes = $512 * 1024 = 524288 \text{ bytes} / 64 =$ **8192 linhas**

– Tamanho das memórias associativas:

Quantas?

Uma única memória associativa

Tamanho da MA?

Cada linha da MA tem tamanho do Tag = 22 bits

O número de linhas da MA é igual ao número de linhas da *cache* que ela endereça. Como a *cache* tem 8192 a MA endereça 8192 linhas

Uma MA tem então $8192 \text{ (linhas)} * 22 \text{ (tag)} = 180224 \text{ bits} / 8 = 22528 \text{ bytes} / 1024 =$
22 Kbytes

Cache com mapeamento direto

– Número de linhas da *cache*:

Tamanho da linha?

Conteúdo da linha

1 (Validade)	? (Tag)	512 (Bloco)
--------------	---------	-------------

Cada linha tem um bit de validade, os bits de Tag e bloco de 16 palavras de 32 bits

O problema é o tamanho do Tag, pois como ele depende do número de linhas da *cache*, que é o que estou procurando, não conseguimos calcular ...

Para resolver esta questão, utilizamos uma aproximação inicial e depois realizamos um ajuste. Sabemos que esta cache terá controle dentro da linha (bit de validade e etiqueta) além do bloco. Para a aproximação inicial, no entanto, usamos apenas o tamanho do bloco (como nas duas técnicas anteriores) o que resultará em mais linhas do que a resposta correta.

Cada linha tem bloco de 16 palavras de 32 bits = $16 * 32 = 512$ bits / 8 = 64 bytes
 Quantas linhas cabem na *cache*?
 Cache tem 512 Kbytes = $512 * 1024 = 524288$ bytes / 64 = **8192 linhas**

Para este número de linhas precisamos de 13 bits no endereçamento. Agora precisamos calcular a etiqueta real e fazer o ajuste no número de linhas.

com **13 bits** para linha (podendo endereçar **8192** linhas da *cache*)
 Tamanho da linha = 1+9 (Tag)+512 = 522 bits / 8 = 65,25 bytes
 Quantas linhas cabem na *cache*?
 Cache tem 512 Kbytes = $512 * 1024 = 524288$ bytes / 66 = **7943,75** linhas
 Cache tem **7944 linhas** e Tag = 26 (endereço) – 4 (palavra) – 13 (linha) = **9**

– Divisão de bits do endereço:

Endereço de 26 bits (64 Mbytes)

9 (Tag)	13 (Linha)	4 (Palavra)
---------	------------	-------------

– Aproveitamento efetivo:

Dados em cada linha: um bloco de 16 palavras de 32 bits = 512 bits
 Tamanho total da linha: 1 (validade) + 9 (Tag) + 512 (bloco) = 522 bits
 Percentual de aproveitamento: $512 \rightarrow 100\%$
 $512 \rightarrow ?\%$ (uso para dados)

Aproveitamento efetivo = $512 * 100 / 522 = \mathbf{98,08\%}$

– Tamanho das memórias associativas:

Não utiliza MAs

- 2) A área de memória disponível para implementação de uma *cache* L2 é 256 Kbytes. Considerando que a memória a ser endereçada possui 256 Mbytes (2^{28}) e a *cache* deve trabalhar com blocos de 8 palavras de 16 bits calcule para a técnica direta, totalmente associativa e conjunto associativa (16 conjuntos):

- Divisão de bits do endereço
- Aproveitamento efetivo da área da *cache* (relação entre dados e controle)
- Número de linhas da *cache*
- Quantidade e tamanho em Kbytes das memórias associativas (quando necessário)

Cache com mapeamento conjunto associativo (16 conjuntos)

– Divisão de bits do endereço:

Endereço de 28 bits (256 Mbytes)

21 (Tag)	4 (Conjunto)	3 (Palavra)
----------	--------------	-------------

– Aproveitamento efetivo:

100% (só dados na *cache*, controle fica nas MAs)

– Número de linhas da *cache*:

Tamanho da linha?

Cada linha tem bloco de 8 palavras de 16 bits = $8 * 16 = 128$ bits / 8 = 16 bytes

Quantas linhas cabem na *cache*?

Cache tem 256 Kbytes = $256 * 1024 = 262144$ bytes / 16 = **16384 linhas**

– Tamanho das memórias associativas:

Quantas?

Uma para cada conjunto, ou seja 16

Tamanho de cada uma?

Cada linha da MA tem tamanho do Tag = 21 bits

O número de linhas de cada MA nessa técnica é igual ao número de linhas do conjunto que ela endereça. Como a *cache* tem 16384 linhas e são 16 MAs, cada MA endereça $16384 / 16 = 1024$ linhas

Uma MA tem então 1024 (linhas) * 21 (tag) = 21504 bits / 8 = 2688 bytes / 1024 = **2.625 Kbytes**

Cache com mapeamento totalmente associativo

– Divisão de bits do endereço:

Endereço de 28 bits (256 Mbytes)

25 (Tag)	3 (Palavra)
----------	-------------

– Aproveitamento efetivo:

100% (só dados na *cache*, controle fica nas MAs)

– Número de linhas da *cache*:

Tamanho da linha?

Cada linha tem bloco de 8 palavras de 16 bits = $8 * 16 = 128$ bits / 8 = 16 bytes

Quantas linhas cabem na *cache*?

Cache tem 256 Kbytes = $256 * 1024 = 262144$ bytes / 16 = **16384 linhas**

– Tamanho das memórias associativas:

Quantas?

Uma única memória associativa

Tamanho da MA?

Cada linha da MA tem tamanho do Tag = 25 bits

O número de linhas da MA nessa técnica é igual ao número de linhas da *cache* que ela endereça. Como a *cache* tem 16384 a MA endereça 16384 linhas

Uma MA tem então 16384 (linhas) * 25 (tag) = 409600 bits / 8 = 51200 bytes / 1024 = **50 Kbytes**

Cache com mapeamento direto

– Número de linhas da *cache*:

Tamanho da linha?

Conteúdo da linha

1 (Validade)	? (Tag)	128 (Bloco)
--------------	---------	-------------

Cada linha tem um bit de validade, os bits de Tag e bloco de 8 palavras de 16 bits

O problema é o tamanho do Tag, pois como ele depende do número de linhas da *cache* que é o que estou procurando, não conseguimos calcular ...

Para resolver esta questão, utilizamos uma aproximação inicial e depois realizamos um ajuste. Sabemos que esta cache terá controle dentro da linha (bit de validade e etiqueta) além do bloco. Para a aproximação inicial, no entanto, usamos apenas o tamanho do bloco (como nas duas técnicas anteriores) o que resultará em mais linhas do que a resposta correta.

Cada linha tem bloco de 8 palavras de 16 bits = $8 * 16 = 128$ bits / $8 = 16$ bytes

Quantas linhas cabem na *cache*?

Cache tem 256 Kbytes = $256 * 1024 = 262144$ bytes / $16 = \mathbf{16384}$ linhas

Para este número de linhas precisamos de 14 bits no endereçamento. Agora precisamos calcular a etiqueta real e fazer o ajuste no número de linhas.

com **14 bits** para linha (podendo endereçar **16384** linhas da *cache*)

– Divisão de bits do endereço:

Endereço de 28 bits (256 Mbytes)

11 (Tag)	14 (Linha)	3 (Palavra)
----------	------------	-------------

Tamanho da linha = $1 + 11$ (Tag) + $128 = 140$ bits / $8 = 17.5$ bytes

Quantas linhas cabem na *cache*?

Cache tem 256 Kbytes = $256 * 1024 = 262144$ bytes / $17.5 = \mathbf{14979}$ linhas

Cache tem **14979 linhas** e Tag = 28 (endereço) – 3 (palavra) – 14 (linha) = **11**

– Divisão de bits do endereço:

Endereço de 28 bits (256 Mbytes)

11 (Tag)	14 (Linha)	3 (Palavra)
----------	------------	-------------

– Aproveitamento efetivo:

Dados em cada linha: um bloco de 8 palavras de 16 bits = 128 bits

Tamanho total da linha: 1 (validade) + 11 (Tag) + 128 (bloco) = 140 bits

Percentual de aproveitamento: $140 \rightarrow 100\%$

$128 \rightarrow ?\%$ (uso para dados)

Aproveitamento efetivo = $128 * 100 / 140 = \mathbf{91.42\%}$

– Tamanho das memórias associativas:

Não utiliza MAs nessa técnica

3.2 Integridade dos dados na *cache*

- Problema: ocorreu um *miss* e o endereço desejado foi buscado no nível inferior da hierarquia de memória. Só que a *cache* está cheia e não há lugar para escrever este dado. O algoritmo de substituição é acionado e uma posição é escolhida. Só que estes dados foram alterados e não podem ser simplesmente descartados.
- Este problema ocorre porque uma escrita foi efetuada apenas no nível da *cache* e as cópias deste dado nos outros níveis não estão atualizadas

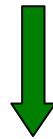
- Perguntas:
 - Como saber que os dados foram alterados?
 - Como salvar essas alterações?
 - Em que momento salvar as informações?
- Existem duas técnicas para manter a Integridade dos dados
 - Write-through
 - Write-back

3.2.1 Write-through (escrevo através)

- Técnica mais antiga
- Escrevo as alterações em todos os níveis (escrevo através)
- Quando? Sempre que escrevo
- Quanto? Somente a palavra alterada
- Vantagens / Desvantagens dessa técnica



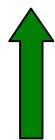
Dado sempre atual em todos os níveis
Escreve menos dado



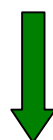
Escreve mais vezes
Precisa mais do barramento

3.2.2 Write-back (escrevo de volta)

- Técnica mais recente
- Escrevo alterações só quando substituo
- Quando: substituição
- Quanto: unidade do nível (bloco no caso da *cache*)
- Como sei quem foi alterado: *dirty-bit* (bit de sujeira) no bloco
- Vantagens / Desvantagens dessa técnica



Escreve menos vezes
Precisa menos do barramento



Escreve mais dados de cada vez
Aumento tempo de substituição

3.3 Passos para escrita e leitura na *cache*

- Para solucionar o problema da integridade dos dados as técnicas acima são incorporadas nas operações de leitura e escrita

3.3.1 Leitura

1. Verifico se foi *hit*, se não foi vou para 3
2. Procuro por bloco desejado (Tag ou direto), leio e repasso ao processador. Vou para 8
3. Requisito ao nível mais baixo
4. Recebo bloco, procuro onde colocar e se *cache* cheia vou para 5. Se acho posição livre escrevo bloco, leio a palavra desejada no bloco, repasso dado ao processador e vou para 8
5. Procuro bloco para substituir (uso política)
6. Se **Write-back** e *dirty-bit* ligado, salvo bloco a ser substituído no nível mais baixo
7. Substituo bloco, leio a palavra desejada no bloco e repasso dado ao processador
8. Pronto

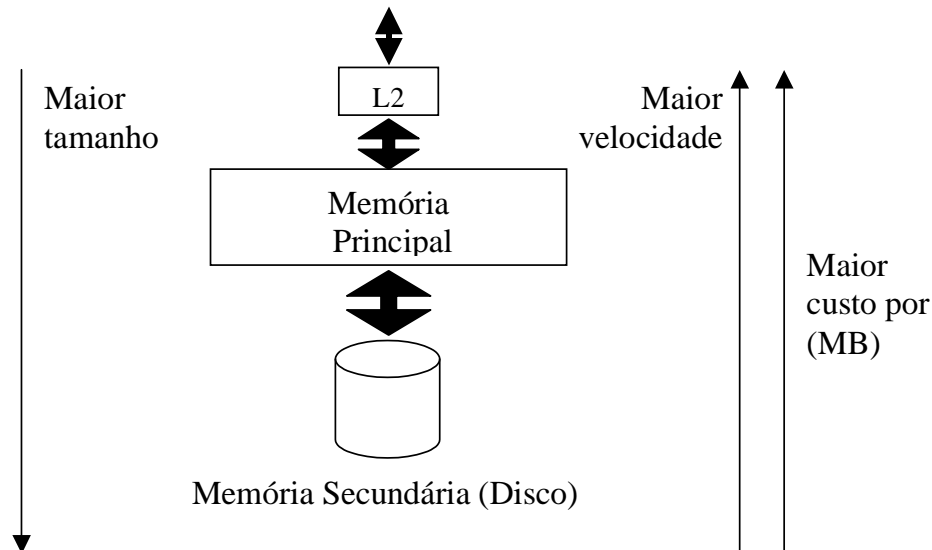
3.3.2 Escrita

1. Verifico se foi *hit*, se não foi vou para 3
 2. Procuro por bloco desejado (Tag ou direto) e escrevo. Se **Write-through** escrevo palavra também nos níveis mais baixos. Se **Write-back** ligo *dirty-bit*. Vou para 8
 3. Requisito ao nível mais baixo
 4. Recebo bloco, procuro onde colocar e se *cache* cheia vou para 5. Se acho posição livre escrevo bloco e efetuo escrita da palavra. Se replicação **Write-through** escrevo palavra também nos níveis mais baixos. Se **Write-back** ligo *dirty-bit*. Vou para 8
 5. Procuro bloco para substituir (uso política)
 6. Se replicação **Write-back** e *dirty-bit* ligado, salvo substituído no nível mais baixo
 7. Substituo bloco e escrevo palavra. Se replicação **Write-through** escrevo palavra também nos níveis mais baixos. Se **Write-back** ligo *dirty-bit*.
 8. Pronto
- Pergunta: Para que buscar dados do nível mais baixo quando tenho *miss* na operação de escrita se vou escrever os dados novamente? Para que buscar para escrever por cima ? Porque só escrevo uma palavra e estou buscando o bloco.

4 GERÊNCIA DA MEMÓRIA PRINCIPAL

- Referência: Stallings, Capítulos 7 e 8
- Em um sistema monoprogramado a memória principal é dividida em duas partes: uma para o sistema operacional (monitor residente, núcleo) e outra para o programa que está sendo executado
- Em um sistema multiprogramado a memória “de usuário” ainda tem que ser dividida entre vários processos

- Essa divisão é feita pelo sistema operacional de forma dinâmica e é chamada de **Gerencia de Memória**
- Uma gerência de memória eficiente é vital em um sistema multiprogramado. Se somente poucos processos couberem na memória o processador ficará parado grande parte do tempo esperando por operações de E/S. Sendo assim, uma técnica de gerência que consiga colocar mais processos na memória melhora a taxa de utilização do processador e consequentemente o desempenho da máquina como um todo
- A memória principal pode ser vista como mais um nível da hierarquia de memória de forma que o princípio da gerência é o mesmo dos outros níveis: os dados mais usados são trazidos para a memória para diminuir o tempo médio de acesso ao nível mais baixo, neste caso o disco



- Teoricamente poderiam ser aplicadas na gerência de memória principal as mesmas técnicas que foram vistas para a gerência de *caches*, mas não são por dois motivos:
 - **Evolução Histórica:** a idéia de uma memória como área de armazenamento temporário de dados já existe há muito tempo, muito antes de se pensar em uma hierarquia de memória e em *caches*
 - **Diferentes Características:** a memória principal é normalmente muito maior que as memórias *caches* e os tempos de acesso são muitas vezes maiores. Ao contrário das *caches* parte da gerência pode ser feita em software e as unidades de gerência possuem a identificação do seu dono (processo), o que pode ser utilizado na estratégia de gerência
- Estes dois motivos fizeram com que a gerência de memória tenha se desenvolvido de forma um pouco diferente do que a gerência de *caches*

4.1 Histórico da gerência de memória

- Inicialmente a memória era empregada como área temporária para acelerar o acesso aos dados
- Como era um recurso caro na época as o seu tamanho era bastante reduzido (poucos Kbytes)

- Em consequência disto muitas vezes um programa não cabia inteiramente na memória juntamente com o seu ambiente de execução (interpretador, bibliotecas, etc.)
- A gerência de memória foi introduzida em sistemas monoprogramados para permitir que um programa maior que a memória pudesse executar na máquina
- As primeiras estratégias eram baseadas em *overlays* (sobreposição) e de responsabilidade total do programador
 - O programador fazia a divisão do seu programa em partes que podiam executar de forma autônoma na memória (*overlay*)
 - No final de cada uma dessas partes era colocado o código responsável pela carga da próxima parte que poderia ser sobreposta à anterior já que os endereços antigos não se faziam mais necessários
 - O programador tinha controle total da memória da máquina e era responsável pela troca das partes
 - O programa tinha que ser escrito de forma que não necessitasse ser quebrado em muitas partes já que essa troca tinha um custo bastante alto
- Com o advento da multiprogramação surgiram algumas dificuldades
 - A gerência de overlays tinha que possibilitar agora que vários programas que somados não coubessem na memória pudessem na máquina executar de forma concorrente
 - Para que a gerência de overlays de cada programa não interferisse na dos outros programas era necessário que um agente externo fosse responsável pela gerência
 - A responsabilidade de gerenciar a memória passou então do usuário para o monitor residente, mas tarde chamado de sistema operacional

4.2 O problema da fragmentação de memória

- Um problema diretamente relacionado com a gerência de memória é sua fragmentação
- Fragmentação resulta em desperdício de memória
- Existem dois tipos de fragmentação
 - Fragmentação Interna: ocorre quando é usada uma unidade de gerência de **tamanho fixo** (Ex: página). Requisições dos usuários que não sejam exatamente divisíveis por essa unidade tem que ser arredondadas para cima resultando em uma unidade alocada, mas não completamente ocupada
 - Fragmentação Externa: ocorre quando é usada uma unidade de gerência de **tamanho variável** (Ex: segmento). Uma sequência de alocações e liberações desse tipo podem gerar várias lacunas de tamanho variável na memória. Uma requisição de usuário pode então ser negada apesar de existir memória livre suficiente para atendê-la (por causa da fragmentação não foi possível encontrar uma unidade contígua)

4.3 Endereçamento da memória principal

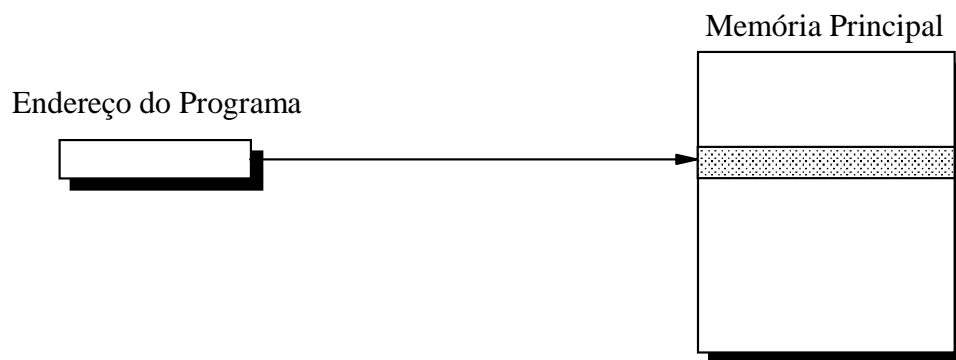
- Problema: como os endereços utilizados em um programa são usados para acessar posições da memória principal

4.3.1 Endereçamento contíguo

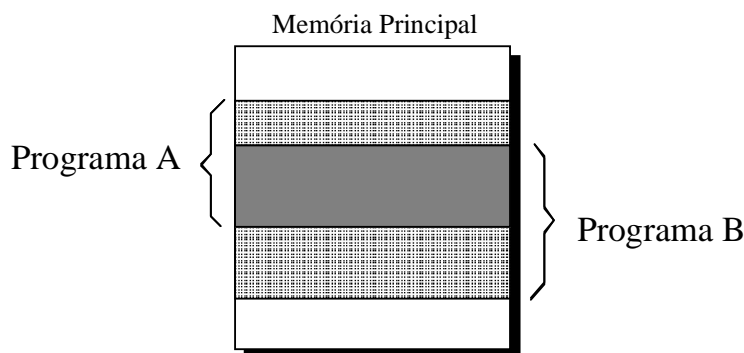
- O programa é carregado inteiro em uma única área de memória contígua
- Posso ter duas formas de endereçamento no caso da gerência de áreas de memória **contíguas**:
 - direto
 - relativo

Endereçamento Direto

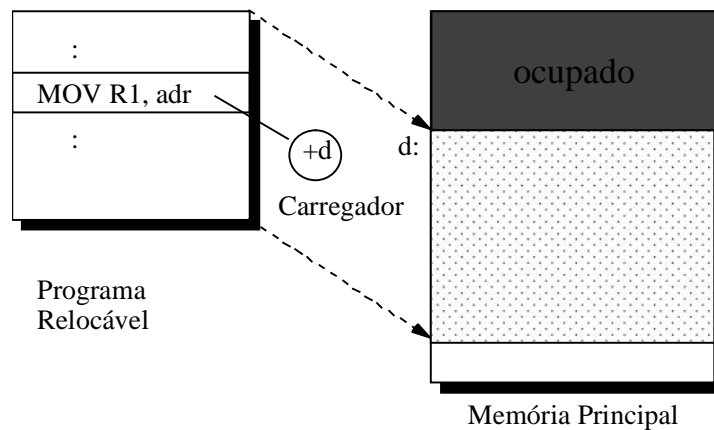
- Os endereços de um programa executável são usados diretamente no acesso à memória principal
- Estes endereços são definidos **durante a compilação/lincagem**
- A posição do programa na memória está assim definida e não pode ser alterada



- No caso de multiprogramação este tipo de endereçamento pode gerar conflitos no acesso à memória já que a posição do programa na memória foi definida sem conhecimento do que estava alocado



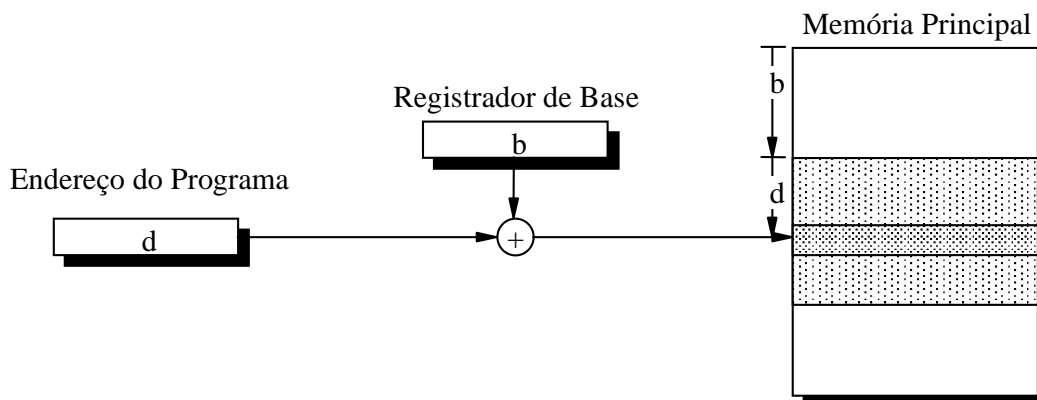
- Programas só podem ser carregados na memória se seus espaços de endereçamento são disjuntos
- Uma alternativa é a definição dos endereços somente **durante a carga** do programa
- O espaço de endereçamento de um programa inicia sempre em 0 e podendo ser facilmente relocados na carga pelo carregador



- A conversão dos endereços pode atrasar consideravelmente a operação de carga
- O programa não pode ser mudado de lugar durante a sua execução (compactação, *swap*)

Endereçamento Relativo

- Uma alternativa bem mais flexível é a composição do endereço somente **na hora do acesso**
- O programa é carregado para a memória com seus endereços relativos (espaço de endereçamento lógico inicia em 0)
- O processador possui um registrador de endereçamento (registrador de base) que contém a base do espaço de endereçamento físico para cada programa
- Essa base é somada ao endereço relativo do programa em cada acesso à memória



- O custo da conversão é pago em cada acesso à memória
- Nesse caso o programa pode ser trocado de lugar na memória em tempo de execução (compactação, *swap*)

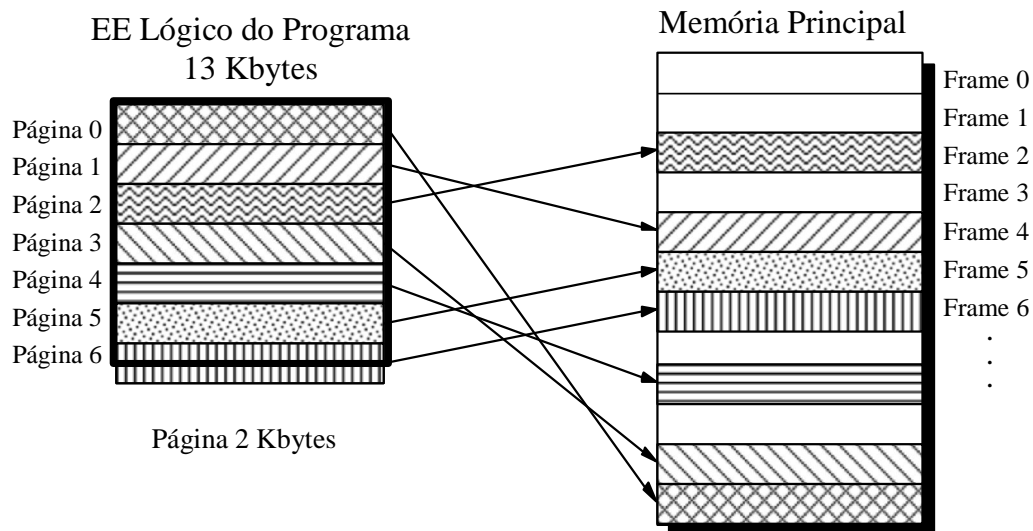
4.3.2 Endereçamento não-contíguo

- A quebra do programa em pedaços que são carregados em áreas distintas de memória sem a necessidade de respeitar qualquer ordem resulta em:
 - Um melhor aproveitamento da memória (menor fragmentação externa pelo aproveitamento de lacunas)
 - A gerência de memória fica mais trabalhosa
- A conversão de endereços é dita **dinâmica**

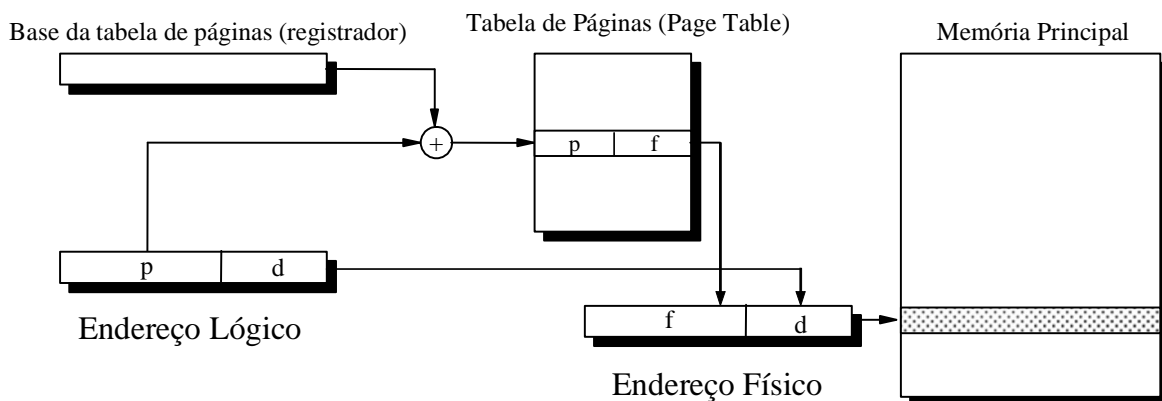
- Os endereços do programa (endereços lógicos) são convertidos na hora do acesso em endereços físicos (normalmente feito em hardware para acelerar o procedimento, Ex. MMU – *Memory Manager Unit*)
- Posso ter três formas de endereçamento no caso da gerência de áreas de memória **não-contíguas**:
 - **Paginação**
 - **Segmentação**
 - **Segmento-paginação**

Endereçamento Paginado

- Memória física quebrada em **frames** (moldura) com um tamanho fixo de 2k, 4k, ou 8k
- Espaço de endereçamento lógico do programa quebrado em **páginas** que possuem o mesmo tamanho dos frames
- Quando o processo é executado todas as suas páginas são carregadas para frames livres da memória (qualquer página em qualquer frame livre)
- Unidade de gerência de memória de tamanho **fixo** (página-frame)



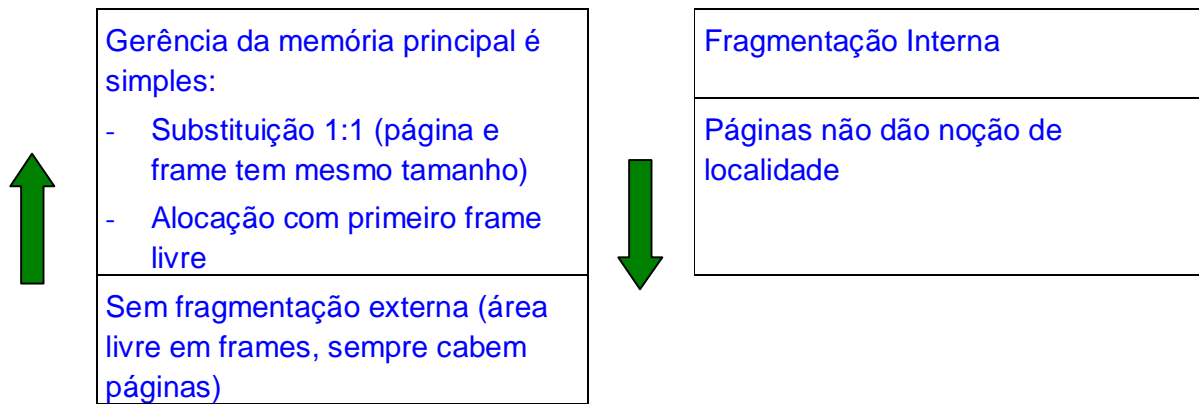
- **Conversão de endereços**



- Número de bits usados para **d** determina o tamanho das páginas (e frames)

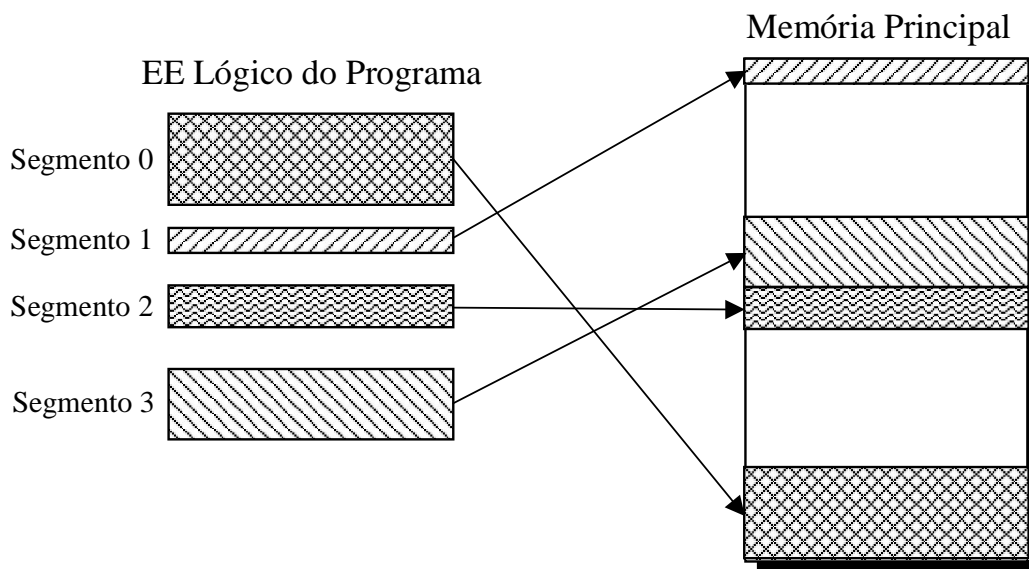
- Número de bits usados para **p** determina o número máximo de páginas de um processo
- Número de bits de **f** determina o número total de frames da memória principal e é dado pela divisão do tamanho da memória principal pelo tamanho da página (Ex: MP 1MByte (2^{20}), **p** 10 bits (2^{10} – página de 1kbyte) resulta em 10 bits para **f** (2^{10} – 1024 frames na MP))
- Unidade de gerência de tamanho fixo não gera fragmentação externa e gera fragmentação interna
- A tabela de páginas (*page table*) é usada para a conversão de páginas em frames (uma tabela para cada processo)
- Tabela de páginas pode ser armazenada em:
 - Registradores: rápido, mas limita tamanho da tabela
 - Memória principal (área do sistema): lento, pois são necessários dois acessos, mas tabela de páginas pode ter tamanho ilimitado (porém alguns sistemas limitam a tabela pelo tamanho da página)
 - TLB (*Translation Lookaside Buffer*): área de memória associativa usada como *cache* para as conversões mais efetuadas (ver 4.2.3)
- Tamanho da página
 - Pequena: menor fragmentação interna, tabela de páginas fica maior
 - Grande: maior fragmentação interna, tabela de páginas fica menor
 - Deve ser considerada também a operação de transferência para o nível mais baixo, o disco, cuja unidade de transferência é o setor com normalmente 512 bytes
- Tabela de Frames (*Frame Table*) usada para controle de quais frames se encontram livres ou não estão sendo mais usados (para alocação e substituição de páginas)
 - Uma tabela de frames para todo o sistema
 - Seu tamanho é fixo, já que o número de frames da memória principal é conhecido (tamanho da memória / tamanho da página)
 - Procura por first-“found” (qualquer lacuna serve para qualquer página)
 - Possui campos adicionais para controle da política de troca de páginas (LRU, LFU)
- Quando os processos morrem suas páginas na tabela de frames são marcadas como livres
- É possível o compartilhamento de frames entre vários processos (leitura)
 - Fácil de implementar com várias tabelas de página apontando para o mesmo frame
 - Controle adicional para verificação de quando frame compartilhado pode ser desalocado

- Vantagens/Desvantagens dessa técnica

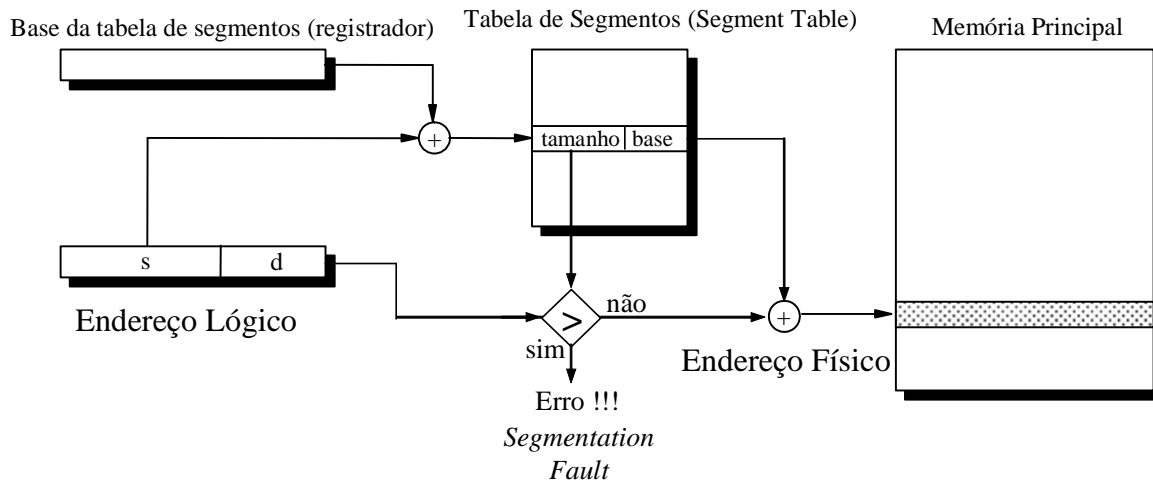


Endereçamento Segmentado

- Não divido a memória física, posso alocar unidade em qualquer posição
- Unidade de gerência tamanho **variável** denominada **segmento**
- Segmento é definido pelo usuário ou pelo compilador e resulta em uma maior localidade
 - Endereços de um mesmo segmento estão relacionados
 - Gerência de memória mais preocupada com a visão do usuário
 - Exemplo de segmentos: dados, código, pilha
- Quando o processo é executado todos os seus segmentos são carregados para a memória (em qualquer posição livre)

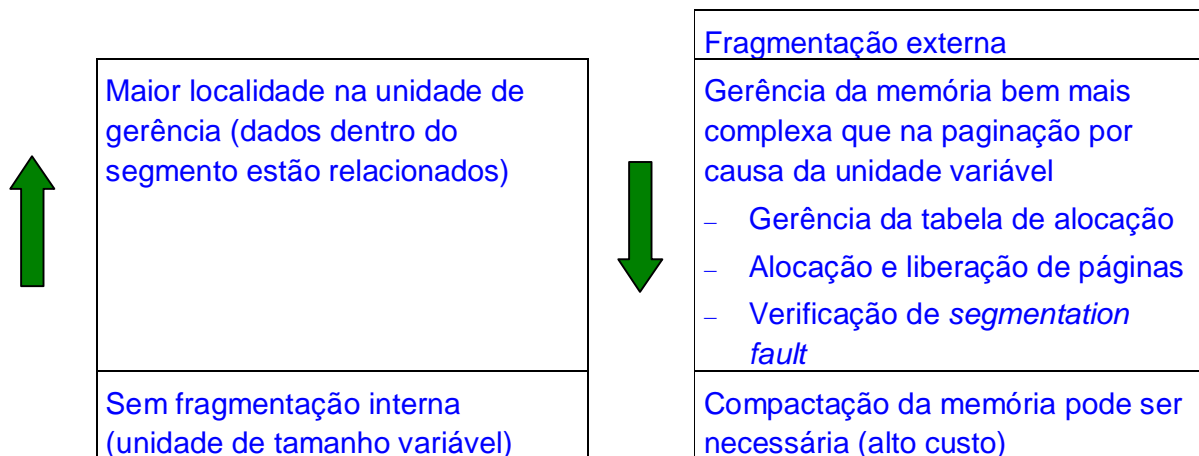


- Conversão de endereços



- **O deslocamento fornecido pelo usuário precisa ser comparado com o tamanho do segmento para que não possam ocorrer invasões de segmentos vizinhos (este teste não era necessário na paginação pela limitação do número de bits)**
- Número de bits usados para **d** determina o tamanho máximo dos segmentos
- Número de bits usados para **s** determina o número máximo de segmentos que um processo pode ter
- Unidade de gerência de tamanho variável não gera fragmentação interna e gera fragmentação externa
- Procedimento de compactação da memória pode vir a ser necessário por causa da alta fragmentação externa
 - Segmentos são agrupados novamente eliminando lacunas (migração de processos – alto custo)
- Tabela de Segmentos (*Segment Table*) usada para encontrar a base de um segmento na memória e verificar se o deslocamento requisitado pelo usuário se encontra dentro do limite máximo do segmento (seu tamanho)
- Tabela de segmentos pode ser armazenada em (igual a paginação):
 - Registradores: rápido, mas limita tamanho da tabela
 - Memória principal (área do sistema): lento, pois são necessários dois acessos, mas tabela de segmentos pode ter tamanho ilimitado
 - TLB (*Translation Lookaside Buffer*): área de memória associativa usada como *cache* para as conversões de endereço mais efetuadas (ver 4.2.3)
- Tabela de Alocação (*Allocation Table*) usada para controle de quais lacunas se encontram livres (para alocação e substituição de páginas)
 - Uma tabela de alocação para todo o sistema
 - Seu tamanho é variável, já que o número de áreas da memória principal é inicialmente 1 (toda a memória) e ao longo da gerência pode variar de acordo com a inclusão de áreas ocupadas (uma área ocupada no meio da memória faria a tabela ficar com 3 entradas)
 - Procura por first-fit, next-fit, best-fit, worst-fit, etc.

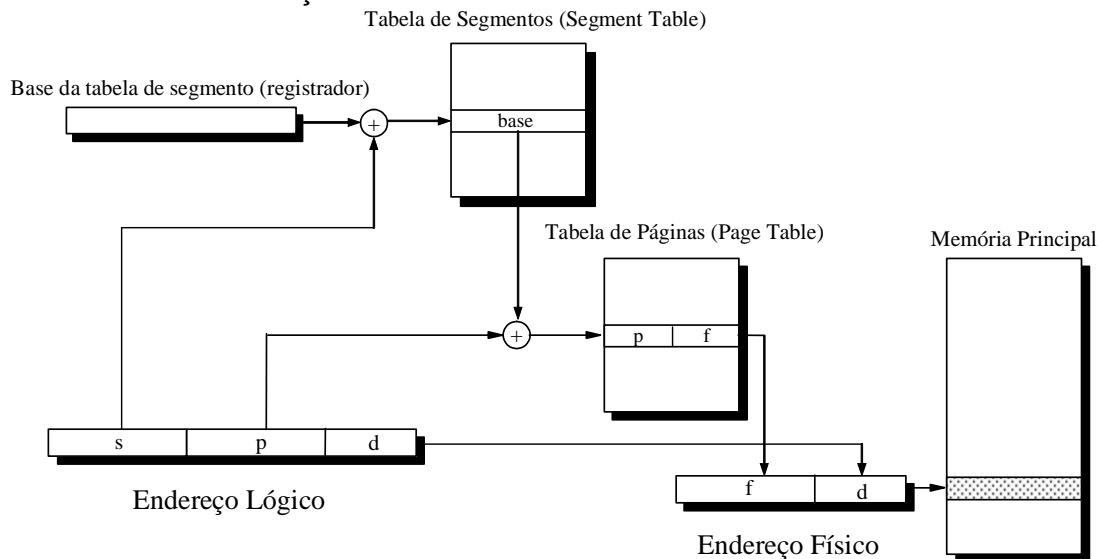
- Liberação de segmentos pode resultar na junção de vários segmentos e na diminuição do número de entradas da tabela
- Possui campos adicionais para controle da política de troca de páginas (LRU, LFU)
- É possível o compartilhamento de segmentos entre vários processos (leitura)
 - Fácil de implementar com várias tabelas de segmento apontando para a mesma área de memória
 - Controle adicional para verificação de quando segmento compartilhado pode ser desalocado
- Vantagens/Desvantagens dessa técnica



Endereçamento Segmento-Paginado

- Combinação entre segmentação e paginação
- Aproveito a visão lógica de memória mais próxima do usuário da segmentação e a gerência de memória mais simples da paginação
- Usuário aloca segmentos e estes são mapeados em um grupo de páginas
- Memória física quebrada em **frames** (moldura) com um tamanho fixo de 2k, 4k, ou 8k como na paginação, porém existe a informação de que páginas compõem cada segmento
- Unidade de gerência de memória é na realidade de tamanho **fixo** (página-frame)

- Conversão de endereços

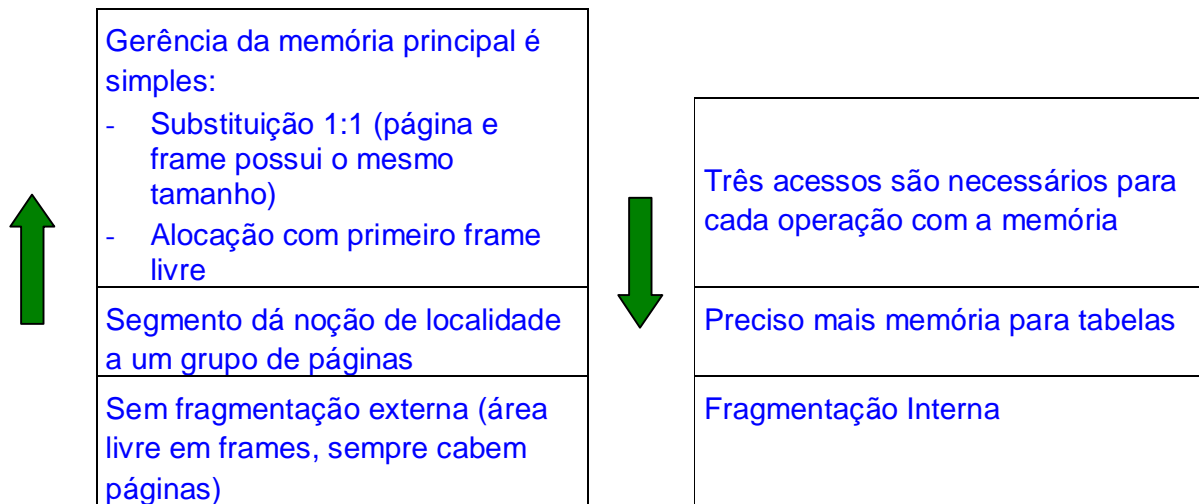


- Sequência de um acesso

1. Com o número do segmento descubro na tabela de segmentos a base da tabela de páginas deste segmento (usando o registrador como base)
2. Com o número da página e a base da tabela de páginas descubro o frame correspondente
3. Com o número do frame e o deslocamento dentro dele acesso a memória

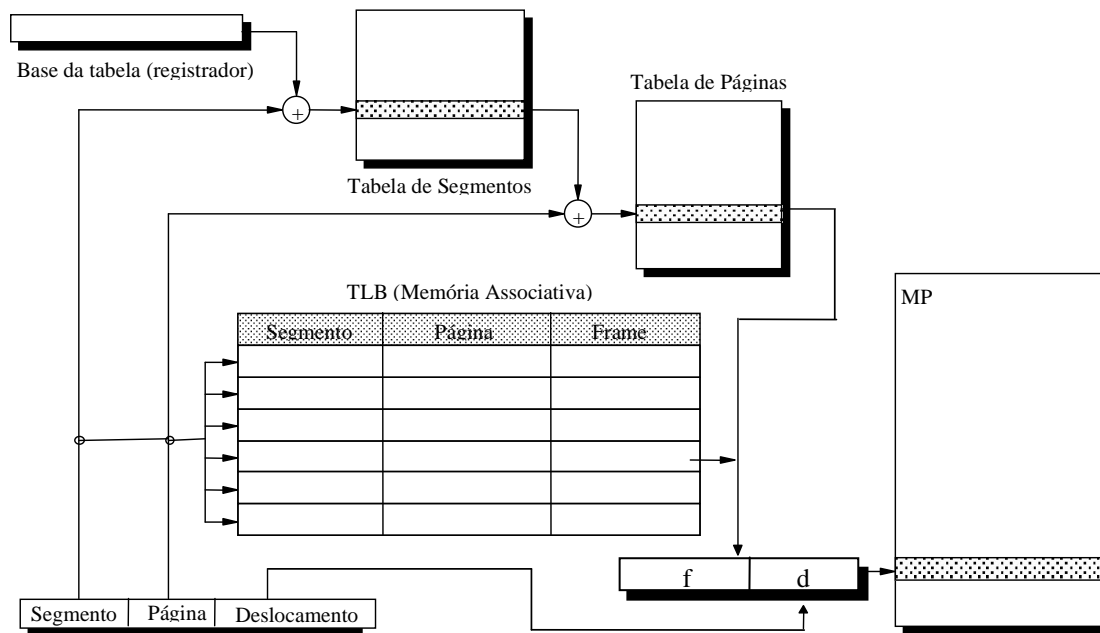
- Unidade de gerência de tamanho fixo não gera fragmentação externa e gera fragmentação interna como na paginação
- Tabela de Segmentos (*Segment Table*) usada para a obtenção da base da tabela de páginas do segmento desejado – uma para cada processo
- A tabela de páginas (*page table*) é usada para a conversão das páginas de cada segmento em frames (uma tabela para cada segmento de cada processo)
- Tabela de Frames (*Frame Table*) usada para controle de quais frames se encontram livres ou não estão sendo mais usados (para alocação e substituição de páginas) como na paginação
 - Uma tabela de frames para todo o sistema
 - Seu tamanho é fixo, já que o número de frames da memória principal é conhecido (tamanho da memória / tamanho da página)
 - Procura por first-“found” (qualquer lacuna serve para qualquer página)
 - Possui campos adicionais para controle da política de troca de páginas (LRU, LFU)
- Quando os processos morrem suas páginas na tabela de frames são marcadas como livres

- Vantagens/Desvantagens dessa técnica



4.3.3 Otimização do endereçamento não-contíguo (TLB)

- Problema: tabelas de páginas e de segmento acabam ficando tão grandes que precisam ser armazenadas na memória
- Dessa forma para cada acesso à memória se faz necessário no mínimo um outro acesso (na segmento paginada até dois acessos) para a obtenção das tabelas reduzindo consideravelmente a velocidade de acesso à memória
- Para acelerar a conversão é usada uma área de memória associativa adicional chamada TLB – *Translation Lookaside Buffer*
- A TLB funciona como uma *cache* guardando as conversões mais usadas
- Como é uma memória associativa, a procura é feita em paralelo em todas as suas posições
- Características
 - Tamanho da linha 4-8 bytes (pode conter segmento, página e frame)
 - Número de linhas 24-1024
Ex: TLB de um Pentium III possui 32 linhas
 - Hit-time 1 ciclo, Miss-Penalty 10-30 ciclos
 - Hit-ratio aproximadamente 99%
Como pode ser tão alto? Devido à localidade dos acessos!!!
- Exemplo de utilização (segmento paginação)

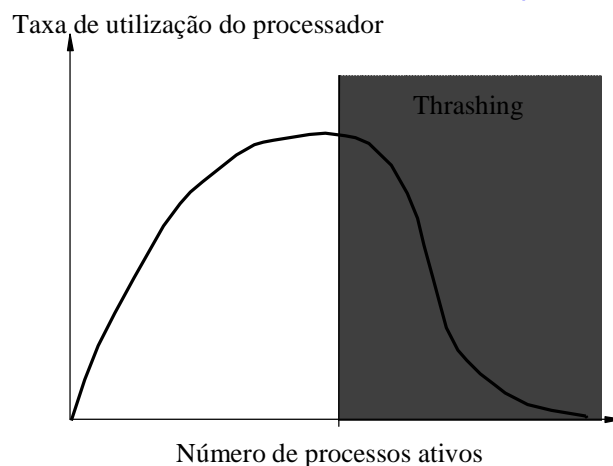


- A TLB é consultada antes do acesso as tabelas e se ocorrer um hit retorna o frame desejado
- Ocorrendo um miss a consulta as tabelas é feita normalmente e o frame obtido é colocado na TLB no lugar da conversão menos recentemente utilizada (LRU)

4.4 Memória Virtual

- Na gerência de memória convencional as unidades (páginas ou segmentos) de um programa eram todas carregadas para memória principal antes de sua execução
- Como vimos anteriormente, devido às regras da localidade, um programa só precisa de algumas dessas unidades em um determinado momento
- Dessa forma é possível gerenciar a memória de forma que só as unidades que são necessárias em um determinado momento se encontrem na memória principal
- A memória física pode ser melhor aproveitada dessa forma sendo possível:
 - A execução de um programa maior que a memória
 - A execução de vários programas “ao mesmo tempo” que somados são maiores que a memória
- O processador gera endereços para um espaço de endereçamento lógico bem maior do que o tamanho da memória física (daí o nome **virtual** já que essa quantidade de memória não existe fisicamente na memória principal) e o SO aplica as seguintes regras:
 - Quando um processo inicia suas unidades não são todas carregadas para memória
 - Nas tabelas de conversão de endereços é indicado que as unidades não estão na memória (Ex: bit de validade)
 - Quando uma conversão de endereço se faz necessária, o sistema gera um *page-fault* e manda buscar a página do disco (**são carregadas por demanda**). O processo perde o processador
 - O processo só vai para a fila de pronto quando a unidade estiver disponível na memória

- Se a memória ficar cheia, novas unidades são colocadas no lugar das menos recentemente usadas (LRU)
- Atualmente são comuns EE (espaço de endereçamento) lógicos de 4 Gigabytes (2^{32}) por causa dos registradores de endereçamento de 32 bits. Os novos processadores de 64 bits possibilitarão EE lógicos maiores
- Essa técnica tem os seguintes custos associados:
 - *Miss-penalty* é alto, pois a unidade tem que ser buscada do disco
 - Aumenta o número de trocas de contexto por causa de *page-faults* e de eventos a serem tratados (evento gerado quando a unidade foi trazida) o que gera um custo para o SO
- Um problema sério que pode ocorrer na gerência de memória virtual é **Thrashing**
 - Thrashing vem da palavra trash que significa lixo e indica que nada de produtivo está sendo feito
 - Thrashing ocorre quando o sistema fica a maior parte do tempo trocando páginas e o processador não consegue executar nenhum processo
 - Isso resulta do fato de muitos processos estarem ativos e a memória física ser muito pequena para acomodar suas unidades
 - Sendo assim, quando um processo ganha a CPU ele manda trazer suas unidades e volta a dormir, quando essas unidades são trazidas elas apagam as unidades de um outro processo por causa da pouca memória. Quando for a vez desse outro, suas páginas já não estão mais na memória, e assim por diante ...



- O gráfico acima mostra que existe um número de processos ideal para que a multiprogramação obtenha a maior taxa de utilização do processador
- Acima deste número o desempenho da máquina começa a diminuir. Este número é bastante dinâmico e depende:
 - Da arquitetura da máquina
 - Do tamanho da memória principal
 - Do número de processos ativos
 - Do tipo de processos que estão ativos (io-bound, cpu-bound)
- Este efeito pode ser observado em PCs quando muitos processos estão ativos. A luz do disco pisca intensamente e pouca coisa acontece. A solução é comprar mais

memória ou executar menos programas ao mesmo tempo (ou tornar a gerência de memória mais eficiente)

4.5 Estudo de Casos

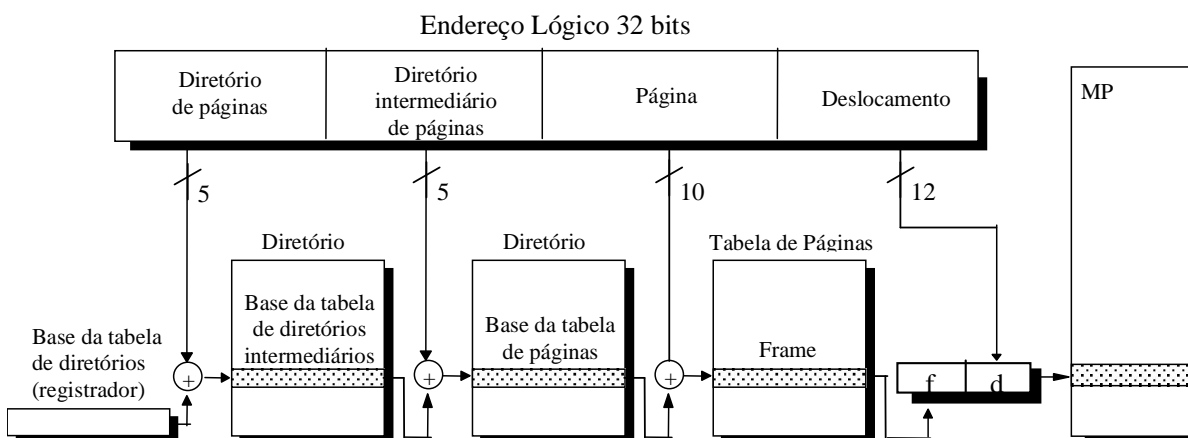
- A maioria dos sistemas operacionais modernos aplica a técnica de memória virtual na gerência de memória
- No caso de sistemas que executam em múltiplas plataformas a gerência de memória pode variar de caso para caso, dependendo do hardware disponível (MMU – *Memory Manager Unity*)

4.5.1 UNIX e SOLARIS

- Versões antigas do UNIX aplicavam um particionamento variável da memória sem memória virtual
- Versões mais modernas (SVR4 e Solaris2.x) implementam memória virtual paginada para os processos de usuário
- A substituição de páginas é feita com uma variação do algoritmo do relógio
- Para as necessidades do kernel (pequenos blocos) é utilizado uma variação do algoritmo de buddy

4.5.2 Linux

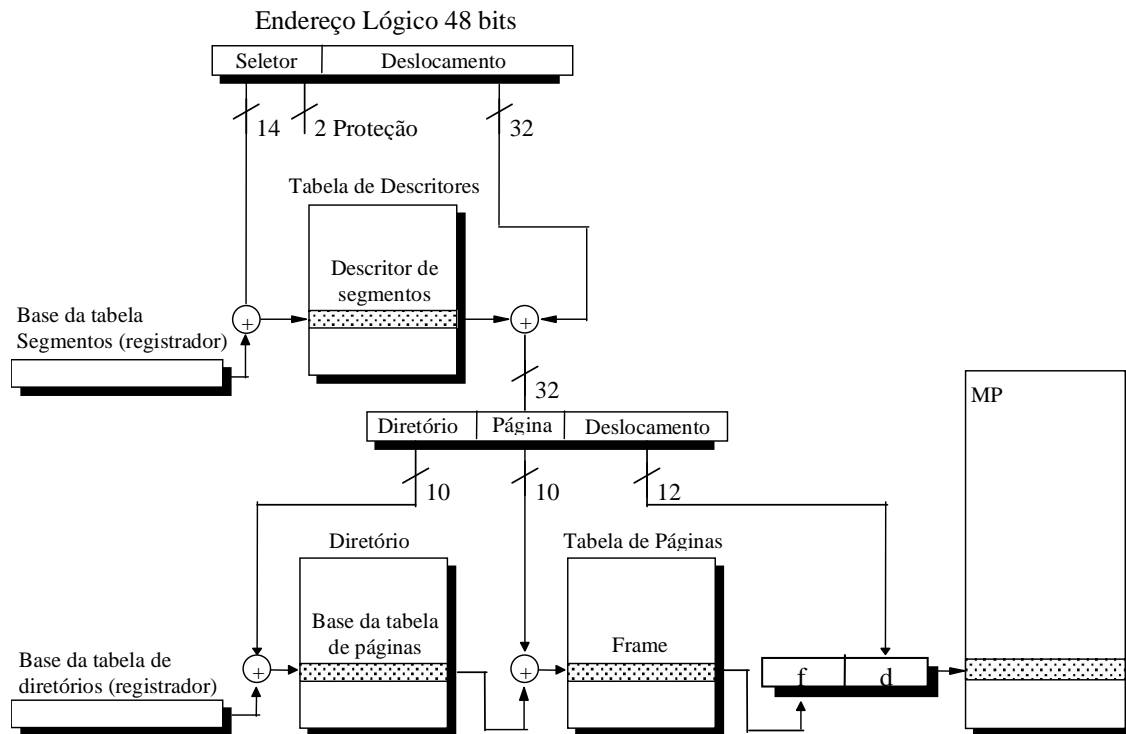
- Tem várias características em comum com gerência de outras implementações de UNIX mas tem suas peculiaridades
- Implementa memória virtual paginada com uma tabela de página de três níveis (diretório, diretório intermediário, página)



- Conceito de diretório reduz o tamanho das tabelas aumentando a complexidade do acesso (várias consultas para converter o endereço)
- A figura acima mostra apenas as tabelas usadas em uma conversão mas a estrutura das tabelas não é linear e sim uma **árvore** (Várias tabelas de diretórios intermediários para cada entrada da tabela de diretórios e várias tabelas de páginas para cada entrada de cada tabela de diretórios intermediários!!!)
- Páginas de 4K na família X86 e 8k na família Alpha

- Para facilitar a portabilidade tem um nível intermediário de gerência de memória *Architecture Independent Memory Model*
- A substituição de páginas é feita com uma variação do algoritmo do relógio
- Alocação dinâmica feita na pilha do sistema (*stack*)
- Para as necessidades do kernel (pequenos blocos) é utilizado uma variação do algoritmo de buddy. O linux implementa o Buddy sobre páginas mas subdivide-as em menores unidades para melhor atender as necessidades do kernel

4.5.2 IBM OS/2 (hardware Intel)



4.5.3 Windows 2000

- Utiliza uma variação da memória virtual paginada
- Divide o seu espaço de memória virtual de 4 Gigabytes (registrador de 32 bits) em 2 Giga para os processos de usuário e 2 Giga para o sistema operacional
- Quando um processo de usuário é disparado ele recebe um número de frames de memória (*Working Set - WS*)
 - Substituições de páginas são efetuadas apenas dentro deste WS, ou seja, só páginas do mesmo processo são candidatas para a substituição
 - Se o número de substituições de um processo for grande e bastante memória estiver disponível, o WS pode ser aumentado
 - Se a memória livre ficar escassa, o sistema operacional diminui o WS dos processos de usuário retirando as suas páginas menos recentemente utilizadas

5 EXERCÍCIOS

1. O que vem a ser uma hierarquia de memória? Qual problema tento resolver com essa solução?
2. Se o tempo para ler um dado da memória principal aumenta com os vários níveis de uma hierarquia de memória, como o tempo médio de acesso pode ficar melhor?
3. Qual o princípio que faz com que uma hierarquia de memória funcione e quais os seus dois tipos (cite exemplos práticos)?
4. O que vem a ser *miss-penalty*? Por que ele tem que ser necessariamente expresso por um tempo médio?
5. Quais os dois principais problemas que os mapeamentos para memória *cache* têm que resolver?
6. Compare o mapeamento de *cache* direto com o associativo em relação à política de substituição, uso de área da *cache* com dados de controle, e a possibilidade de se ter mau aproveitamento da *cache*.
7. Explique como funciona o mapeamento de *cache* conjunto associativo e responda quais são as suas duas principais vantagens.
8. Para uma *cache* com 8 posições desenhe como fica a divisão de conjuntos no caso de associatividade 1 way, 2 way, 4 way e 8 way.
9. Desenhe a divisão de bits do endereço lógico, calcule o aproveitamento efetivo e o tamanho das memórias associativas em Kbytes (se for o caso) para as técnicas de mapeamento de memórias *cache* direta, totalmente associativa e conjunto associativa com 4 conjuntos (4 Gbytes de memória, *cache* com 1024 linhas, bloco de 8 palavras de 32 bits)
10. A área de memória disponível para implementação de uma *cache* L2 é 256 Kbytes. Considerando que a memória a ser endereçada possui 256 Mbytes (2^{28}) e a *cache* deve trabalhar com blocos de 8 palavras de 16 bits calcule para a técnica direta e conjunto associativa (16 conjuntos): Divisão de bits do endereço e número de linhas da *cache*.
11. O que vem a ser o problema de integridade de dados na *cache*? Descreva de forma sucinta as duas possíveis estratégias para sua solução.
12. Por que preciso trazer um dado da memória principal para a *cache* no caso de uma escrita? Este dado não vai ser de qualquer forma apagado pelo dado a ser escrito?
13. Compare a técnica de gerência de memória paginada com a gerência segmentada em relação a fragmentação gerada, alocação de unidades e substituição de unidades ?
14. Desenhe como fica a conversão de endereços na gerência de memória segmento-paginada e descreva os passos de uma conversão.
15. Faça uma tabela com o número de tabelas de páginas, tabelas de segmentos, tabelas de frames e tabelas de gerência de memória para as três técnicas de gerência de memória vistas em aula.
16. O que vem a ser *trashing* e qual a sua principal causa?