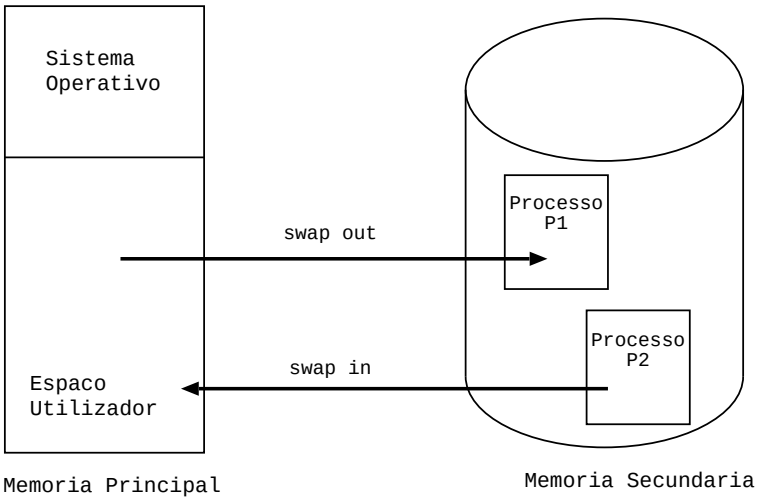
Gestão de Memória

- Um programa para ser executado tem de ser carregado para memória e colocado no âmbito de um processo.
- Por questões de eficiência, um S.O. permite a coexistência em memória de mais de um processo – multiprogramação.
- Questões relevantes a responder em Gestão de Memória:
  - Como organizar a memória de forma a saber-se qual o espaço livre para carregar novos processos e qual o espaço ocupado por processos já em memória?
  - Como associar endereços de variáveis no programa a endereços de memória real?
  - O que fazer se o processo ou conjunto de processos precisar de mais memória do que há disponível?
  - Carregar um processo do disco para memória tem os seus custos, como reduzir os custos de acesso a disco?
- É uma das tarefas fundamentais de um SO, sendo implementado ao nível do kernel.

Swapping

- Num SO é comum que um processo seja, temporariamente, deslocado da memória principal para a memória secundária (disco), e depois carregado novamente para memória para continuar a execução.
- A memória secundária é um disco rápido onde estão guardadas imagens dos processos dos utilizadores e que permitem acesso directo a essas imagens.
- Torna a gestão de memória mais flexível, permitindo mais fácilmente suportar sistemas com time-sharing.
- Esta operação tem custos que se devem, em grande parte, ao tempo de transferência (directamente proporcional à quantidade de memória a transferir).



Partição Simples

- Também designado por: monoprogramação, sem paginação ou swapping.
- Memória dividida em duas partições:
  - SO residente em memória e vector de interrupções.
  - Processos dos utilizadores.
- Neste modelo, em cada momento, apenas um programa reside em memória, sendo carregado de disco ou tape.

Partições Fixas

- Para permitir multiprogramação, dividiu-se a memória em N partições (possivelmente de tamanho diferente).

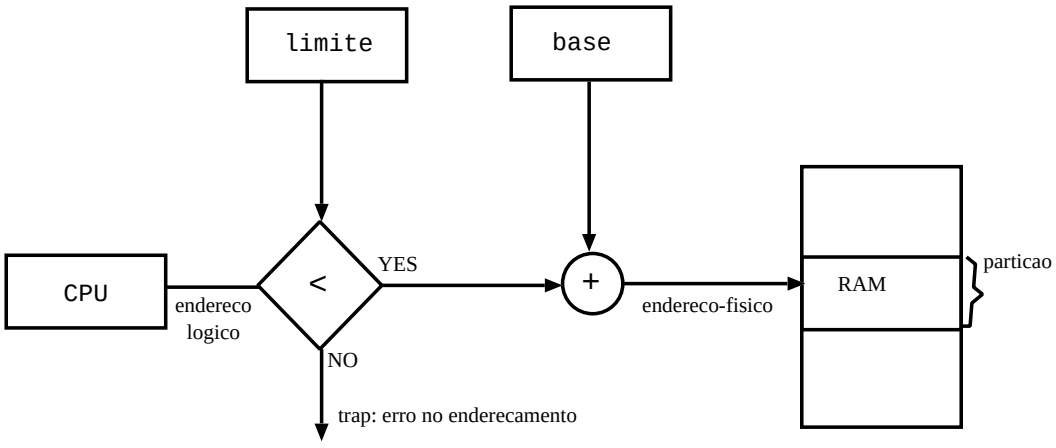
  Quando um processo é carregado para memória, atribui-se-lhe a partição mais pequena, com tamanho suficiente para conter o processo. O espaço da partição que possa sobrar é desperdiçado.
- Os processos a carregar para memória podem esperar numa única fila ou em uma das filas associadas às partições consoante o tamanho da partição que precisa.
- É simples de implementar, mas conduz a um uso ineficiente da memória devido à fragmentação interna e ao número de processos activos, que é fixo à partida.

Recolocação dos programas e protecção

A multiprogramação introduz dois problemas:

- **Recolocação de programas**: os endereços dos programas têm de ser relativos a um endereço base da partição para onde vão ser carregados, sendo o endereço real a soma dos dois (endereço lógico + base).
- **Protecção**: como evitar que um processo "salte" para uma instrução na memória de outro processo, ou que leia ou escreva na memória desse processo?

Uma solução para estes problemas, recolocação e protecção, é usar dois registos especiais de hardware: base e limite.



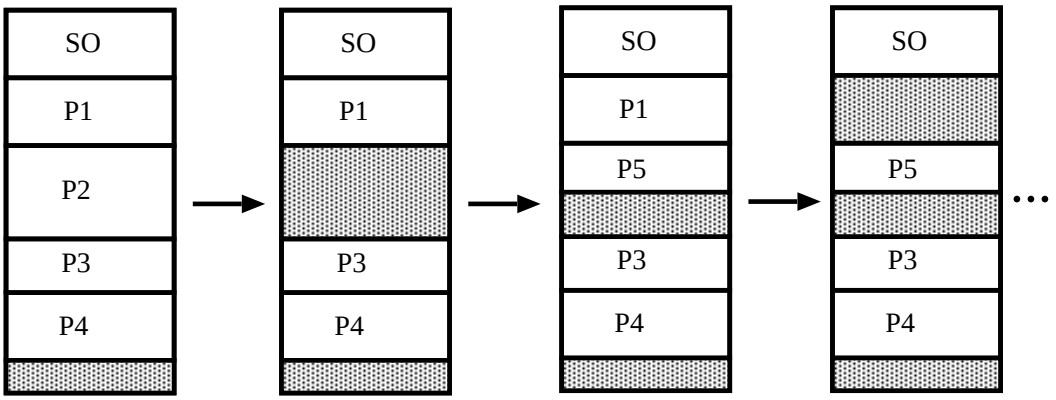
Para cada endereçamento feito, endereço lógico, se este for menor que o tamanho da partição (reg. limite), é-lhe adicionado o endereço-base, para se obter o endereço-real.

Esta solução tem a vantagem de o processo (programa) ficar independente da partição onde foi colocado em 1o. lugar.

Multiprogramação com Partições Variáveis

Neste modelo, as partições a associar aos processos variam dinâmicamente de tamanho, estando dependentes dos processos a executar.

- As partições são criadas dinâmicamente, de forma a que um processo seja carregado para uma partição com o tamanho suficiente para acomodar o processo.



- Uso mais eficiente de memória.
- O tamanho variável das partições dá origem a fragmentação externa (buracos) que pode afectar a performance do gestor.

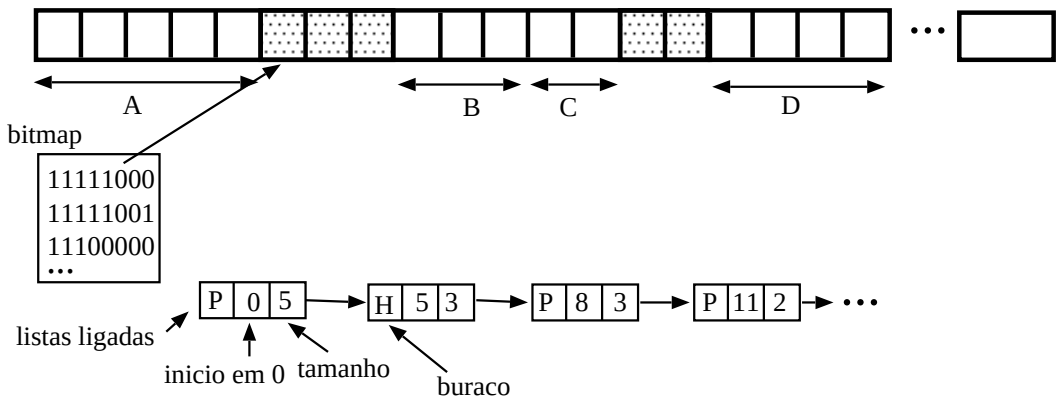
  Uma solução é usar compactação de forma a agrupar os buracos num só, mas é dispendioso em termos de CPU.
- O Gestor de Memória mantém informação sobre quais as partições livres (buracos) e as partições ocupadas. Existem 3 métodos para manter essa informação: bitmaps, listas e buddy.

Bitmaps e Listas ligadas

A memória é dividida em unidades do mesmo tamanho (pode ir de alguns bytes até vários kbytes).

Bitmaps: A cada unidade é associado um bit (1 - usado; 0 - livre).

Listas: cada elemento da lista regista conjunto de blocos ocupados ou livres. Podem estar ordenados por endereços.



Vantagens e inconvenientes:

- **Bitmaps**: extremamente simples ligar blocos livres; mas é lento na procura de uma partição livre para um novo processo. O bitmap tem de ser percorrido para se encontrar uma sequência de k zeros que determinam um espaço de tamanho k unidades para o processo.
- **Listas**: junção de buracos é um pouco complicada. A procura de um buraco pode ser feita com base num de vários algoritmos.

Algoritmos de procura sobre listas ligadas

Como satisfazer um pedido de tamanho N de uma lista de buracos livres?

first-fit usa o 1o buraco suficiente para conter o processo. É rápido na procura.

next-fit relembra a posição onde foi escolhido o último buraco e começa a procurar a partir desse ponto (como se fosse first-fit) em vez de começar do início da lista. Ligeiramente pior que first-fit.

best-fit pesquisa a lista toda à procura do melhor encaixe. Em média é mais lento do que o first-fit e tende a deixar muitos buracos de pequeno tamanho e dispersos.

worst-fit considera sempre o maior buraco, de forma que ao ser dividido em 2 (1 parte para o processo e um novo buraco). Procura a lista toda. O buraco que sobra ainda será útil.

quick-fit mantém listas separadas para os pedidos mais frequentes. É muito rápido a encontrar um buraco, mas tem o inconveniente de quando um processo liberta um segmento, ser muito é díficil encontrar os vizinhos para uma possível junção.

Sistema Buddy

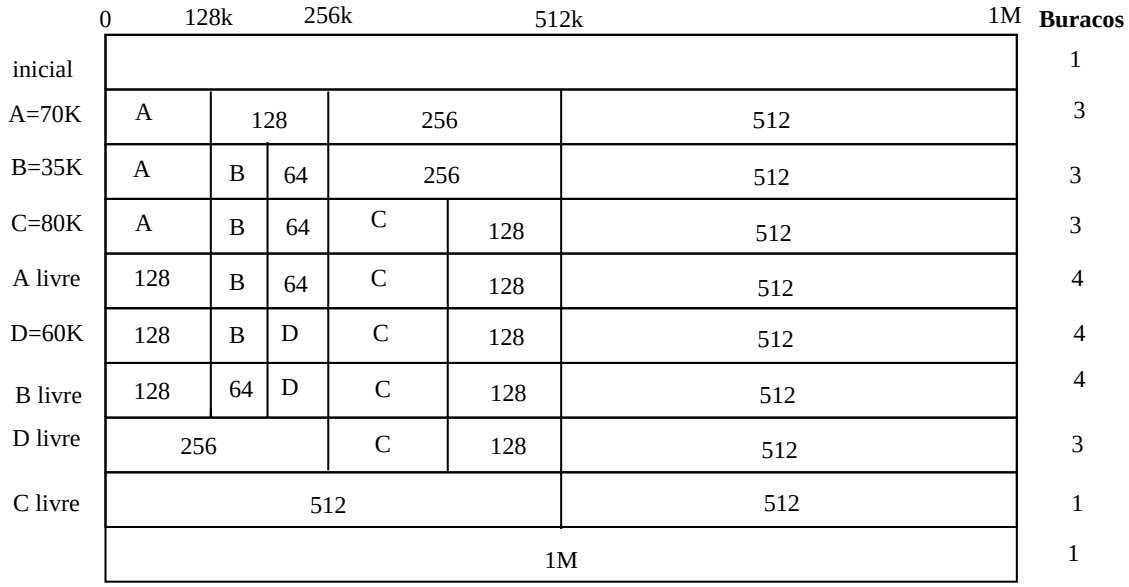
Modelo mais eficiente para gerir a atribuição dinâmica de partições de tamanho variável a processos.

Num sistema buddy, os blocos de memória existem em tamanhos 2^K, $L \leq K \leq U$, onde:

- 2^L – é o menor tamanho de bloco atribuído.
- 2^U – é o maior tamanho de bloco atribuído.

De início toda a memória é vista como um único bloco, tamanho 2^U. Se aparecer um pedido de tamanho s t.q. $2^{U-1} < s \leq 2^U$, então o bloco inteiro é atribuído. Caso contrário, o bloco é dividido em dois blocos (buddies) de tamanho 2^{U-1}. Se $2^{U-2} < s \leq 2^{U-1}$, então o pedido é satisfeito por um dos buddies, senão continua-se a a dividir um dos buddies.

Quando um bloco de tamanho 2^K é libertado, o gestor de memória precisa apenas da lista dos buracos de tamanho 2^K para vêr se a junção é possível.



Este sistema conduz a fragmentação interna, devido aos arredondamentos para uma potência de 2.

Memória Virtual

O que fazer se algum dos programas a executar for demasiado grande para residir como um todo na memória física do sistema?

Overlays: divide-se a aplicação em blocos modulares que são chamados em sucessão para a memória do sistema e desse modo executa-se a aplicação.

Inconvenientes desta solução: díficil programação das aplicações; problemas de portabilidade; maior complexidade no sistema e dificuldade com I/O.

Memória Virtual: (Univ. Manchester 1961) consiste em usar duas noções de endereço:

- *endereço virtual*: no contexto do espaço de endereçamento do processo que pode exceder o tamanho físico da memória;
- *endereço real*: no contexto da execução, apenas uma parte do espaço de endereçamento do processo é carregada para memória, situação em que um endereço virtual irá corresponder a um determinado endereço físico.

Como principal vantagem da memória virtual saliente-se o permitir um uso mais eficiente da memória física.

Paginação

É um outro método de organizar a memória e que se adequa bem a implementação de memória virtual.

- memória física (RAM) é dividida em blocos de tamanho fixo, designadas por *molduras* de página (tamanho potência de 2: entre 512 bytes e 8 kbytes).
- memória lógica (disco) dividida em blocos do mesmo tamanho designados por *páginas*.
- o espaço de endereçamento de um processo pode não ser contíguo (é um conjunto de páginas).
- para executar um programa com tamanho de n páginas, é necessário procurar n molduras livres e carregá-lo para essas molduras.
- é necessário um mecanismo para saber quais as molduras que estão livres.
- é necessário uma tabela de páginas que traduza endereços lógicos (na página) em endereços físicos (na moldura que contém a página).

Endereços Lógicos vs. Físicos

- endereços lógicos (ou virtuais) – gerados pelo CPU.
- endereços físicos – visíveis pela unidade de memória.
- os endereços lógicos e físicos coincidem em tempo de compilação, mas diferem em tempo de execução (recolocação).
- a MMU (Memory-Management Unit), periférico de hardware, tem a responsabilidade de traduzir endereços lógicos em endereços físicos:
- um endereço virtual é composto por dois campos:
  - *Número de página* (p) – usado como um índice para a tabela de páginas que contém os endereços base de cada página em memória.
  - *Deslocamento* (d) – é combinado com o endereço-base para definir o endereço físico para ser enviado à unidade de memória.

Tradução de endereços pela MMU

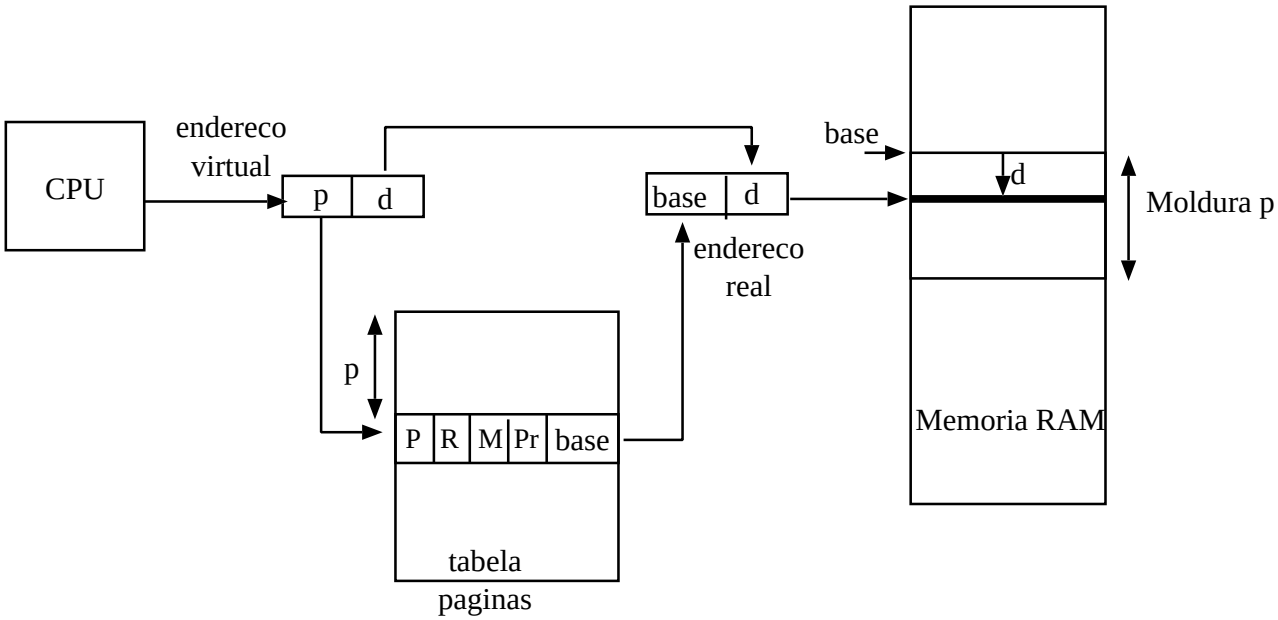


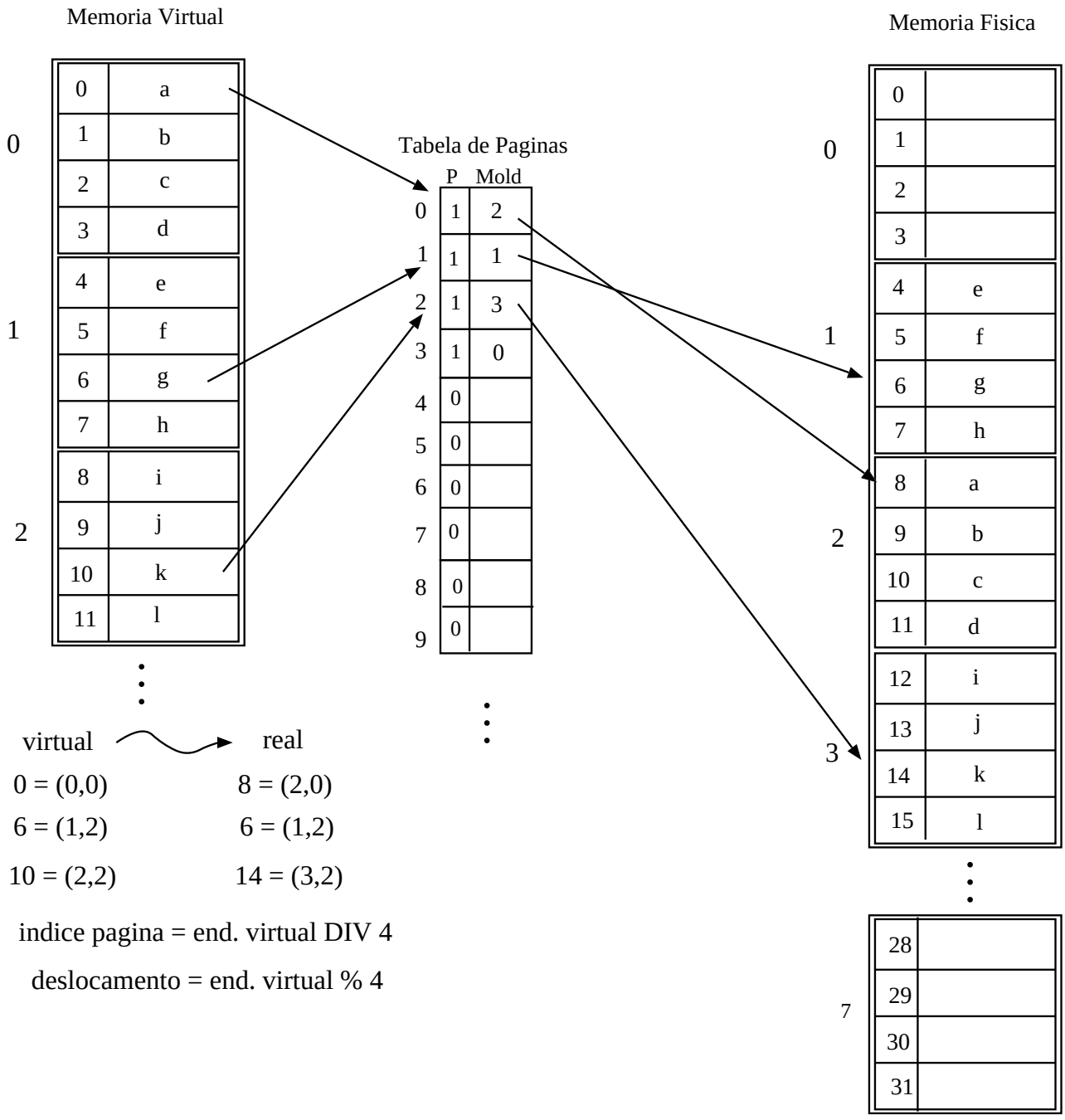
Tabela de páginas

Reside em memória e cada entrada da tabela dá informação sobre o estado das páginas do processo, nomeadamente:

- `P` bit de presença. Se P=0, a MMU gera uma interrupção – *falta de página*, procedendo-se de seguida à cópia da página do disco para a moldura de página correspondente.
- `R` referenciada, `M` modificada - bits que permitem saber se a página está a ser usada.
- `Pr` - informação de protecção no acesso à página.
- `base` - endereço base que combinado com o deslocamento do endereço virtual dá o endereço real.

Exemplo de tradução de endereços

Memoria fisica: 32 palavras Tamanho de pagina: 4 palavras Molduras de pagina: 8



Dificuldades com Paginação

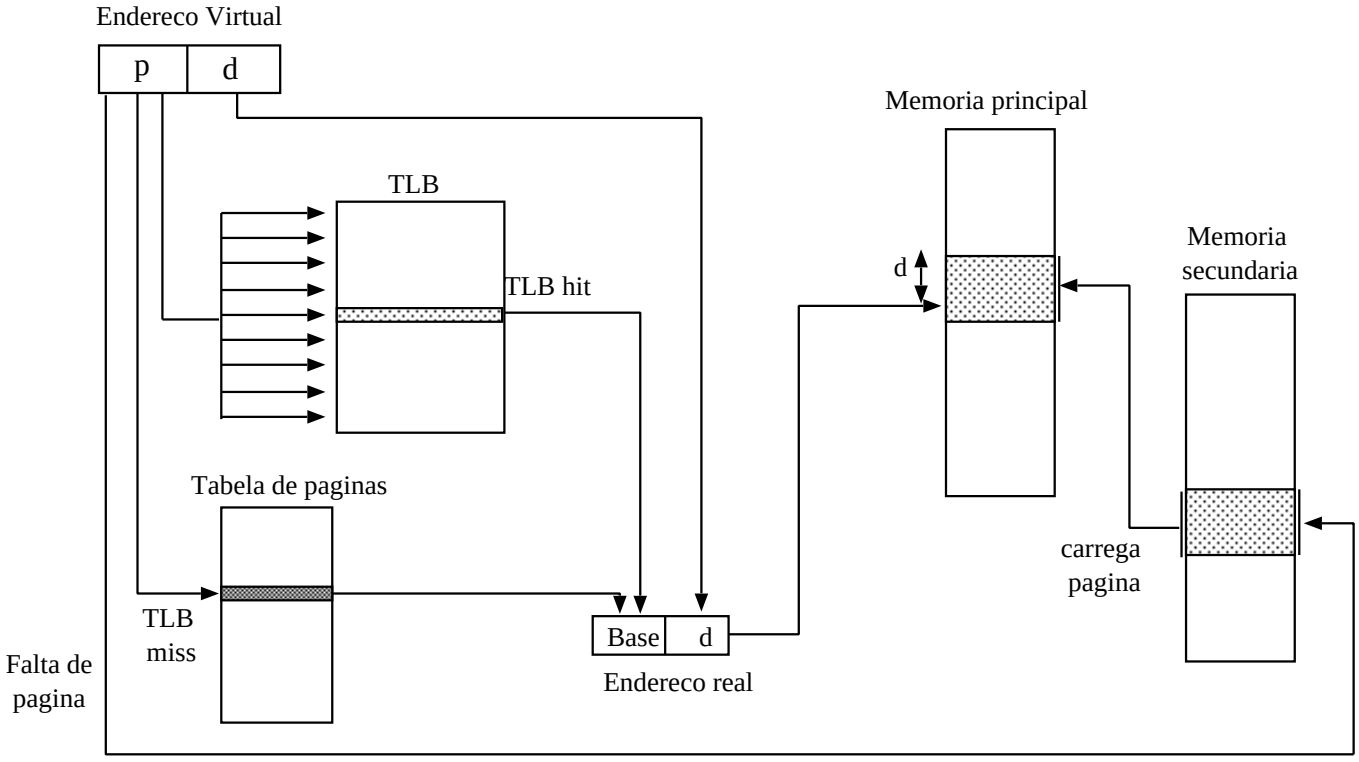
- A implementação eficiente e simples das tabelas de páginas requer que estas residam em memória, o que só é possível em sistemas com um número pequeno de páginas de memória virtual.
- A tradução de endereços virtuais em endereços reais deve ser muito eficiente, pois uma percentagem importante das instruções de um programa usam acessos a memória.
- Em sistemas com um grande espaço virtual de endereçamento, os custos de transferência das tabelas de páginas podem ser extremamente elevados.
- *Custos no acesso a memória:*
  - sem paginação – uma instrução que referencie a memória necessita de apenas 1 acesso à memória para a sua execução.
  - com paginação – são necessários dois acessos à memória, um para consultar a tabela de páginas e outro para aceder ao endereço real.

  Dadas as diferenças abismais de velocidade entre a memória e CPU, como minimizar os custos resultantes do acesso à memória?

  Dado que a maioria dos programas exibe *localidade de referência*, i.e acedem muitas vezes a um pequeno número de páginas, então usar uma estrutura de acesso rápido que funcione como *cache* será uma boa solução.
- Solução: usar *memórias associativas* (TLBs - Transaction Lookaside Buffer).

Memórias Associativas (TLBs)

Têm a particularidade de todas as suas entradas poderem ser comparadas com um valor em paralelo, de uma só vez.



Designa-se por **hit-ratio** a percentagem de referências à memória satisfeitas pelo TLB. **miss-ratio** corresponde à percentagem de falhas (= 100 - hit-ratio).

Um acesso à memória que seja satisfeito pela TLB é muito mais rápido do que um acesso via tabela de páginas, pelo que pode dizer-se que:

maior hit-ratio ⇒ melhor performance

Em geral, a performance média no acesso a memória depende:

- do tempo de acesso à TLB;
- do tempo de acesso à tabela de páginas;
- do hit-ratio (o tamanho da TLB é importante).

Exemplo de tempos de acesso à memória

Considere-se:

hit-rate=80%
procura na TLB= 20 ns
acesso à memória= acesso à tabela de páginas= 100 ns

Tempo de acesso no caso de TLB-hit:

t1= 20 + 100 = 120 ns

Tempo de acesso no caso de TLB-miss:

t2= 20 + 100 + 100 = 220 ns

Tempo efectivo de acesso (TEA) à memória para um hit-rate de 80% é:

TEA= 0.8*120 + 0.2*220 = 140 ns

Se hit-rate= 90%, então TEA= 0.9*120+0.1*220= 130ns

O hit-rate está relacionado com o tamanho da TLB. É comum ter-se valores de hit-rate na ordem dos 98%.

Algoritmos de Substituição de Páginas

Dado que a memória física é normalmente muito menor que a memória virtual, é natural que durante a execução as molduras de página acabem por ficar todas ocupadas. Assim, na implementação de memória virtual é indispensável usar-se um algoritmo que determine qual a página em memória que deve ser substituída para permitir que outra possa ser carregada. Alguns dos algoritmos mais comuns são:

Página Óptima: substitui a página que irá ser referenciada num futuro mais distante. Não é implementável na sua versão pura, pois não é possível determinar as páginas nessas condições em tempo de execução.

Página Não Usada Recentemente: NRU associa a cada página de memória virtual dois bits, R e M, que indicam se a página foi referenciada e/ou modificada recentemente. As páginas são então divididas em 4 grupos de acordo com os estados possíveis destes 2 bits:

```
classe 0: (R=0, M=0)  classe 1: (R=0, M=1)
classe 2: (R=1, M=0)  classe 3: (R=1, M=1)
```

O algoritmo escolhe aleatoriamente uma página pertencente à classe com numeração mais baixa, e simultaneamente, não vazia.

Neste algoritmo, quando um processo inicia a sua execução os bits R e M de todas as suas páginas estão a 0. Sempre que uma página for referenciada (por leitura) ou modificada (por escrita) os bits correspondentes passam a 1. Periódicamente (em cada interrupção do relógio), o bit R é colocado a 0, para que seja possível distinguir páginas que não tenham sido referenciadas recentemente das que foram. Isto pode levar a que uma página que tenha os dois bits a 1, passe da classe 3 para a classe 1.

→ fácil de perceber e implementar; performance razoável.

Algoritmos de Substituição de Páginas (cont.)

Primeira a Entrar, Primeira a Sair: o sistema mantém uma lista de todas as páginas actualmente em memória física por ordem cronológica de transferência. Neste contexto, a página à cabeça da lista (a mais antiga) é substituída.

→ este algoritmo pode remover páginas que estejam a ser muito referenciadas, o que é mau! Raramente usado na sua forma pura.

Segunda Tentativa: idêntico ao anterior, mas a cada página está associado um bit R indicando se foi referenciada recentemente. O algoritmo substitui a página mais antiga com o bit R=0.

Se a página que está no início da fila tem o bit R a 1, ela é deslocada para o fim da fila e o bit R é colocado a 0. O algoritmo continua com a página que fica à cabeça da fila, procedendo de modo análogo até que encontre uma página com o bit R a 0.

Algoritmo do relógio: optimiza o algoritmo "segunda tentativa", diminuindo os custos de manipulação da lista de páginas mantendo-as numa lista circular e manipulando apenas apontadores.

O algoritmo analisa a página que está sob o ponteiro do relógio (actual início da fila). Se esta tiver o bit R a 1, o bit R passa a zero e o ponteiro avança para a página seguinte da lista circular (novo início da fila). O algoritmo repete-se até encontrar uma página com bit R igual a 0. Essa página é substituída, e o ponteiro do relógio avança para a página seguinte.

Algoritmos de Substituição de Páginas (cont.)

Menos Usada recentemente: (LRU) explora a propriedade da localidade de referência das aplicações. O algoritmo escolhe a página que foi referenciada pela última vez há mais tempo.

Baseia-se na observação de que páginas que foram muito usadas recentemente, terão tendência para continuar a serem muito usadas próximamen

A implementação deste algoritmo na sua forma pura é dispendiosa pois requer que se mantenha uma lista ligada de todas as páginas em memória, com a menos usada no início e a mais usada no fim. A dificuldade está em actualizar as posições das páginas na lista sempre que se verifiquem acessos às páginas.

Uma alternativa de implementação é não ordenar as páginas, usar a indexação da tabela de páginas, e associar a cada entrada de página o tempo do último acesso. Para determinar a página menos usada recentemente, é necessário percorrer todas as entradas de páginas para determinar a que tem o tempo mais antigo.

Algoritmos de Substituição de Páginas (cont.)

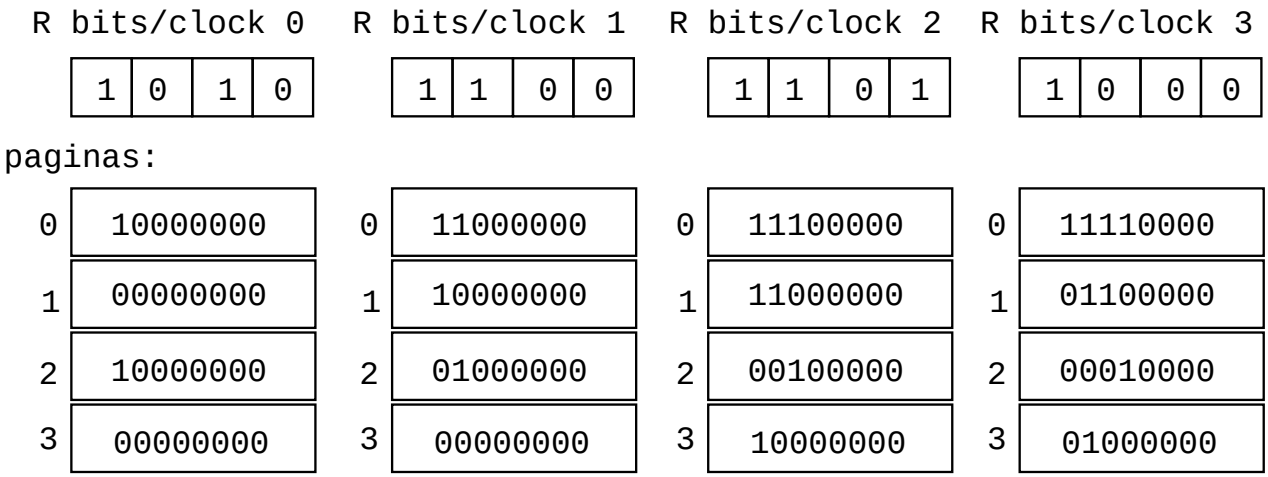
Menos Usada Frequentemente: NFU corresponde a uma realização do algoritmo LRU por software. Este algoritmo, associa um contador a cada página sendo o seu valor inicial zero. Em cada interrupção-do-relógio, o algoritmo percorre todas as páginas em memória e adiciona o bit R de cada página ao contador. Estes contadores indicam o quanto a página é referenciada.

O algoritmo escolhe a página com menor valor de contador para substituir.

Um problema com este algoritmo é que se uma página for muito acedida numa fase inicial, o seu contador é incrementado para valores elevados, fazendo com que mais tarde, mesmo não estando a ser usada possa não ser escolhida para ser substituída.

Uma técnica de envelhecimento a aplicar consiste no seguinte. Usar contadores de 8 bits. Em cada interrupção-do-relógio, faz-se um deslocamento de 1 bit para a direita sobre cada contador antes de se lhe adicionar o bit R. O bit R deve ser adicionado no bit mais à esquerda (bit 8) do contador.

Com esta técnica asseguramos que uma página recentemente usada fica com um valor elevado, mas se entretanto deixar de ser usada, em cada interrupção do relógio o seu valor vai diminuindo (vai sendo dividido por 2). Veja-se o exemplo da figura:

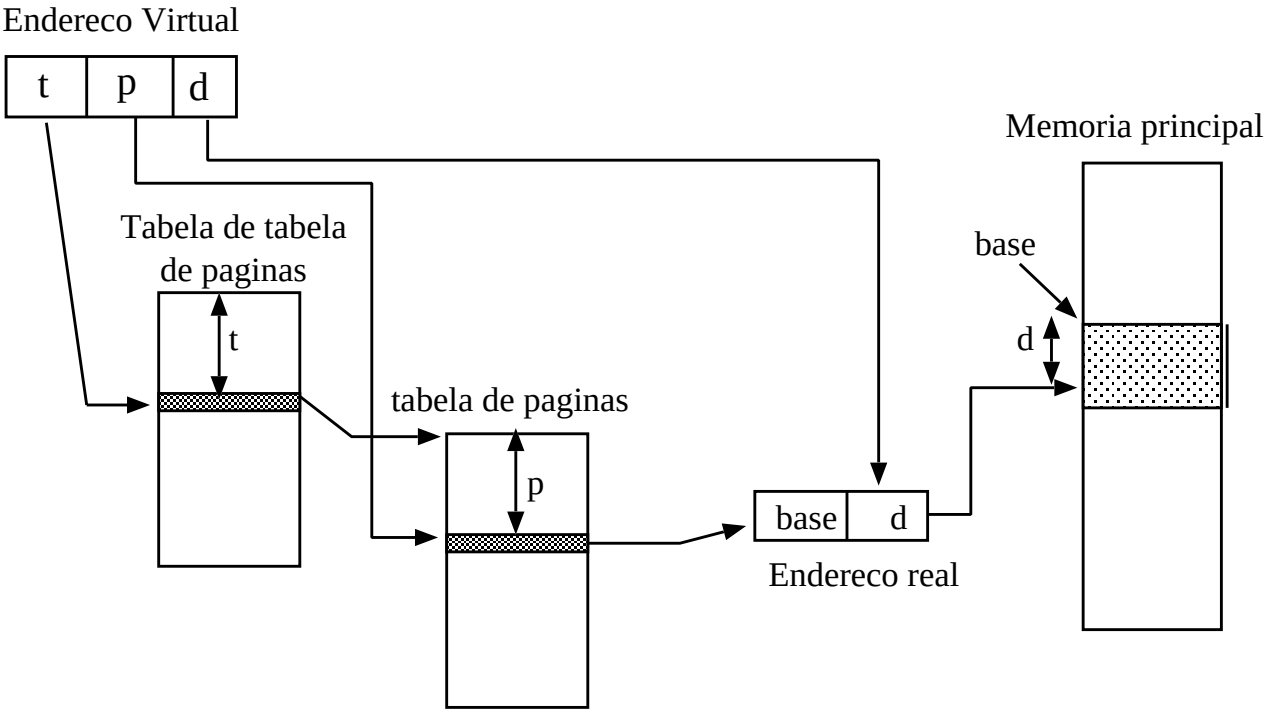


Paginação com níveis múltiplos

Num sistema baseado em paginação, a tabela de páginas pode tornar-se demasiado grande e originar gastos excessivos de memória. Pode então pensar-se paginar a tabela de páginas e apenas ter em memória as partes da tabela (páginas) que são relevantes.

Uma forma de reduzir o número de "páginas" da tabela de páginas a carregar para a memória é criar níveis de indexação, i.e. usar paginação com N-níveis.

Na paginação a 2-níveis, os endereços virtuais incluem agora 3 componentes de bits que controlam o número de tabelas de páginas, o número de páginas por tabela e o tamanho das páginas.



Segmentação

A memória virtual paginada tem apenas uma dimensão, pois os endereços virtuais vão de 0 até um máximo. No entanto, para muitas aplicações seria preferível ter 2 ou mais espaços de endereçamento virtual em vez de um.

Um exemplo de uma tal aplicação é um compilador, pois pode ser visto como um conjunto de componentes lógicas:

– tabela de simbolos,
– código,
– árvore de parsing,
– tabela de constantes e
– pilha de execução.

Cabe ao programador fazer a gestão de cada uma destas áreas, nomeadamente detectar se ocorre *overflow* do espaço reservado, por exemplo, à tabela de código.

A segmentação permite suportar múltiplas visões da memória, podendo-se associar componentes de um programa a espaços de endereçamento virtuais independentes entre si.

→ Alguma vantagem? Sim, a nível de protecção e partilha de recursos.

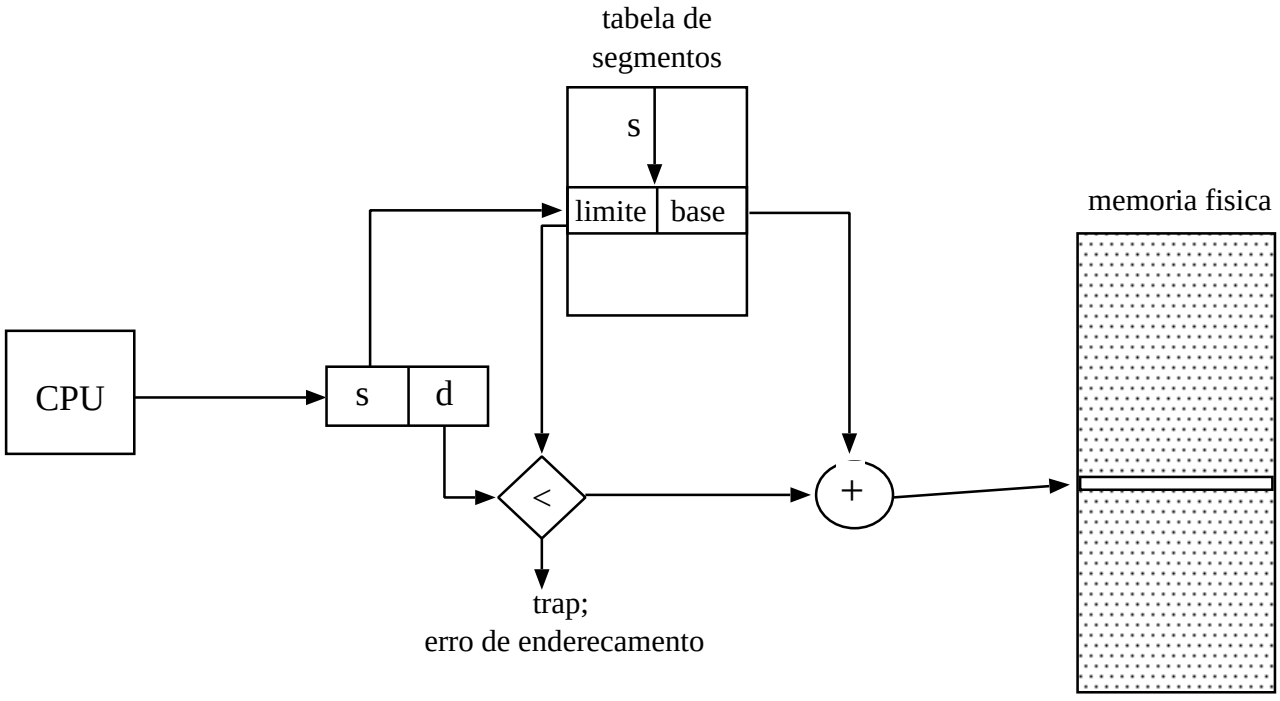
Tradução de endereços com segmentação

- Um endereço lógico (ou virtual) contém agora 2 componentes:

  número do segmento e *deslocamento*.

- *tabela de segmentos* – mapeia endereços bidimensionais definidos pelo utilizador em endereços físicos (uni-dimensionais); cada entrada da tabela contém:

  – *base* – endereço na memória física onde tem início o segmento;
  – *limite* – determina o tamanho do segmento.



Implementação de segmentação

Por uma questão de eficiência a tabela de segmentos deve estar em memória, pelo que se usa dois registos especiais de forma a tornar o acesso mais rápido:

- *STBR (segment-table base register)* – registo que contém o endereço da tabela de segmentos em memória.
- *STLR (segment-table length register)* – número de segmentos usados no programa.
  $\rightarrow$ um dado segmento s é válido se $s < STLR$.

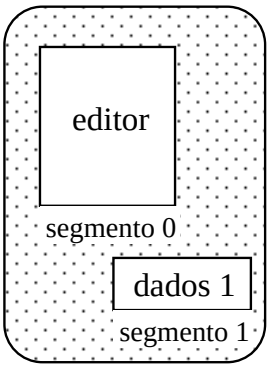
Como encontrar e reservar memória para os segmentos de uma aplicação?

- problema análogo ao de partições dinâmicas de tamanho variável; habitualmente usa-se o best-fit ou first-fit.
- problemas de fragmentação externa.

Vantagens da segmentação:

- *protecção associada aos segmentos*: é possível especificar se um segmento é de leitura ou de escrita. A verificação de protecção é automática.
- *partilha de segmentos*: é possível dois processos partilharem um segmento, bastará que nas respectivas tabelas de segmentos o endereço do início do segmento em memória seja o mesmo.

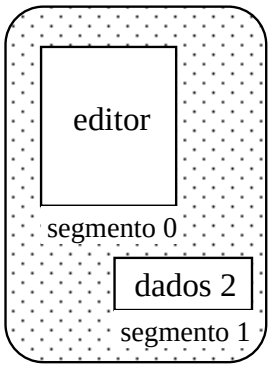
Exemplo de partilha de segmentos



memoria local
ao processo P1

	limite	base
0	25286	43062
1	4425	68348

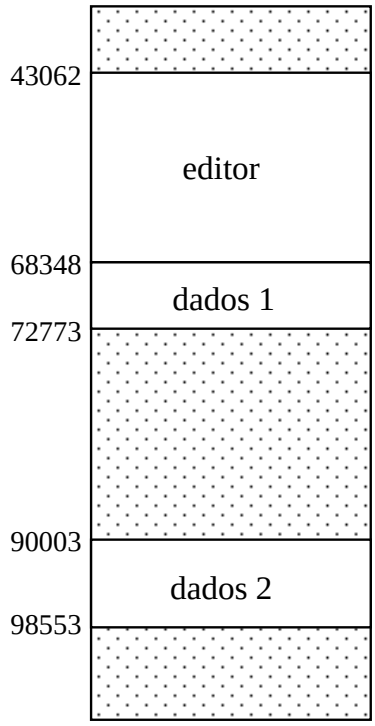
tab-segmentos
processo P1



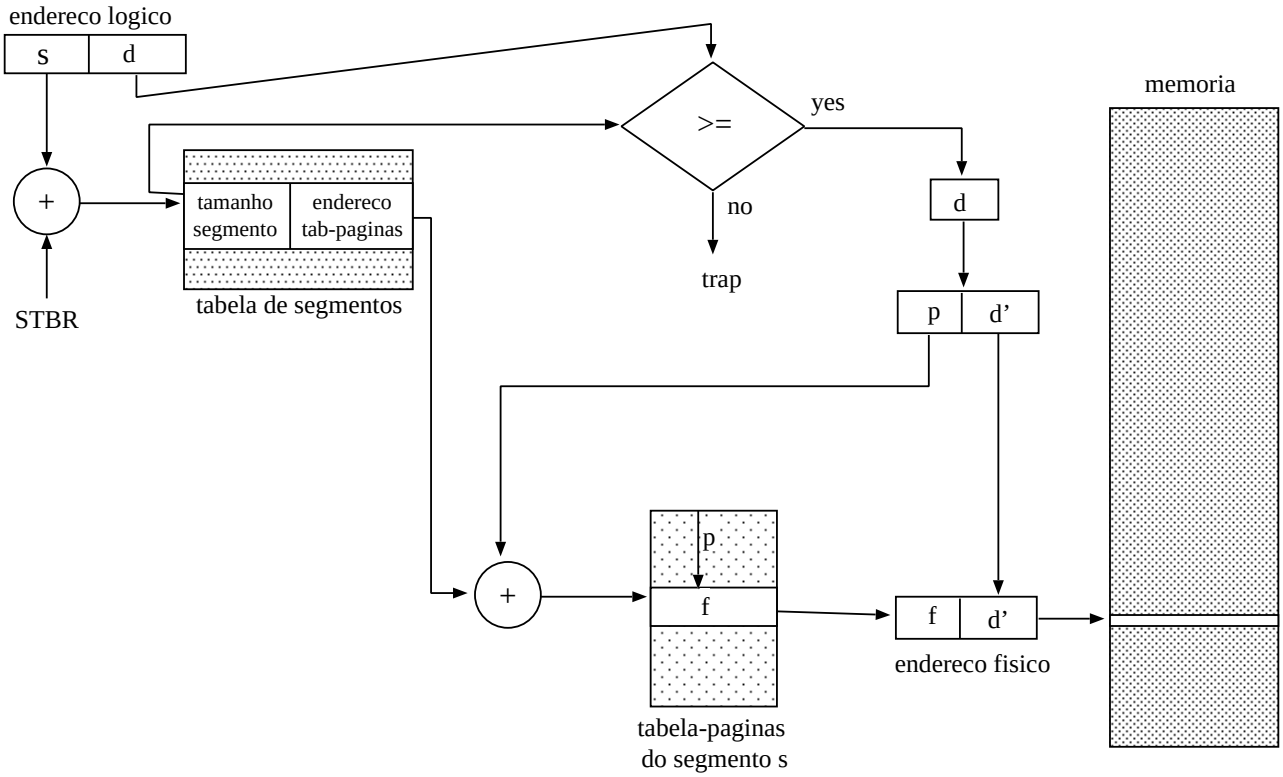
memoria local
ao processo P2

	limite	base
0	25286	43062
1	8850	90003

tab-segmentos
processo P2



Segmentação com paginação



- Os endereços lógicos passam a conter em si: o número do segmento, o número da página dentro do segmento e o deslocamento dentro da página.
- O MULTICS foi o primeiro sistema a usar este esquema de memória.
- O Intel 386 também usava um esquema semelhante de gestão de memória: segmentação e paginação de 2-níveis.