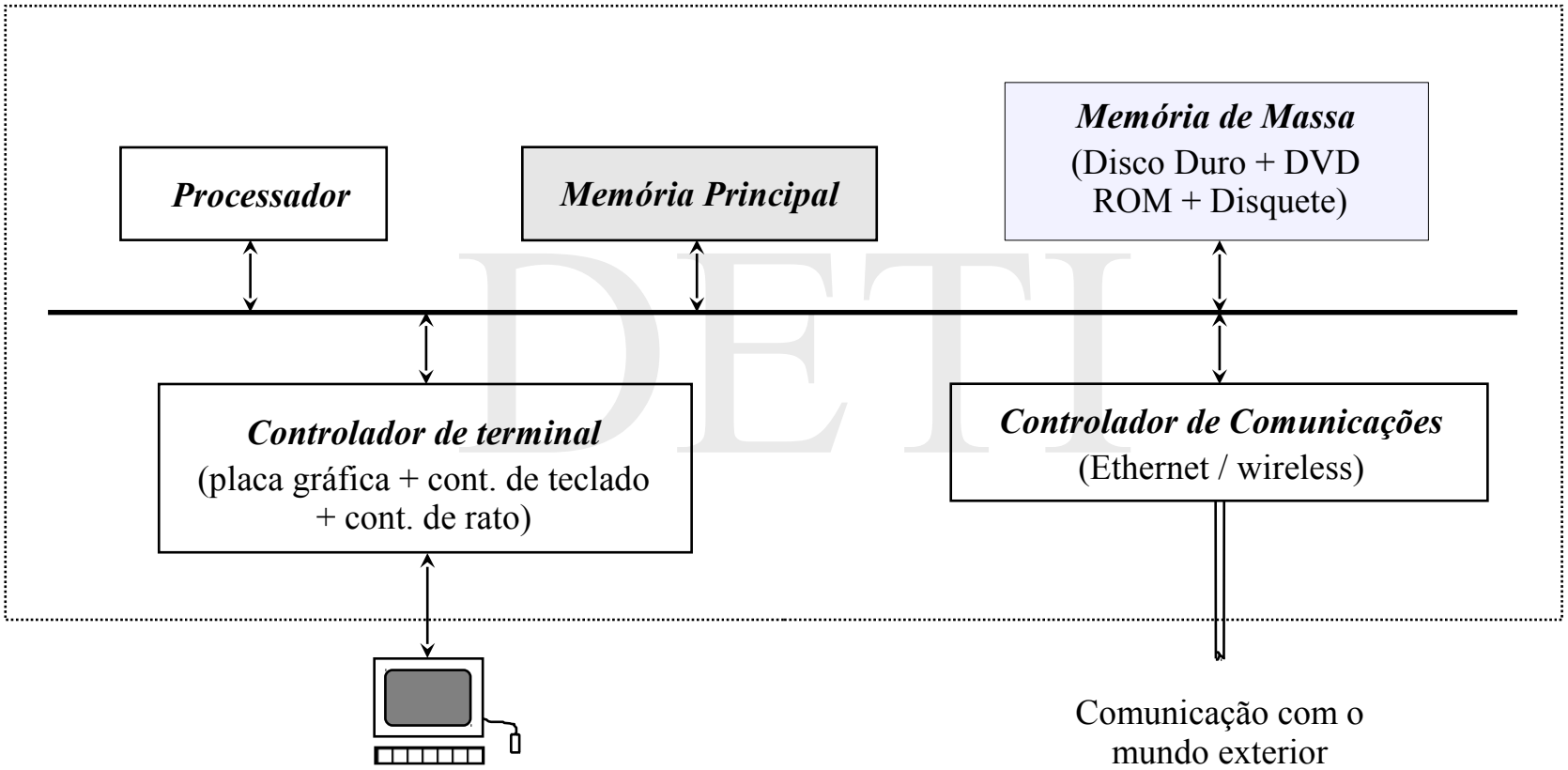
Sistemas de Operação

Gestão da Memória

António Rui Borges / Artur Pereira

Sumário

Sistema computacional



Porquê a gestão da memória

- O espaço de endereçamento de um processo tem que estar, pelo menos parcialmente, residente em *memória principal* para que ele possa ser executado;
- Para maximizar a ocupação do processador e melhorar o *tempo de resposta* (ou o tempo de *turn around*), um sistema computacional deve manter vários processos residentes em *memória principal*;
- Embora a *memória principal* tenha vindo a crescer a um ritmo incessante ao longo dos anos, é um facto que

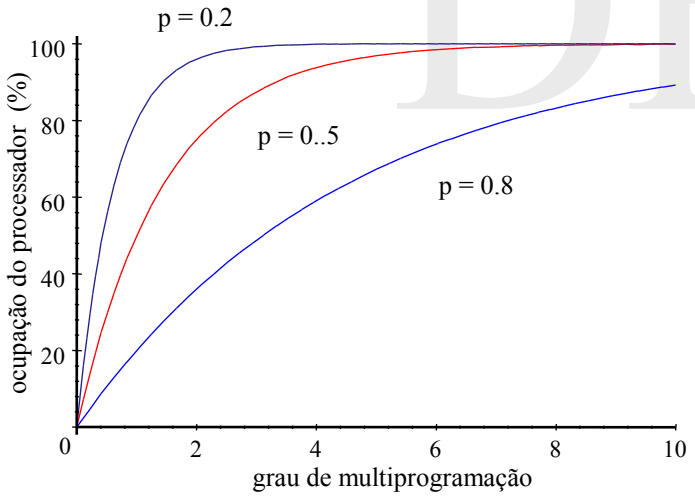
"*os programas tendem a expandir-se preenchendo toda a memória disponível*" - ***Lei de Parkinson***.

Porquê a gestão da memória

fração de ocupação do processador $= 1 - p^n$ (modelo simplificado)

p - fração do tempo em que um processo aguarda bloqueado pelo completamento de operações de I/O, sincronização, etc

n - número de processos que correntemente coexistem em memória principal



N. de processos em MP	*% de ocupação do P*
4	59
8	83
12	93
16	97

p = 0,8

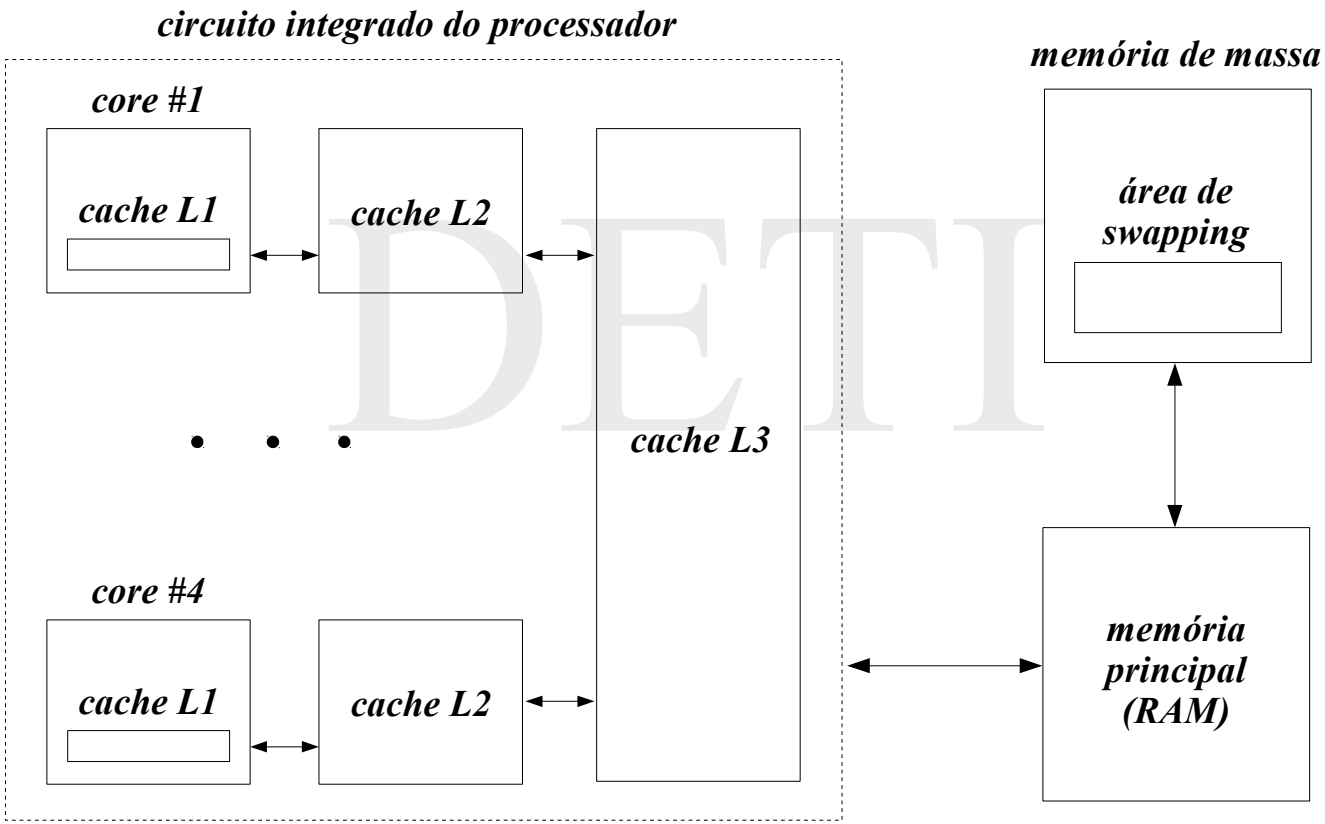
Hierarquia da memória

- Idealmente, o programador de aplicações gostaria de ter disponível uma memória que fosse infinitamente grande, infinitamente rápida, não volátil e barata
  - Na prática, tal não é possível
- Assim, a memória de um sistema computacional está organizada tipicamente em níveis diferentes, formando uma hierarquia
  - *memória cache* - pequena (de dezenas de *KB* a unidades de *MB*), muito rápida, volátil e cara
  - *memória principal* - de tamanho médio (centenas de *MB* ou unidades de *GB*), volátil e de velocidade de acesso e preço médios
  - *memória secundária* - grande (dezenas, centenas ou milhares de *GB*), lenta, não volátil e barata

Hierarquia da memória

- a *memória cache* vai conter uma cópia das posições de memória (instruções e operandos) mais frequentemente referenciadas pelo processador no passado próximo
  - para maximizar a velocidade de acesso, face à frequência de relógio dos proces-sadores atuais, a *memória cache* está localizada no próprio circuito integrado do processador (*nível 1*) e num circuito integrado autónomo colado no mesmo substrato (*níveis 2 e 3*), e o controlo da transferência de dados de e para a *memória principal* é feito de um modo quase completamente transparente ao programador de sistemas
- a *memória secundária*, por outro lado, tem duas funções principais
  - *sistema de ficheiros* - servir como arrecadação para armazenamento mais ou menos permanente de informação (programas e dados);
  - *área de swapping* - servir como extensão da memória principal para que o tamanho desta não constitua um fator limitativo ao número de processos que podem correntemente existir.

Hierarquia da memória



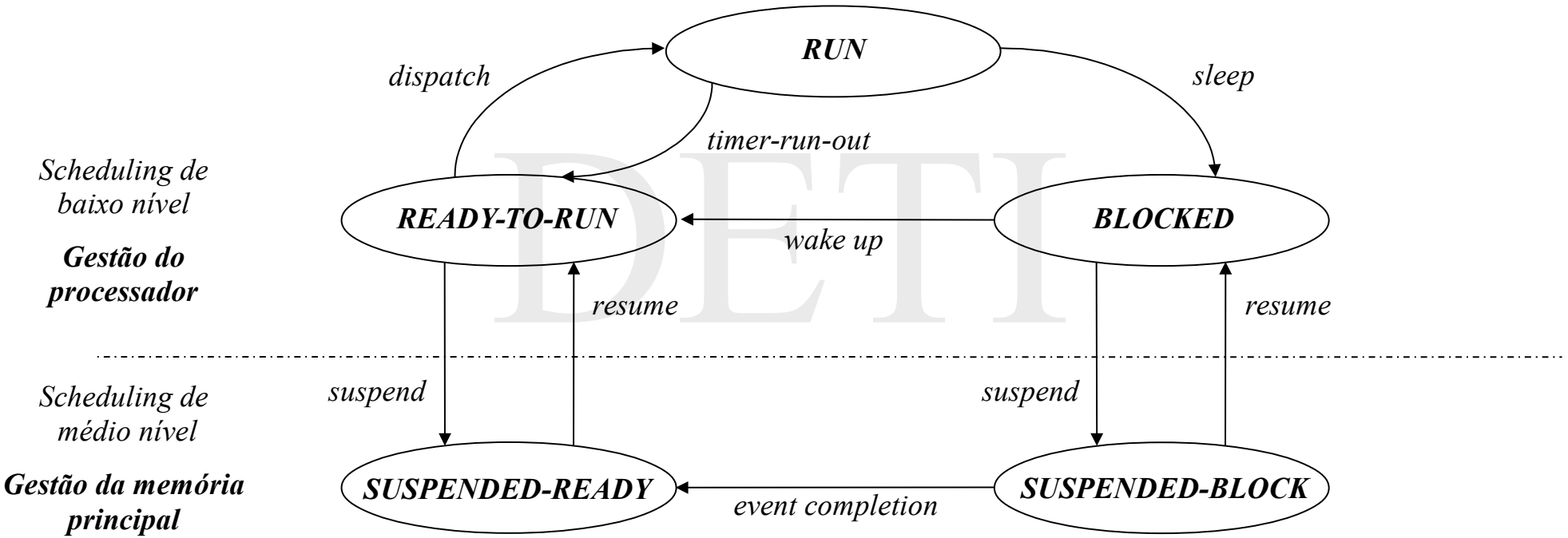
Hierarquia da memória

- Este tipo de organização baseia-se no pressuposto que quanto mais afastado uma instrução, ou um operando, está do processador, menos vezes será referenciado
  - Nestas condições, o tempo médio de uma referência aproxima-se tendencialmente do valor mais baixo
- A justificação deste pressuposto é o *princípio da localidade de referência*;
  - Constatação heurística sobre o comportamento de um programa em execução que estabelece que as referências à memória durante a execução de um programa tendem a concentrar-se em frações bem definidas do seu espaço de endereçamento durante intervalos mais ou menos longos.

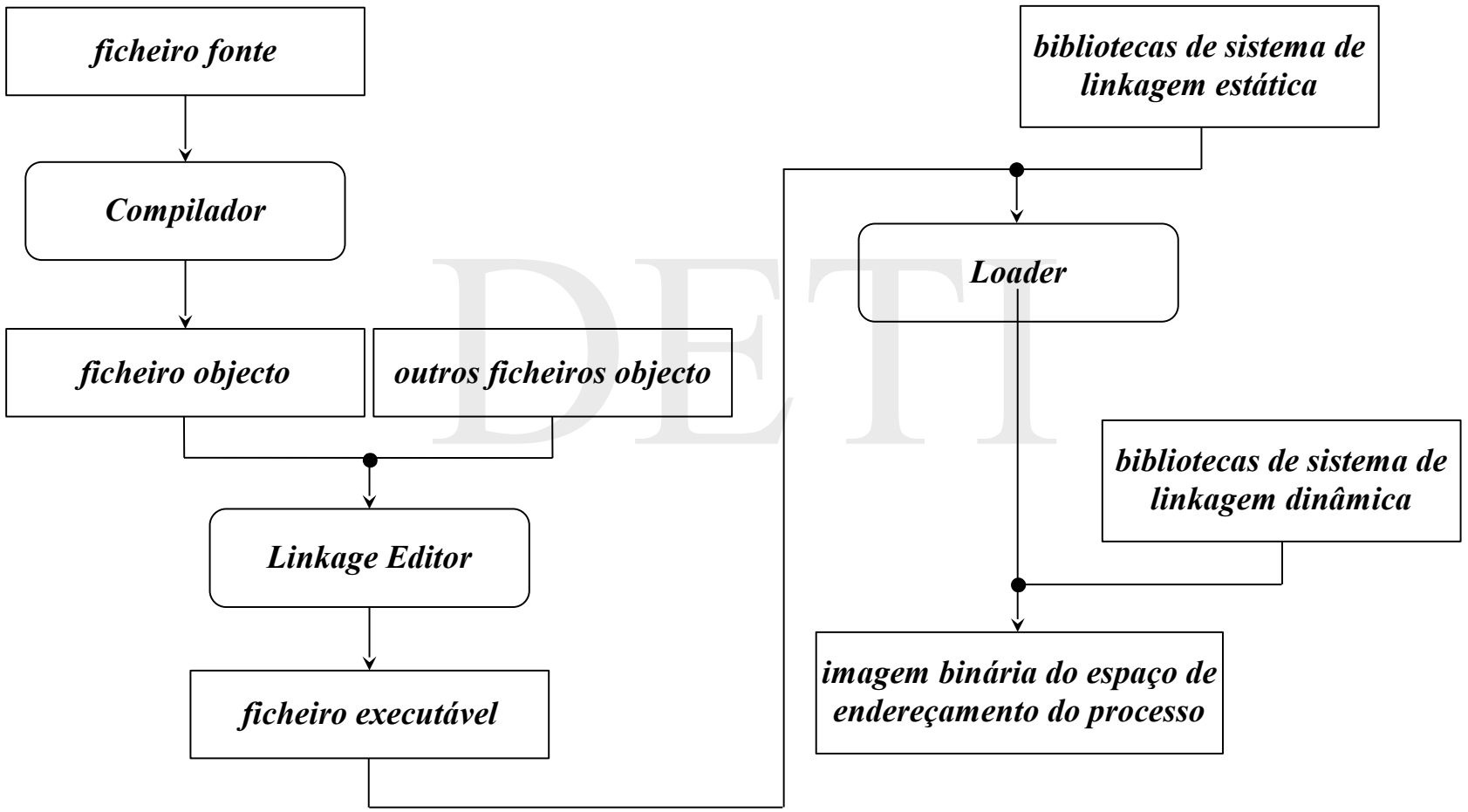
Papel da gestão de memória em multiprogramação

- O papel da gestão de memória num ambiente de multiprogramação centra-se no controlo da transferência de dados entre a *memória principal* e a *memória secundária*, visando
  - a manutenção de um registo sobre as partes da memória principal que estão ocupadas e aquelas que estão livres;
  - a reserva de porções da memória principal para os processos que dela vão precisar, ou a sua libertação quando já não são necessárias;
  - a transferência para a *área de swapping* de todo o, ou parte do, espaço de endereçamento de um processo quando a memória principal é demasiado pequena para conter todos os processos que coexistem.

Papel da gestão de memória em multiprogramação



Espaço de endereçamento



Espaço de endereçamento

- Os *ficheiros objecto*, resultantes do processo de compilação, são *ficheiros relocatáveis* – os endereços das diversas instruções e das constantes e variáveis são calculados a partir do início do módulo, por convenção o endereço 0;
- O papel do processo de *linkagem* é reunir os diferentes *ficheiros objecto* num ficheiro único, o *ficheiro executável*, resolvendo entre si as várias referências externas;
- As *bibliotecas de sistema* podem **não** ser incluídas no *ficheiro executável* para minimizar o seu tamanho;
- O *loader* constrói a *imagem binária do espaço de endereçamento do processo*, que eventualmente será posto em execução, combinando o ficheiro executável e, se for o caso, as bibliotecas de sistema, e resolvendo todas as referências externas que restam;

Espaço de endereçamento

- Por oposição à situação anterior, designada de *linkagem estática*, na *linkagem dinâmica*
  - Cada referência no código do processo a uma rotina de sistema é substituída por um *stub* (pequeno conjunto de instruções que determina a localização de uma rotina específica, se já estiver residente em memória principal, ou promove a sua carga em memória, em caso contrário)
  - Quando um *stub* é executado, a rotina associada é identificada e localizada em memória principal, o *stub* substitui então a referência ao seu endereço no código do processo pelo endereço da rotina de sistema e executa-a
  - Quando essa zona de código for de novo atingida, a rotina de sistema é agora executada diretamente
- Todos os processos que utilizam uma mesma biblioteca de sistema, executam a mesma cópia do código, minimizando portanto a ocupação da memória principal

Espaço de endereçamento

ficheiro fonte

```
#include   <stdio.h>
#include   <stdlib.h>

int main (void)
{
  printf ("hello, world!\n");
  exit (EXIT_SUCCESS);
}
```

produção do ficheiro objecto

```
$ gcc -Wall -c hello.c
```

produção do ficheiro executável

```
$ gcc -o hello hello.o
```

Espaço de endereçamento

```
$ file hello.o
hello.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
$

$ objdump -fstr hello.o
hello.o:     file format elf32-i386
architecture: i386, flags 0x00000011:
HAS_RELOC, HAS_SYMS
start address 0x00000000
SYMBOL TABLE:
00000000 l    df *ABS*  00000000 hello.c
00000000 l    d  .text           00000000
00000000 l    d  .data           00000000
00000000 l    d  .bss            00000000
00000000 l    d  .rodata         00000000
00000000 l    d  .note.GNU-stack 00000000
00000000 l    d  .comment        00000000
00000000 g     F .text  0000002a main
00000000         *UND*  00000000 printf
00000000         *UND*  00000000 exit
```

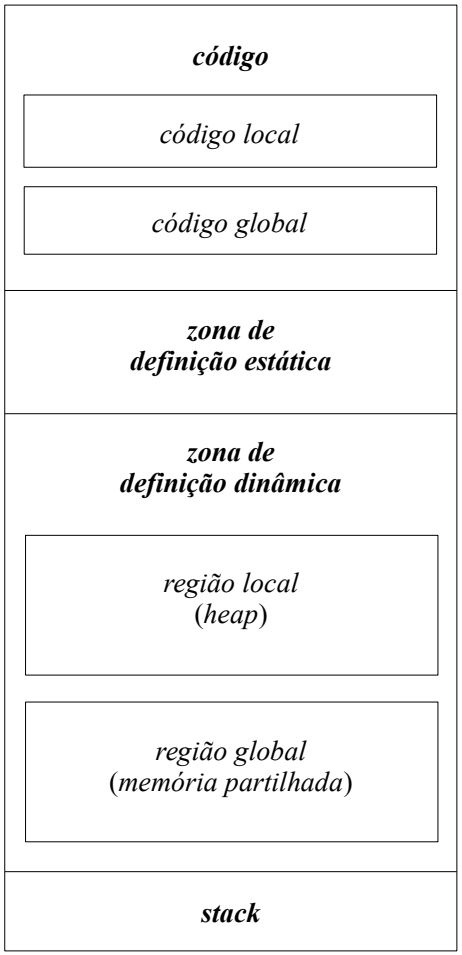
Espaço de endereçamento

```
RELOCATION RECORDS FOR [.text]:
OFFSET   TYPE              VALUE
00000014 R_386_32          .rodata
00000019 R_386_PC32        printf
00000026 R_386_PC32        exit

Contents of section .rodata:
 0000 68656c6c 6f20776f 726c6421 0a000000  hello world!....

Contents of section .comment:
 0000 00474343 3a202847 4e552920 332e332e  .GCC: (GNU) 3.3.
 0010 3120284d 616e6472 616b6520 4c696e75  1 (Mandrake Linu
 0020 7820392e 3220332e 332e312d 326d646b  x 9.2 3.3.1-2mdk
 0030 2900                                 ).
$
```

Espaço de endereçamento

```
$ file hello
hello: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux
       2.2.5, dynamically linked (uses shared libs), not stripped

$ objdump -fTR hello
hello:     file format elf32-i386
architecture: i386, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x080482d0
DYNAMIC SYMBOL TABLE:
0804829c      DF *UND*  000000e6  GLIBC_2.0   __libc_start_main
080482ac      DF *UND*  0000002d  GLIBC_2.0   printf
080482bc      DF *UND*  000000c8  GLIBC_2.0   exit
080484b4 g    DO .rodata        00000004  Base        _IO_stdin_used
00000000  w   D  *UND*  00000000              __gmon_start__
DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080495cc R_386_GLOB_DAT    __gmon_start__
080495c0 R_386_JUMP_SLOT   __libc_start_main
080495c4 R_386_JUMP_SLOT   printf
080495c8 R_386_JUMP_SLOT   exit
$
```

Espaço de endereçamento



- Embora as regiões de *código* e da *zona de definição estática* tenham um tamanho fixo, que é determinado pelo loader, a *zona de definição dinâmica* e a *stack* têm potencialmente um crescimento mais ou menos acentuado durante a execução do processo
- É prática corrente deixar uma área de memória não atribuída no espaço de endereçamento do processo entre a *zona de definição dinâmica* e o *stack* que pode ser usada alternativamente por qualquer deles
- Quando esta área se esgota pelo lado do *stack*, a execução do processo não pode continuar, traduzindo-se na ocorrência de um erro fatal: *stack overflow*

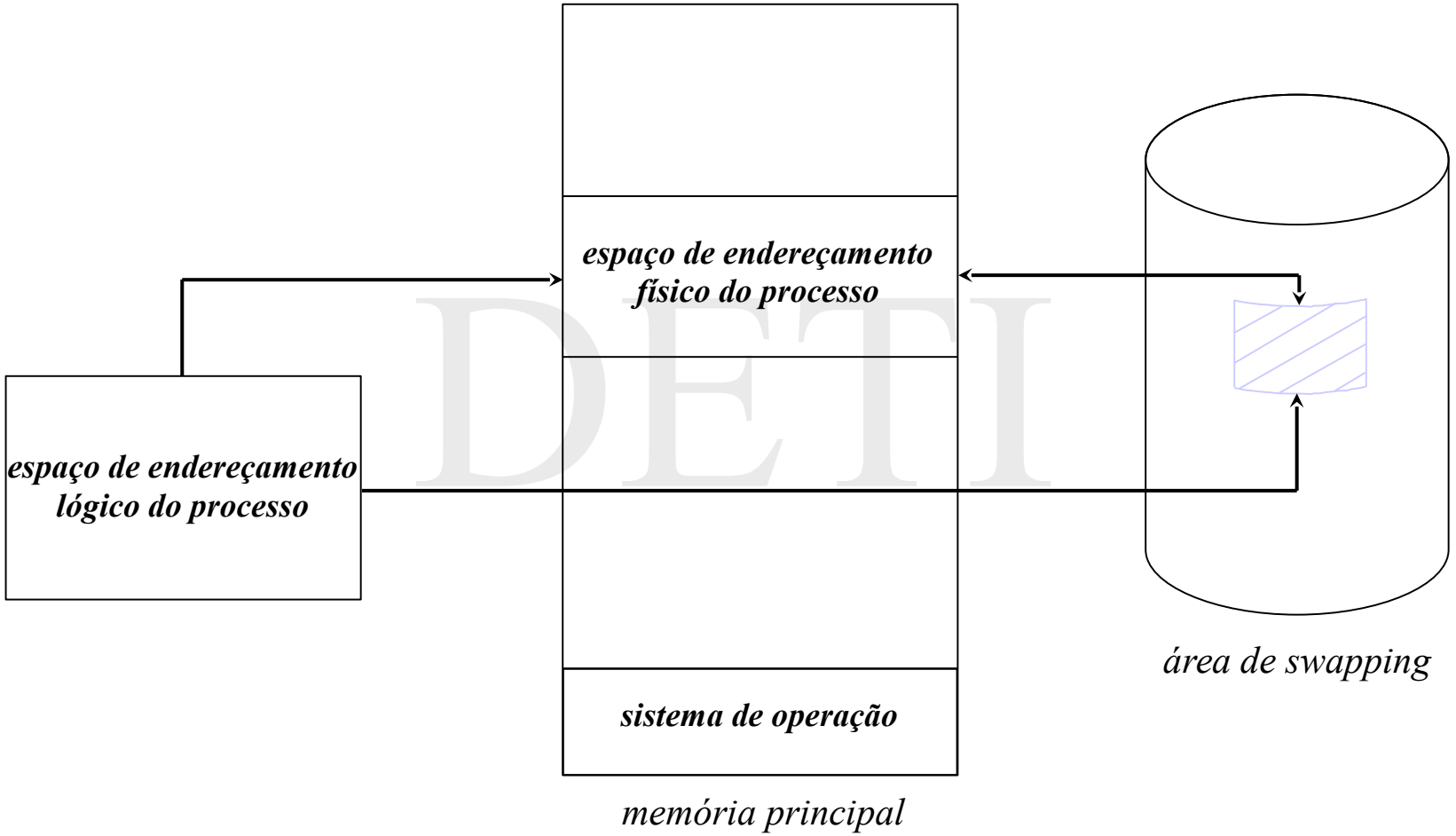
Espaço de endereçamento

- A *imagem binária do espaço de endereçamento do processo* representa um espaço de endereçamento relocatável, o chamado *espaço de endereçamento lógico*, enquanto a região de memória principal onde ele é carregado para execução, constitui o *espaço de endereçamento físico* do processo;
- A separação entre os *espaços de endereçamento lógico e físico* é um conceito central aos mecanismos de gestão de memória num ambiente multiprogramado; há dois problemas que têm que ser resolvidos:
  - *mapeamento dinâmica* – a capacidade de conversão em *run time* de um endereço lógico num endereço físico, de modo a que o espaço de endereçamento físico de um processo possa estar alojado em qualquer região da memória principal e ser deslocado se necessário
  - *protecção dinâmica* – o impedimento em *run time* de referências a endereços localizados fora do espaço de endereçamento próprio do processo

Organização de memória real

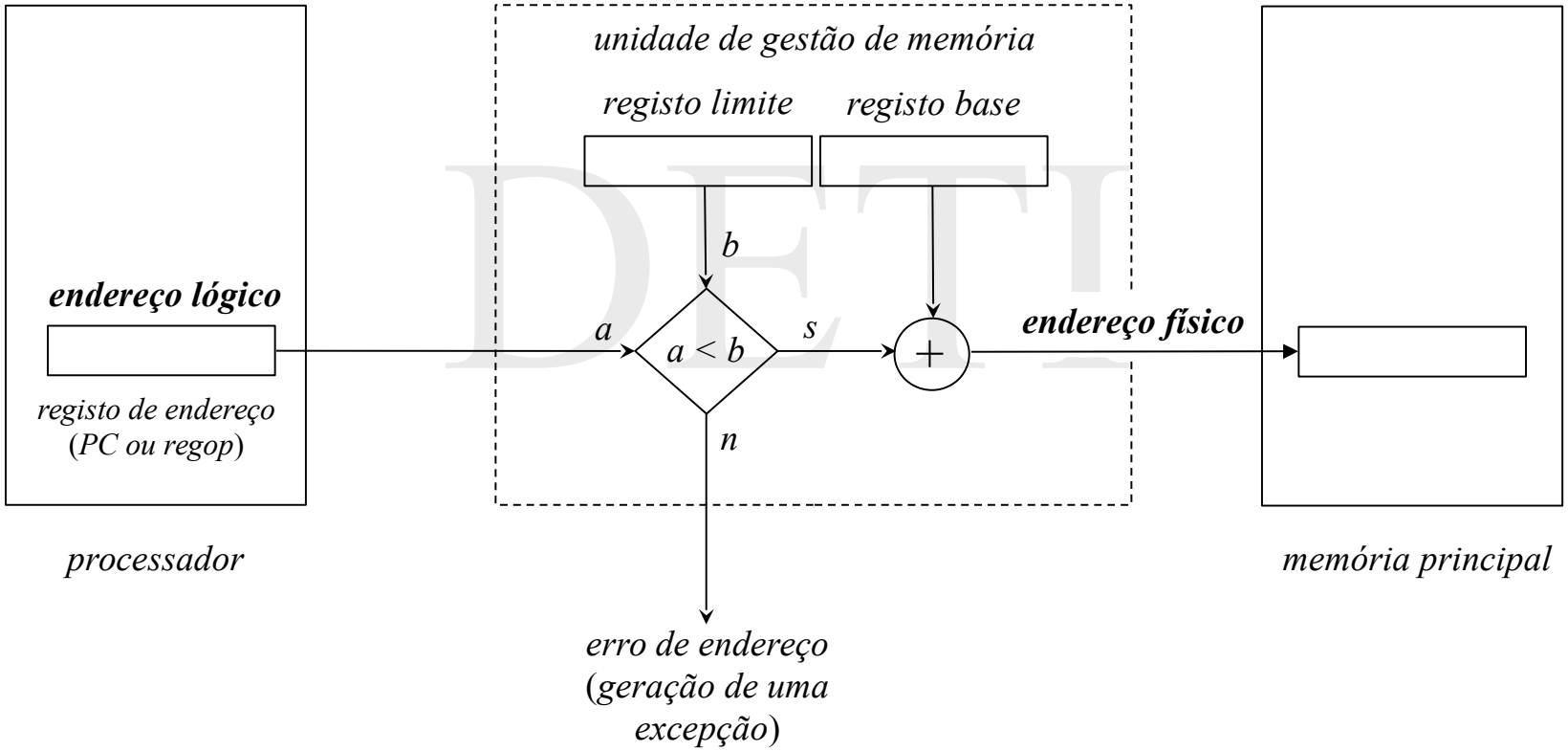
- Numa *organização de memória real* existe uma correspondência biunívoca (de um para um) entre o *espaço de endereçamento lógico* de um processo e o seu *espaço de endereçamento físico*. Consequências:
  - *limitação do espaço de endereçamento de um processo* - em nenhum caso a gestão de memória pode suportar mecanismos automáticos que permitam que o espaço de endereçamento de um processo seja superior ao tamanho da memória principal disponível
  - *contiguidade do espaço de endereçamento físico* - embora não constitua uma condição estritamente necessária, é naturalmente mais simples e eficiente supor que o espaço de endereçamento do processo é contíguo
  - *área de swapping* - o seu papel, enquanto extensão da memória principal, é servir de *arrecadação* para armazenamento do espaço de endereçamento dos processos que não podem ser diretamente carregados em memória principal por falta de espaço.

Organização de memória real



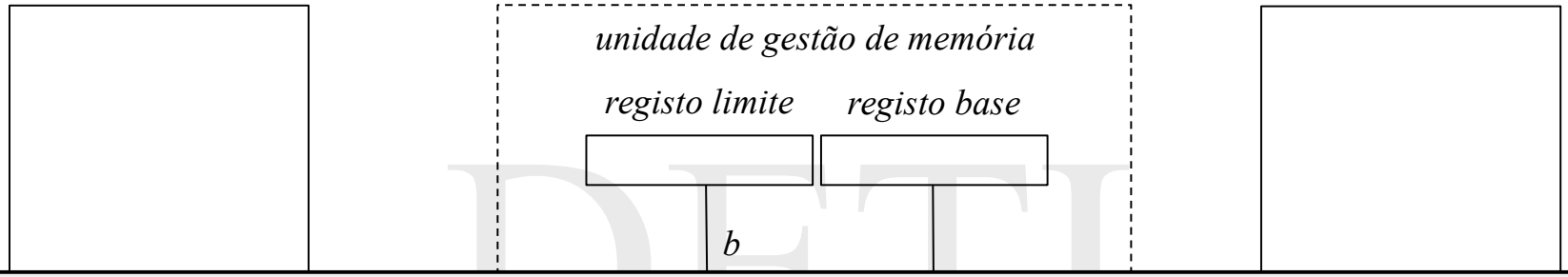
Organização de memória real

Tradução de um endereço lógico num endereço físico



Organização de memória real

Tradução de um endereço lógico num endereço físico

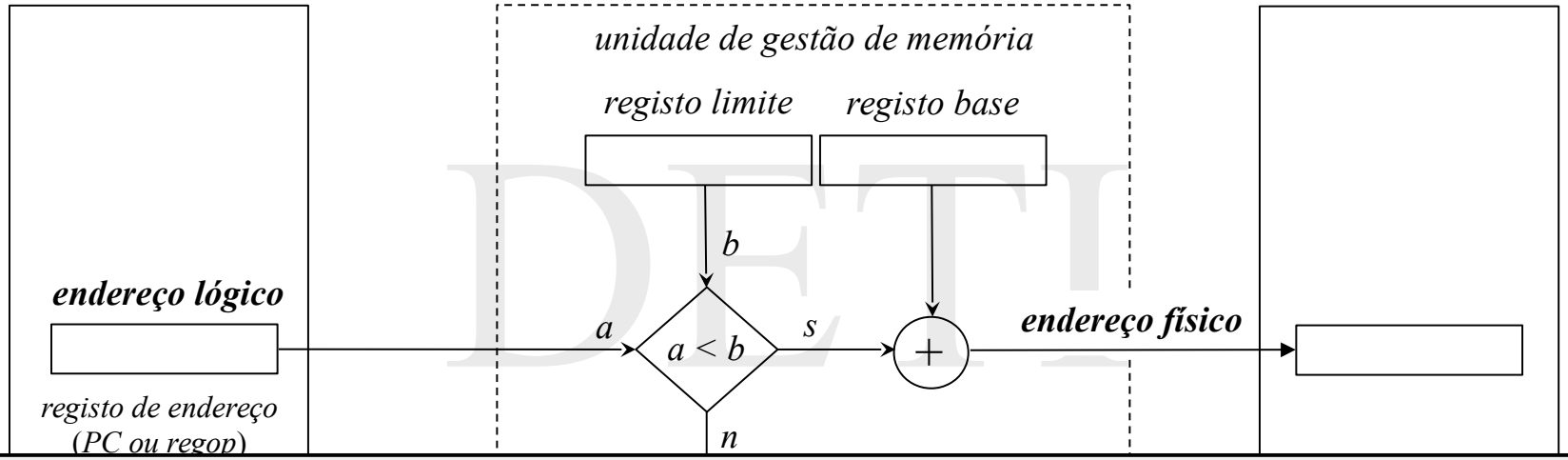


- O *registo base* representa o endereço do início da região de memória principal onde está alojado o *espaço de endereçamento físico* do processo
- O *registo limite* indica o tamanho em *bytes* do espaço de endereçamento
- Na comutação de processos, a operação *dispatch* carrega o *registo base* e o *registo limite* com os valores presentes nos campos correspondentes da entrada da *tabela de controlo de processos* associada com o processo que vai ser calendarizado para execução

(*geração de uma excepção*)

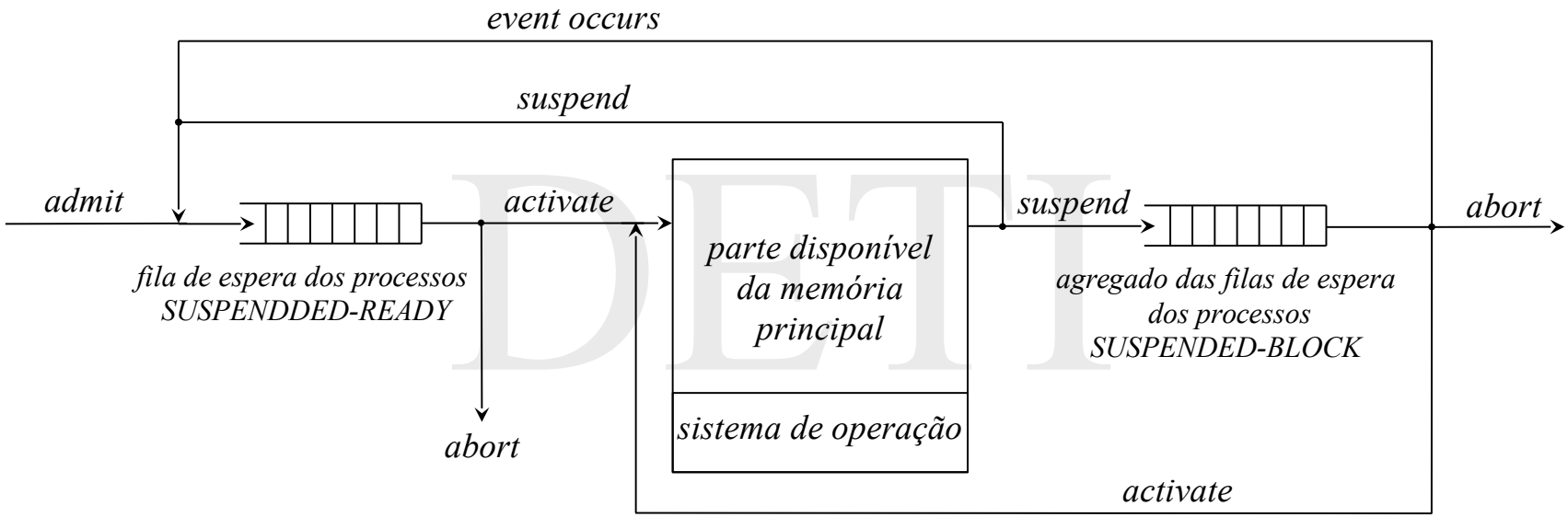
Organização de memória real

Tradução de um endereço lógico num endereço físico



- Sempre que há uma referência à memória, o endereço lógico é primeiro comparado com o conteúdo do registo limite
  - se for menor, trata-se de uma referência válida, ocorre dentro do espaço de endereçamento do processo, e o conteúdo do registo base é adicionado ao endereço lógico para produzir o endereço físico
  - se, ao contrário, for maior ou igual, trata-se de uma referência inválida, um acesso à memória nulo (dummy cycle) é posto em marcha e gera-se uma excepção por erro de endereço

Organização de memória real



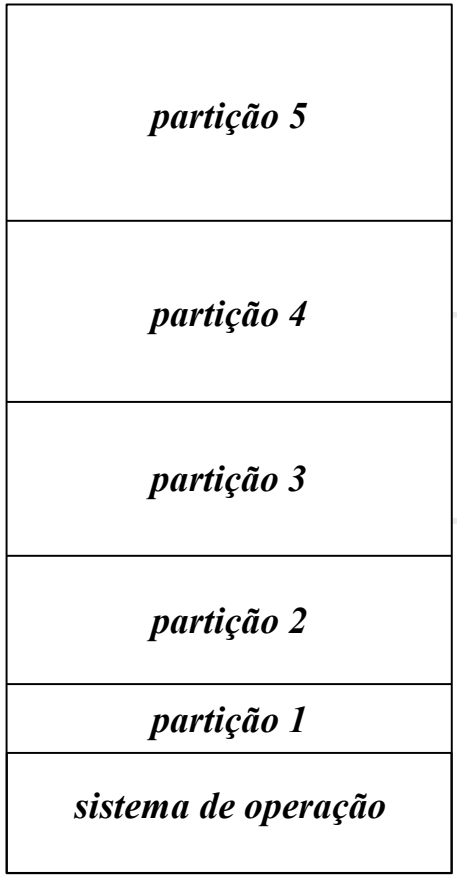
Organização de memória real

- A *parte disponível da memória principal* (a que resta depois de carregado o núcleo do sistema de operação) vai conter os espaços de endereçamento de diferentes processos
- Quando um processo é criado, é posto no estado *CREATED*, sendo inicializadas as estruturas de dados destinadas a geri-lo
- A *imagem binária do seu espaço de endereçamento* é construída, o valor do campo *registo limite* da entrada da tabela de controlo de processos correspondente é determinado e
  - se houver espaço em memória principal, o seu espaço de endereçamento é carregado aí, o campo *registo base* é atualizado com o endereço inicial da região reservada e o processo é colocado na fila de espera dos processos *READY-TO-RUN*
  - caso contrário, o seu espaço de endereçamento é armazenado temporariamente na *área de swapping* e o processo é colocado na fila de espera dos processos *SUSPENDED-READY*

Organização de memória real

- Ao longo da sua execução, o espaço de endereçamento de um processo pode ser deslocado temporariamente para a *área de swapping*, passando o seu estado de *READY-TO-RUN* para *SUSPENDED-READY*, ou de *BLOCKED* para *SUSPENDED-BLOCK*
- Sempre que há espaço na memória, um dos processos presentes na fila de espera dos *processos SUSPENDED-READY* é selecionado, o seu espaço de endereçamento é carregado, o campo *registo base* da entrada da tabela de controlo de processos é atualizado com o endereço inicial da região reservada e o processo é colocado na fila de espera dos processos *READY-TO-RUN*;
- Episodicamente, se esta lista de espera estiver vazia e houver processos na fila de espera dos *processos SUSPENDED-BLOCK*, um deles pode também ser selecionado; o mecanismo é semelhante ao descrito no ponto anterior, só que o processo é colocado agora na fila de espera dos processos *BLOCKED*;
- Finalmente, quando um processo termina, passa para o estado *TERMINATED* e o seu espaço de endereçamento é transferido para a *área de swapping*, se não estiver já lá, aguardando o fim das operações.

Arquitetura de partições fixas



- A *parte disponível da memória principal* é dividida conceptualmente num conjunto fixo de partições mutuamente exclusivas, não necessariamente iguais;
- Cada uma delas vai conter o espaço de endereçamento fisíco de um processo;

Arquitetura de partições fixas

- Diferentes disciplinas de *escalonamento* podem ser usadas na seleção do próximo processo cujo espaço de endereçamento vai ser carregado em memória principal:
  - *Valorizando o critério de justiça* - escolher o primeiro processo da fila de espera dos processos *SUSPENDED-READY* cujo espaço de endereçamento cabe na partição
  - *Valorizando a ocupação da memória principal* - escolher o primeiro processo da fila de espera dos processos *SUSPENDED-READY* com o espaço de endereçamento de tamanho maior que cabe na partição
- Quando a segunda disciplina é aplicada, para se evitar o adiamento indefinido de processos com espaço de endereçamento pequeno, é comum associar um contador a cada processo na lista de espera e incrementá-lo sempre que for ultrapassado na seleção; ao atingir-se um valor pré-definido, o processo já não pode ser descartado e a primeira regra é aplicada.

Arquitetura de partições fixas

Vantagens

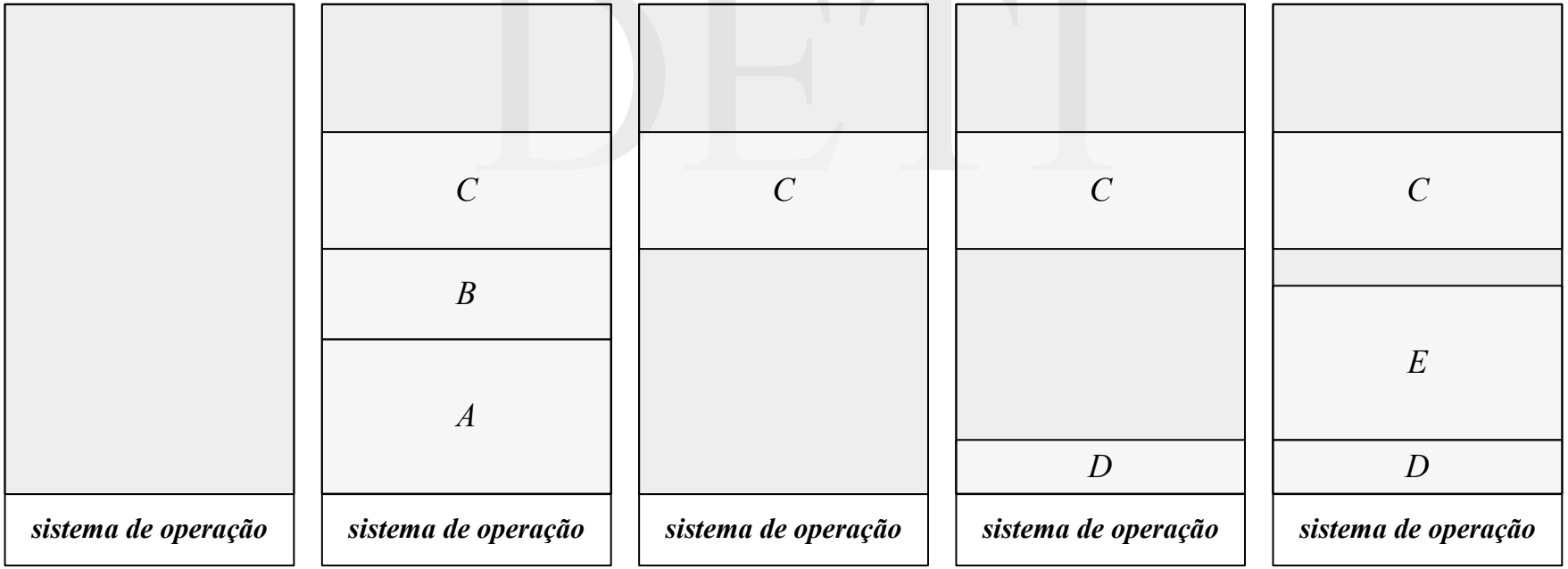
- *simples de implementar* - não exige hardware ou estruturas de dados especiais
- *eficiente* - a seleção pode ser feita muito rapidamente.

Desvantagens

- *conduz a uma grande fragmentação interna da memória principal* - a parte de cada partição que não contém espaço de endereçamento de um processo é desperdiçada, sendo usada para nada
- *de aplicação específica* - a única maneira de se minimizar o desperdício de memória é adequar o tamanho das partições ao tipo de processos (número e tamanho do seu espaço de endereçamento) que vão ser executados, tornando-se por isso muito pouco geral

Arquitetura de partições variáveis

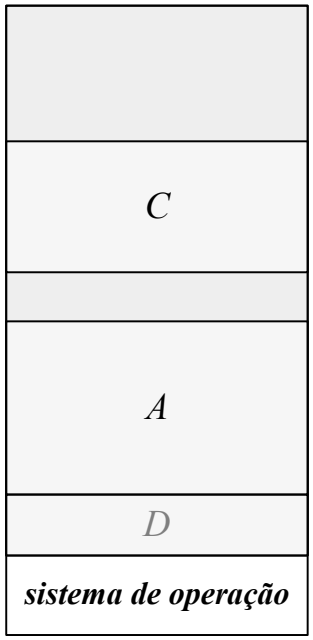
- Uma maneira de aumentar a taxa de ocupação da memória principal é supor que toda a *parte disponível da memória* constitui à partida um bloco único, ir sucessivamente reservando regiões de tamanho suficiente para carregar o espaço de endereçamento dos processos que vão surgindo, e ir mais tarde libertando-as, quando deixarem de ser necessárias.

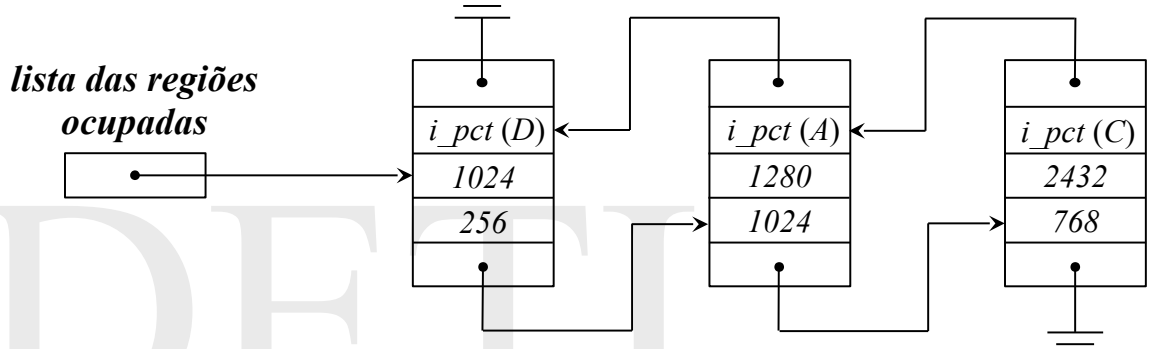


Arquitetura de partições variáveis

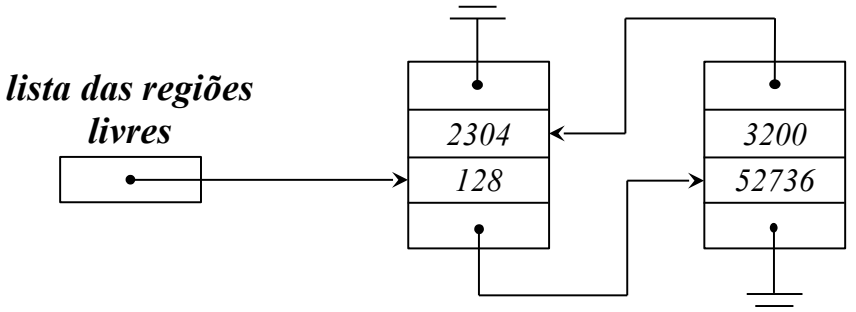
- Como a memória é reservada dinamicamente, o sistema de operação tem que manter um registo atualizado das regiões ocupadas e das regiões livres
- Uma maneira de fazer isso é construindo duas listas (bi)ligadas
  - *lista das regiões ocupadas* – localiza as regiões que foram reservadas para armazenamento do espaço de endereçamento dos processos residentes em memória principal
  - *lista das regiões livres* – localiza as regiões ainda disponíveis para armazenamento do espaço de endereçamento dos novos processos que vão sendo criados, ou que venham a ser transferidos da *área de swapping*
- Se a região de memória reservada fosse exactamente a suficiente para armazenamento do espaço de endereçamento do processo, corria-se o risco de formar regiões livres de tamanho tão pequeno que nunca poderiam ser utilizadas por si próprias, mas que seriam incluídas na lista das regiões livres, tornando pesquisas posteriores mais complexas
- Assim, a memória principal é dividida tipicamente em blocos de tamanho fixo e a reserva de espaço é feita em unidades de tamanho desses blocos

Arquitetura de partições variáveis





	Tamanho (bytes)
Memória Principal	256 M
Sistema de Operação	4M
Unidade de reserva	4K
Processo A	4M
Processo C	3M
Processo D	1M



Arquitetura de partições variáveis

- A *valorização do critério de justiça* é a disciplina de *escalonamento* geralmente adotada, sendo escolhido o primeiro processo da fila de espera dos processos *SUSPENDED-READY* cujo espaço de endereçamento pode ser colocado em memória principal
- O principal problema que se põe numa arquitetura de partições variáveis, tem a ver com o grau de *fragmentação externa* que é produzido na memória principal pelas sucessivas reserva e libertação das regiões de armazenamento do espaço de endereçamento dos processos
- Pode atingir-se situações em que, embora haja memória livre em quantidade suficiente, ela não é contínua e, por isso, o armazenamento do espaço de endereçamento de um novo processo deixa de ser possível
- A solução é efetuar a compactação do espaço livre, *garbage collection*, agrupando todas as regiões livres num dos extremos da memória
  - Esta operação, no entanto, exige a paragem de todo o processamento e, se a memória for grande, tem um tempo de execução muito elevado

Arquitetura de partições variáveis

- É importante desenvolver-se métodos eficientes para a localização da região de memória principal que deve ser reservada para armazenamento do espaço de endereçamento de um processo; entre eles destacam-se
  - *first fit* – a lista das regiões livres é pesquisada desde o princípio até se encontrar a primeira região com tamanho suficiente;
  - *next fit* – é uma variante do *first fit* que consiste em iniciar a pesquisa a partir do ponto de paragem na pesquisa anterior;
  - *best fit* – a lista das regiões livres é pesquisada na sua totalidade, escolhendo-se a região mais pequena de tamanho maior ou igual do que o espaço de endereçamento do processo;
  - *worst fit* – a lista das regiões livres é pesquisada na sua totalidade, escolhendo-se a maior região existente.

- ***Procure avaliar o desempenho dos diferentes métodos em termos do grau e do tipo de fragmentação produzidos e da eficiência na reserva e na libertação de espaço!***

Arquitetura de partições variáveis

Vantagens

- *geral* – o âmbito da sua aplicação é independente do tipo de processos que vão ser executados (número e, dentro de certos limites, tamanho do seu espaço de endereçamento)
- *implementação de pequena complexidade* – não exige hardware especial e as estruturas de dados reduzem-se a duas listas (bi)ligadas

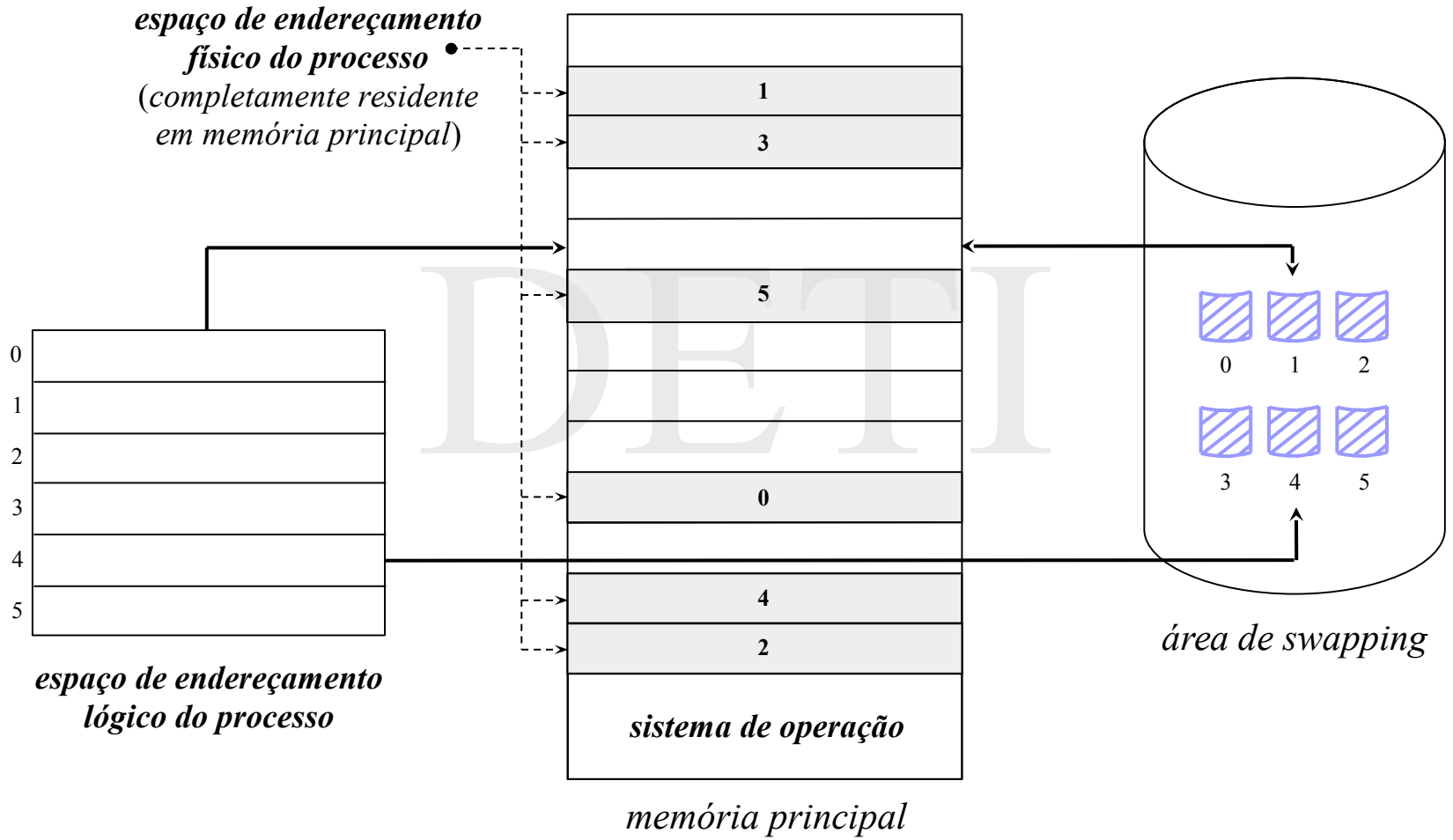
Desvantagens

- *conduz a uma grande fragmentação externa da memória principal* – a fração da memória principal que acaba por ser desperdiçada, face ao tamanho reduzido das regiões em que está dividida, pode atingir em alguns casos cerca de um terço do total (*regra dos 50%*)
- *pouco eficiente* – não é possível construir algoritmos que sejam simultaneamente muito eficientes na reserva e na libertação de espaço

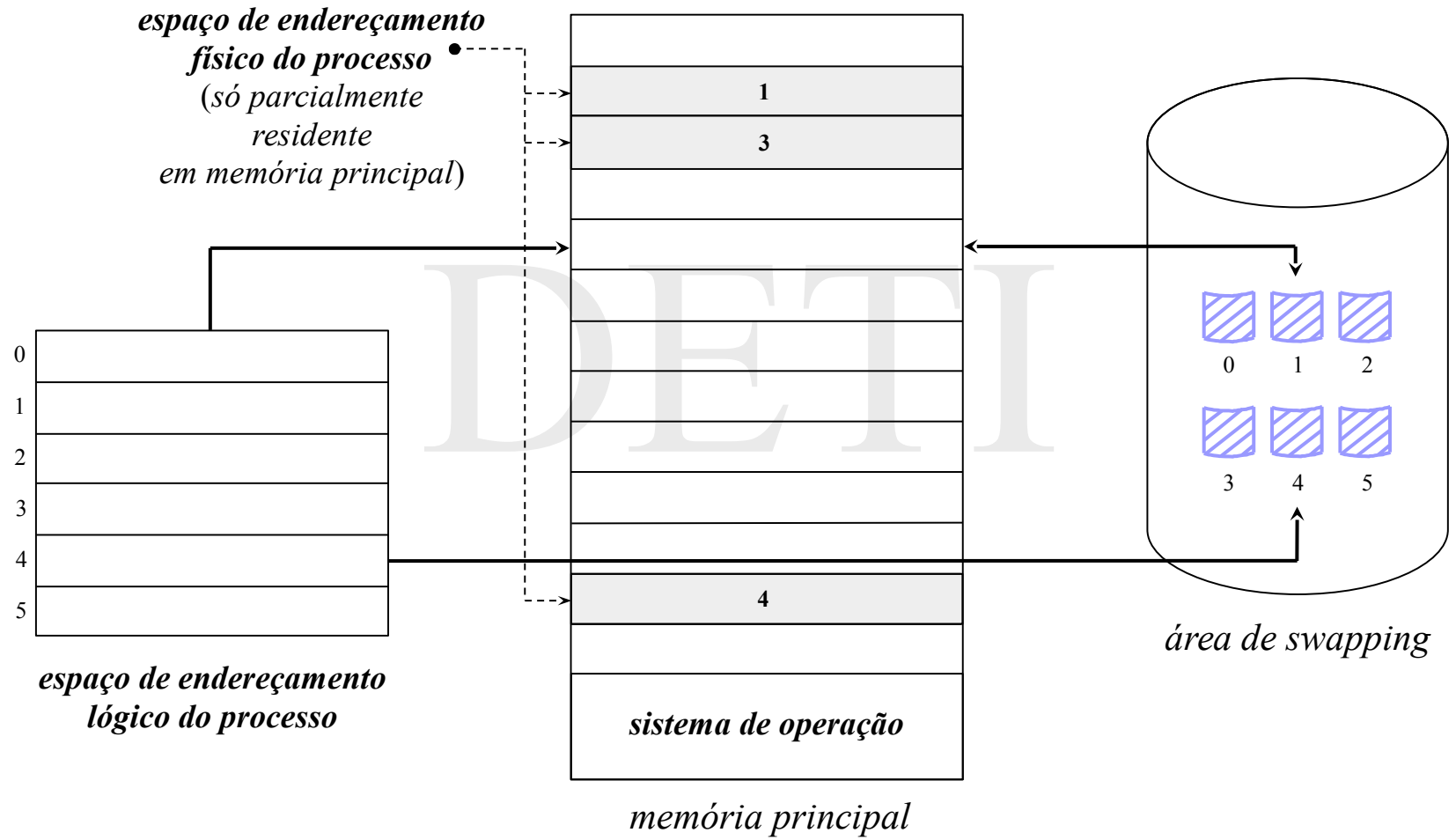
Organização de memória virtual

- Numa *organização de memória virtual* o *espaço de endereçamento lógico* de um processo e o seu *espaço de endereçamento físico* estão totalmente dissociados
- As consequências desta política são de três tipos
  - *não limitação do espaço de endereçamento de um processo* - pode estabelecer-se uma metodologia conducente à execução de processos cujo espaço de endereçamento é superior ao tamanho da memória principal disponível
  - *não contiguidade do espaço de endereçamento físico* - os espaços de endereçamento dos processos, divididos em blocos de tamanho fixo ou variável, estão dispersos por toda a memória, procurando-se desta maneira garantir uma ocupação mais eficiente do espaço disponível
  - *área de swapping* - o seu papel, enquanto extensão da memória principal, é servir para manter uma imagem atualizada dos espaços de endereçamento dos processos que correntemente coexistem, nomeadamente da sua parte variável (zonas de definição estática e dinâmica e *stack*)

Organização de memória virtual

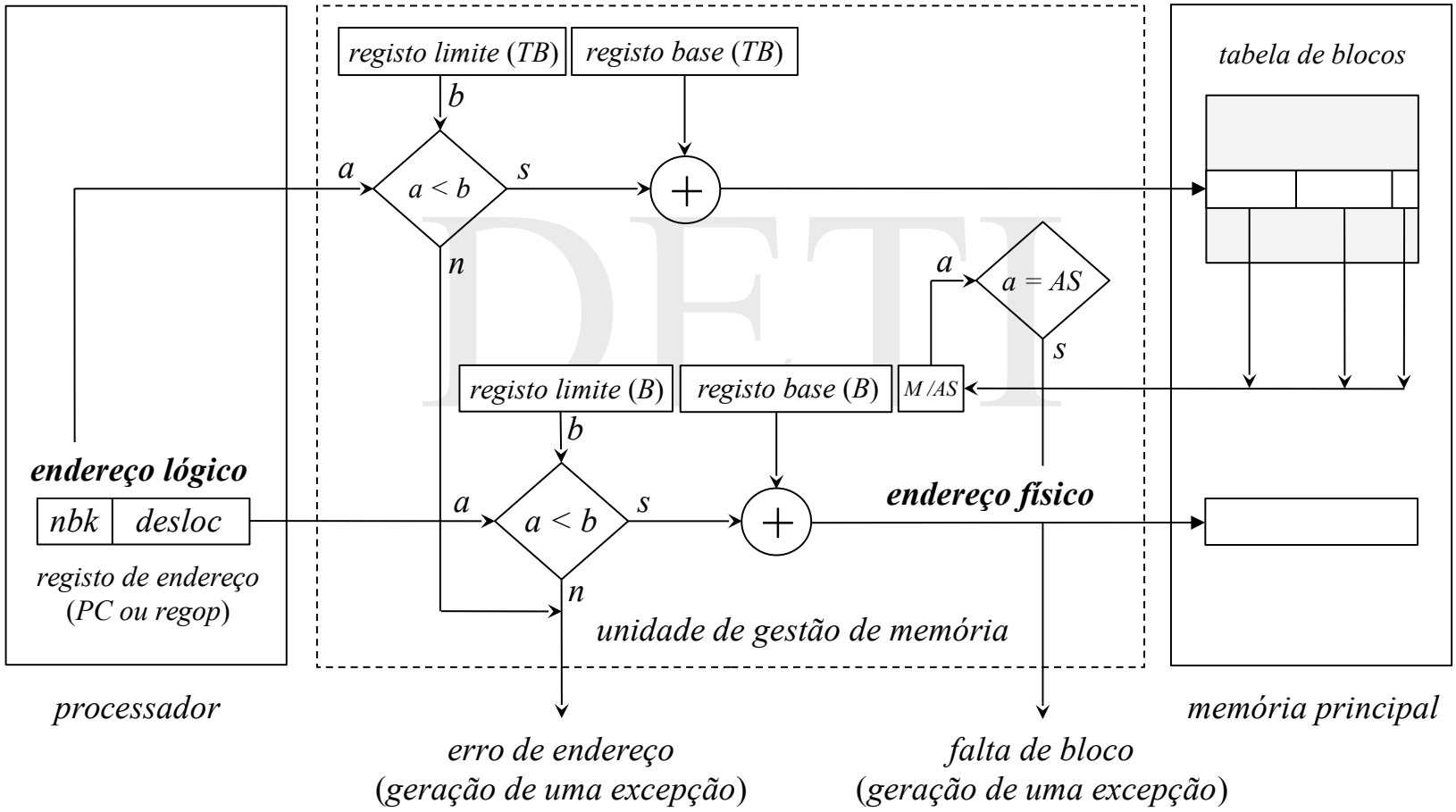


Organização de memória virtual



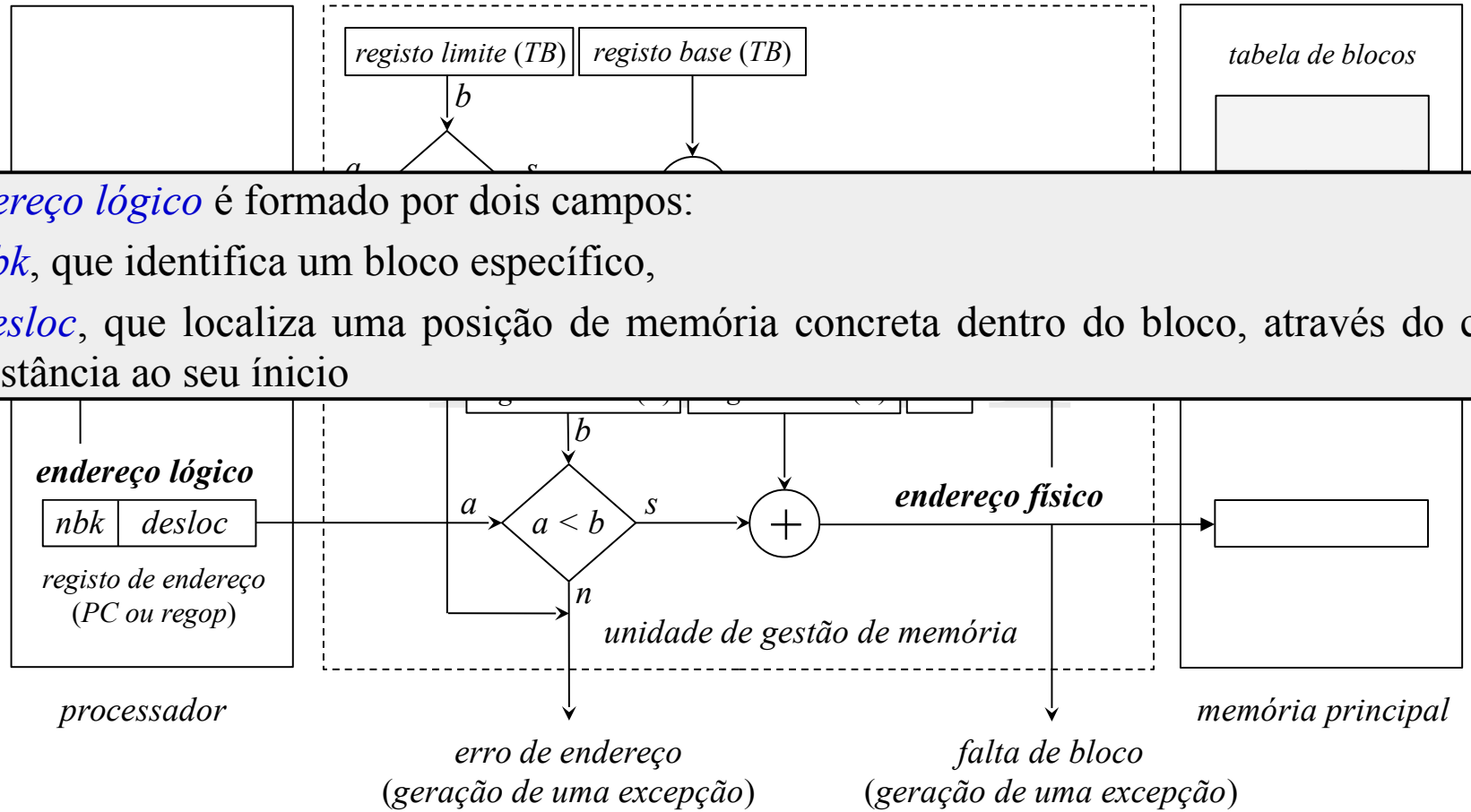
Organização de memória real

Tradução de um endereço lógico num endereço físico



Organização de memória virtual

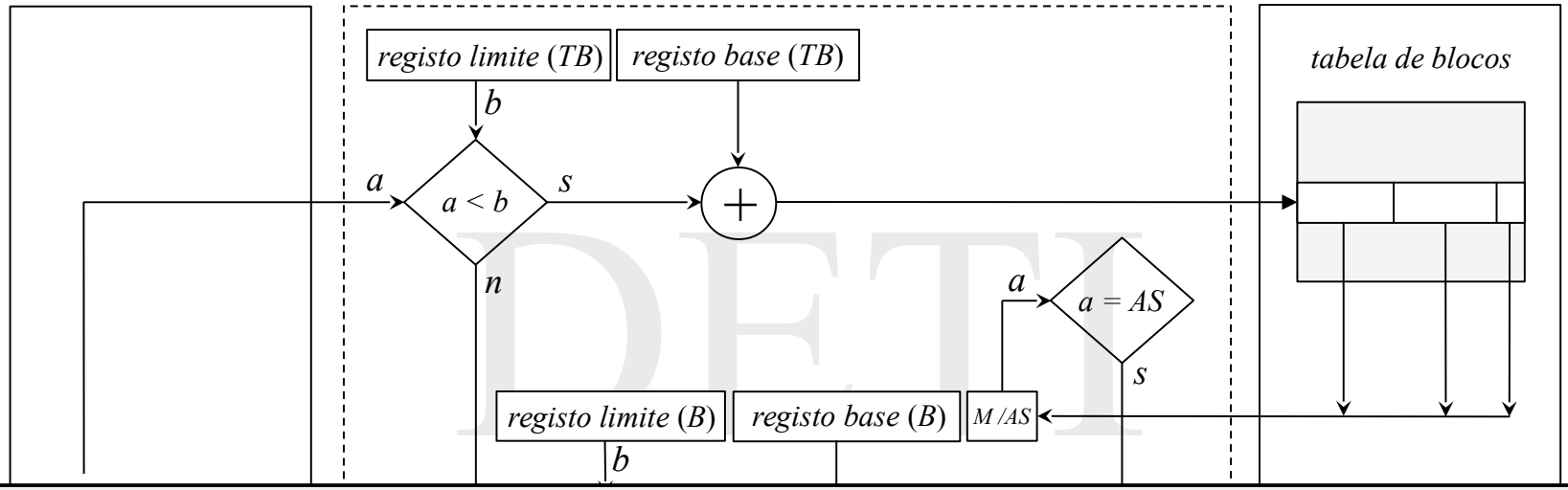
Tradução de um endereço lógico num endereço físico



- o *endereço lógico* é formado por dois campos:
  - *nbk*, que identifica um bloco específico,
  - *desloc*, que localiza uma posição de memória concreta dentro do bloco, através do cálculo da distância ao seu ínicio

Organização de memória virtual

Tradução de um endereço lógico num endereço físico

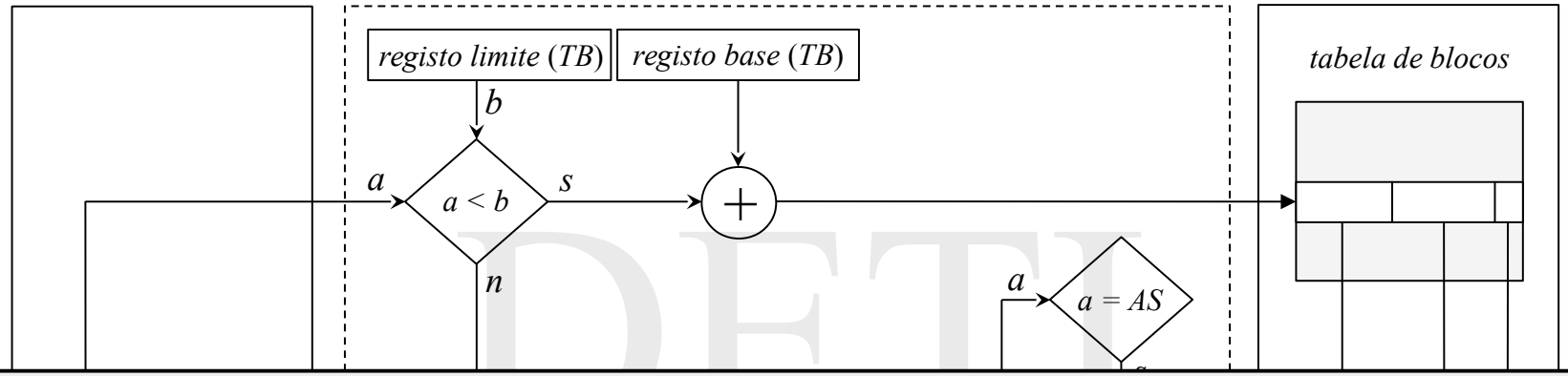


- A unidade de gestão de memória contém agora dois pares de registos *base e limite*
  - um está associado com a *tabela de blocos* do processo, estrutura de dados que fornece a descrição da localização dos vários blocos em que o espaço de endereçamento do processo está dividido
  - o outro está associado com a descrição de um bloco particular, cuja referência está a decorrer num dado momento

(*geração de uma excepção*) (*geração de uma excepção*)

Organização de memória virtual

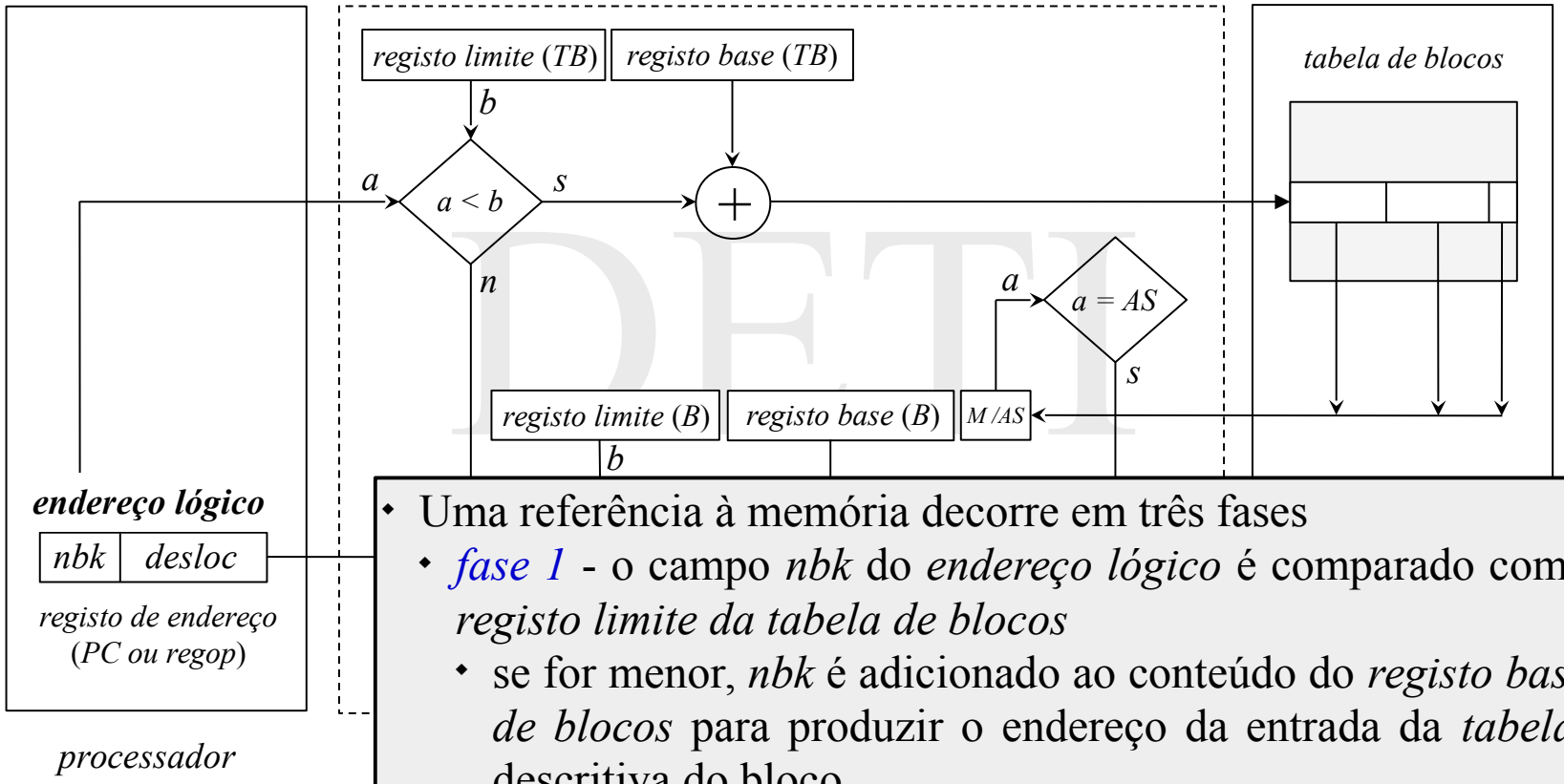
Tradução de um endereço lógico num endereço físico



- Quando ocorre uma comutação de processos, a operação *dispatch* carrega o *registo base* e o *registo limite da tabela de blocos* com os valores presentes nos campos correspondentes da entrada da *tabela de controlo de processos* associada com o processo que vai ser calendarizado para execução
  - o valor do *registo base da tabela de blocos* representa o endereço do início da região de memória principal onde está alojada a *tabela de blocos* do processo
  - o valor do *registo limite* está relacionado com o número de entradas da tabela

Organização de memória virtual

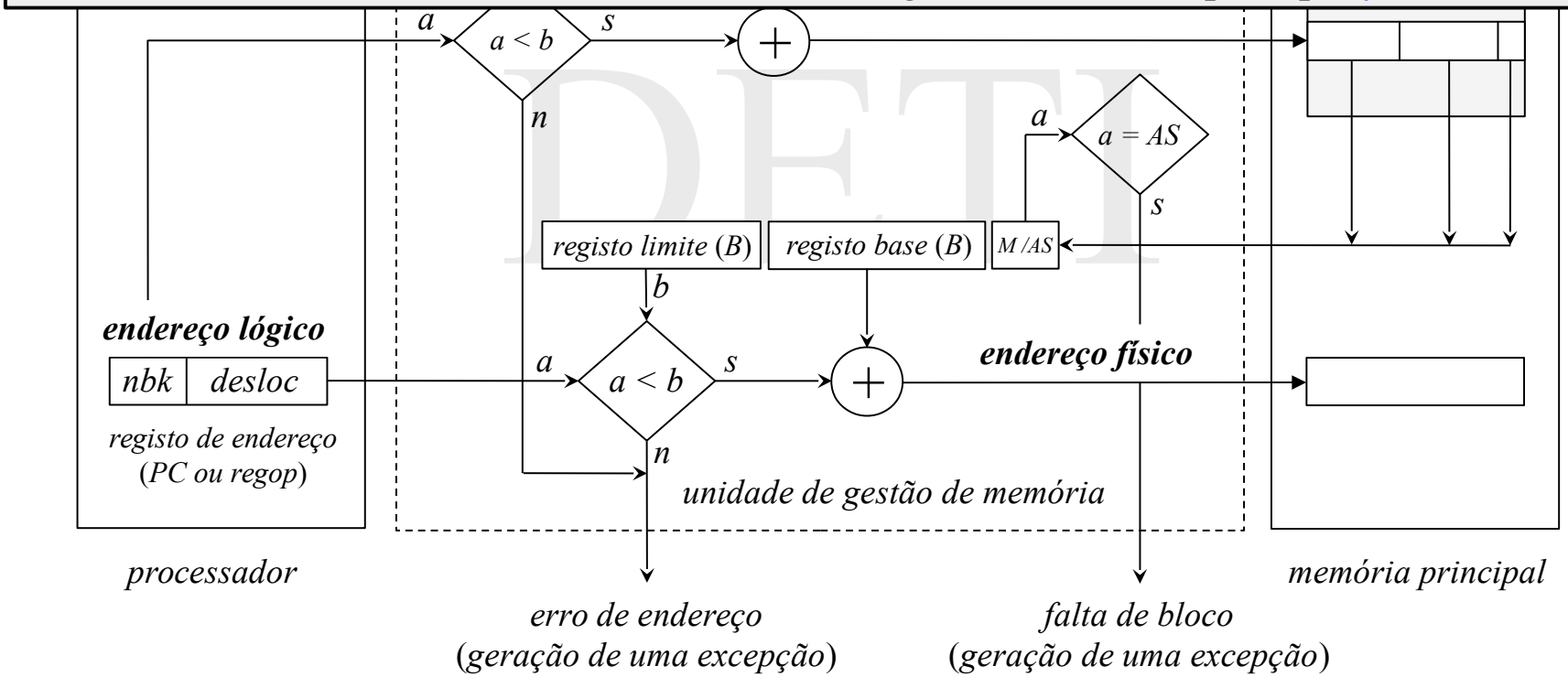
Tradução de um endereço lógico num endereço físico



- Uma referência à memória decorre em três fases
  - *fase 1* - o campo *nbk* do *endereço lógico* é comparado com o valor do *registo limite da tabela de blocos*
    - se for menor, *nbk* é adicionado ao conteúdo do *registo base da tabela de blocos* para produzir o endereço da entrada da *tabela de blocos* descritiva do bloco
    - se, for maior ou igual, a referência é inválida, um acesso à memória nulo (*dummy cycle*) é posto em marcha e gera-se uma excepção por *erro de endereço*

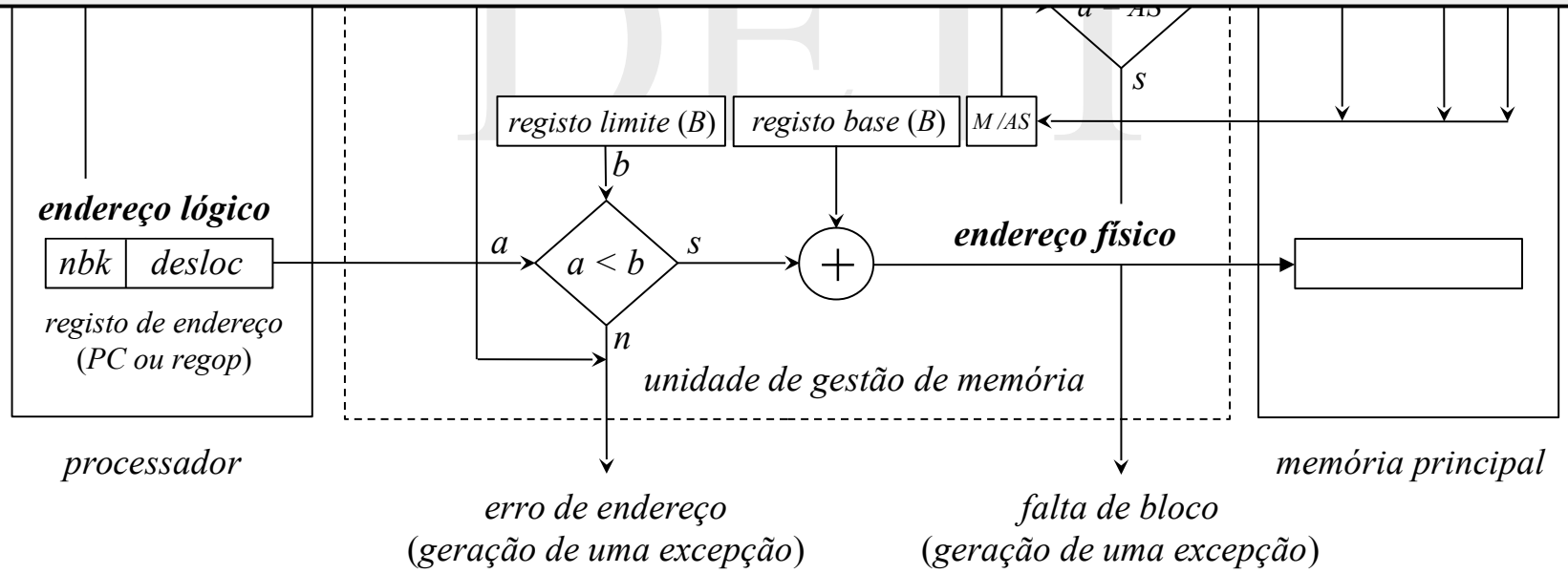
Organização de memória virtual

- *fase 2* – avalia-se o valor do registo *M/AS*:
  - se for *M*, os campos da entrada da *tabela de blocos* referenciada são transferidos para os registos respetivos da unidade de gestão de memória
  - se for *AS*, o bloco não está atualmente presente na memória principal, a instrução é finalizada com um acesso à memória nulo e gera-se uma excepção por *falta de bloco*



Organização de memória virtual

- *fase 3* - o campo *desloc* do *endereço lógico* é comparado com o valor do *registo limite do bloco*
  - se for menor, trata-se de uma referência válida, ocorre dentro do espaço de endereçamento do bloco, e *desloc* é adicionado ao conteúdo do *registo base do bloco* para produzir o *endereço físico*
  - se for maior ou igual, trata-se de uma referência inválida, um acesso à memória nulo é posto em marcha e gera-se uma excepção por *erro de endereço*



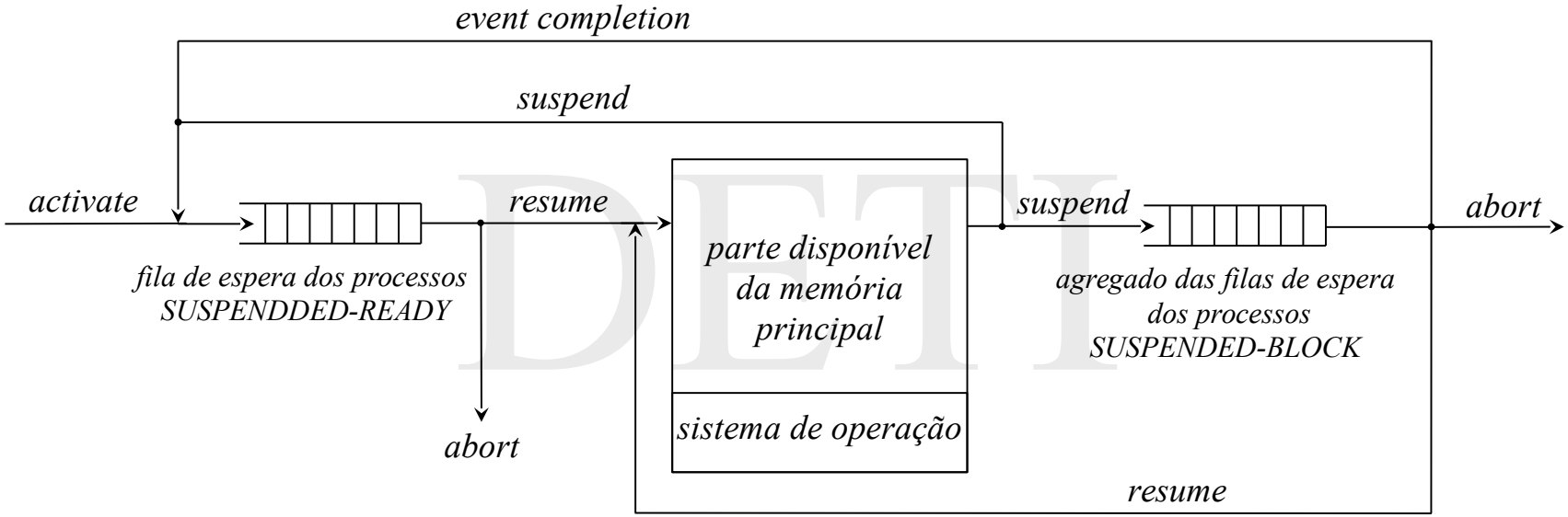
Organização de memória virtual

- O aumento de versatilidade na gestão do espaço em memória principal, introduzido por uma organização de memória virtual, tem um custo que se traduz na transformação de cada acesso à memória em dois acessos:
  - no primeiro é referenciada a entrada da tabela de blocos do processo, associada com o bloco descrito no campo *nbk* do endereço lógico, para se obter o endereço do início do bloco em memória;
  - só no segundo é referenciada a posição de memória específica, sendo o cálculo do seu endereço feito adicionando o campo *desloc* do endereço lógico ao endereço do início do bloco em memória;
- Conceptualmente, a organização de memória virtual resulta num fracionamento do espaço de endereçamento lógico do processo em blocos que são tratados dinamicamente como sub-espaços de endereçamento autónomos numa organização de memória real (*de partições fixas*, se os blocos tiverem todos o mesmo tamanho, *de partições variáveis*, se puderem ter tamanho diferente);
- O que há de novo é a possibilidade de ocorrer um acesso a um bloco atualmente não residente em memória principal, com a consequente necessidade de anulação do acesso e a sua repetição mais tarde, quando ele for carregado

Organização de memória virtual

- A necessidade deste duplo acesso à memória pode ser minimizada tirando partido do *princípio da localidade de referência*;
- Como os acessos tenderão a estar concentrados num conjunto bem definido de blocos durante intervalos de tempo alargados de execução do processo, a unidade de gestão de memória mantém habitualmente armazenado numa memória associativa interna, designada de *translation lookaside buffer* (*TLB*), o conteúdo das entradas da tabela de blocos que foram ultimamente referenciadas
- Deste modo, o primeiro acesso pode ser um:
  - *hit* – quando a entrada está armazenada na *TLB*, caso em que o acesso é interno ao processador
  - *miss* – quando a entrada não está armazenada na *TLB*, caso em que há um acesso externo à memória principal
- O tempo médio de acesso a uma instrução, ou um operando, aproxima-se assim tendencialmente do valor mais baixo (um acesso ao *TLB* + um acesso à memória principal).

Organização de memória virtual



Organização de memória virtual

- A *parte disponível da memória principal* (a que resta depois de carregado o núcleo do sistema de operação) vai conter porções dos espaços de endereçamento de diferentes processos;
- Quando um processo é criado, é posto no estado *CREATED*, sendo inicializadas as estruturas de dados destinadas a geri-lo;
  - A *imagem binária do seu espaço de endereçamento* é construída, sendo transferida para a *área de swapping*, pelo menos, a sua parte variável
  - A *tabela de blocos* associada é organizada a seguir
  - Se houver espaço, a *tabela de blocos*, o primeiro bloco de código do processo e o bloco de *stack* são carregados em memória principal, as entradas correspondentes da *tabela de blocos* são atualizadas com o endereço inicial das regiões reservadas e o processo é colocado na fila de espera dos processos *READY-TO-RUN*
  - Caso contrário, o processo é colocado na fila de espera dos processos *SUSPENDED-READY*

Organização de memória virtual

- Ao longo da execução do processo, sempre que ocorra um acesso a um bloco não residente em memória principal, o processo é posto no estado *BLOCKED* enquanto decorre a transferência do bloco da *área de swapping* para a memória principal; quando esta transferência terminar, o processo é de novo colocado na fila de espera dos processos *READY-TO-RUN*
- Todos os blocos residentes em memória principal, e pertencentes a um mesmo processo, podem ser deslocados temporariamente para a *área de swapping*, passando o seu estado de *READY-TO-RUN* para *SUSPENDED-READY*, ou de *BLOCKED* para *SUSPENDED-BLOCK*;
- Sempre que há espaço na memória, um dos processos presentes na fila de espera dos *processos SUSPENDED-READY* é seleccionado, *a tabela de blocos* e um grupo de blocos do seu espaço de endereçamento são carregados, as entradas correspondentes da *tabela de blocos* são atualizadas com os endereços iniciais das regiões reservadas e o processo é colocado na fila de espera dos processos *READY-TO-RUN*;

Organização de memória virtual

- Se a lista *SUSPENDED-READY* estiver vazia e houver processos na fila de espera dos *processos SUSPENDED-BLOCK*, um deles pode também ser selecionado; o mecanismo é semelhante ao descrito no ponto anterior, só que o processo é agora colocado na fila de espera dos processos *BLOCKED*;
- Finalmente, quando um processo termina, passa para o estado *TERMINATED* e a imagem do seu espaço de endereçamento residente na *área de swapping*, ou pelo menos da sua parte variável, é atualizada com a consequente libertação de todos os blocos existentes em memória principal, aguardando o fim das operações.

Exceção por falta de bloco

- Uma propriedade relevante da *organização de memória virtual* é a capacidade apresentada pelo sistema computacional de execução de processos cujo espaço de endereçamento não está na sua totalidade, e em simultâneo, residente em memória principal.
- Assim, o sistema de operação tem que providencionar meios para resolver o problema de referência a um endereço localizado num bloco que não está atualmente presente em memória.
- Como foi referido atrás, a *unidade de gestão de memória* gera nestas circunstâncias uma exceção por *falta de bloco* e a rotina de serviço à excepção deve pôr em marcha ações que visem a transferência desse bloco da *área de swapping* para a memória principal e, após a sua conclusão, a repetição da execução da instrução que produziu a referência.
- Todas estas operações são realizadas de uma forma completamente transparente ao utilizador que não tem consciência das interrupções introduzidas na execução do processo.

Exceção por falta de bloco

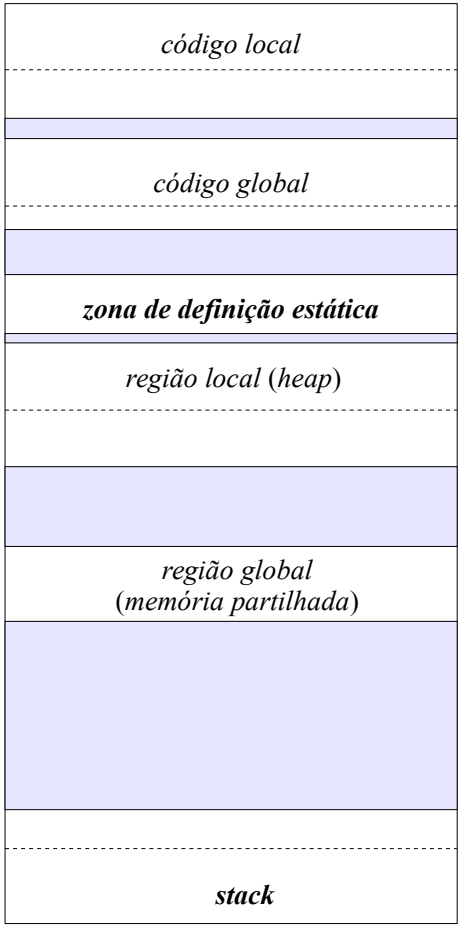
Sequência de operações postas em marcha por uma *falta de bloco*

- salvaguardar o contexto do processo na entrada correspondente da *tabela de controlo de processos*, colocando o seu estado em *BLOCKED* e atualizando o *program counter* para o endereço que produziu a *falta de bloco*;
- determinar se existe espaço em memória principal para carregar o bloco em falta
  - caso exista, selecionar uma região livre;
  - caso não exista, selecionar uma região cujo bloco vai ser substituído:
    - se o bloco tiver sido modificado, proceder à sua transferência para a *área de swapping*;
    - atualizar a entrada da *tabela de blocos* do processo a que o bloco pertence, com a indicação de que o bloco já não está residente em memória;
- transferir o bloco em falta da *área de swapping* para a região seleccionada;
- invocar o *escalonador* para calendarizar para execução um dos processos da fila de espera dos processos *READY-TO-RUN*;
- quando a transferência estiver concluída, atualizar a entrada da *tabela de blocos* do processo com a indicação de que o bloco está residente em memória e de qual é a sua localização, e mudar o seu estado para *READY-TO-RUN*, colocando-o na fila de espera correspondente.

Exceção por falta de bloco

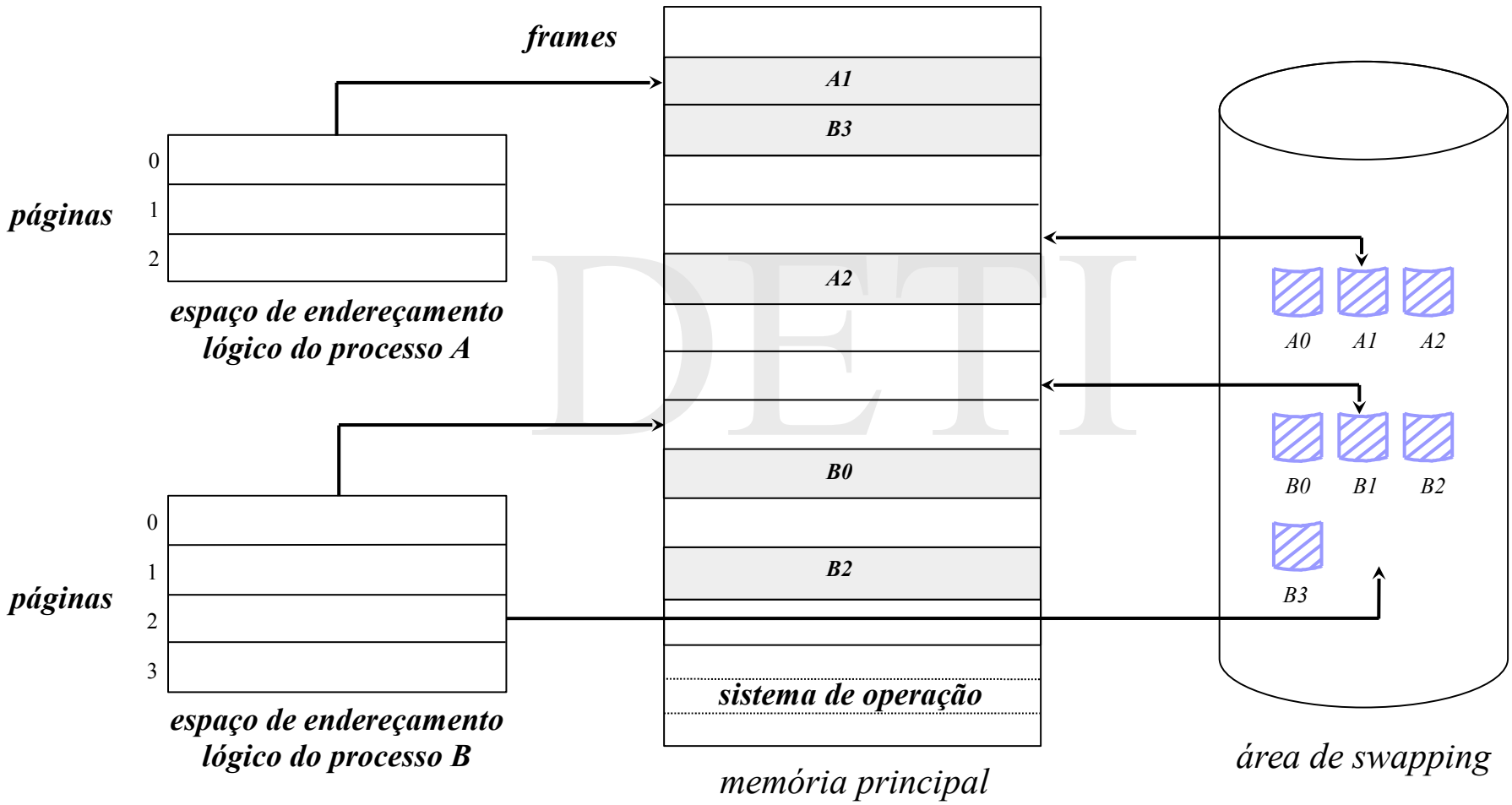
- Se, durante a sua execução, um processo estivesse continuamente a gerar exceções por *falta de bloco*, o ritmo de processamento seria muito lento e, em geral, o *throughput* do sistema computacional seria também muito baixo, pondo em causa a utilidade de uma organização de memória virtual em multiprogramação.
- Na prática, tal não se passa!
- Como foi já referido relativamente à hierarquia de memória de um sistema computacional e à introdução do *translation lookaside buffer* na unidade de gestão de memória, c*onstata-se experimentalmente que um processo, qualquer que seja a fase da sua execução, referencia durante períodos de tempo relativamente alargados apenas uma fração pequena do seu espaço de endereçamento*.
- Esta fração varia naturalmente ao longo do tempo, mas em cada intervalo considerado existem tipicamente milhares de referências que são realizadas sobre frações bem definidas do seu espaço de endereçamento. Desta maneira, desde que os blocos correspondentes estejam presentes em memória principal, o ritmo de execução do processo pode prosseguir sem a ocorrência de *faltas de bloco*.

Arquitetura paginada

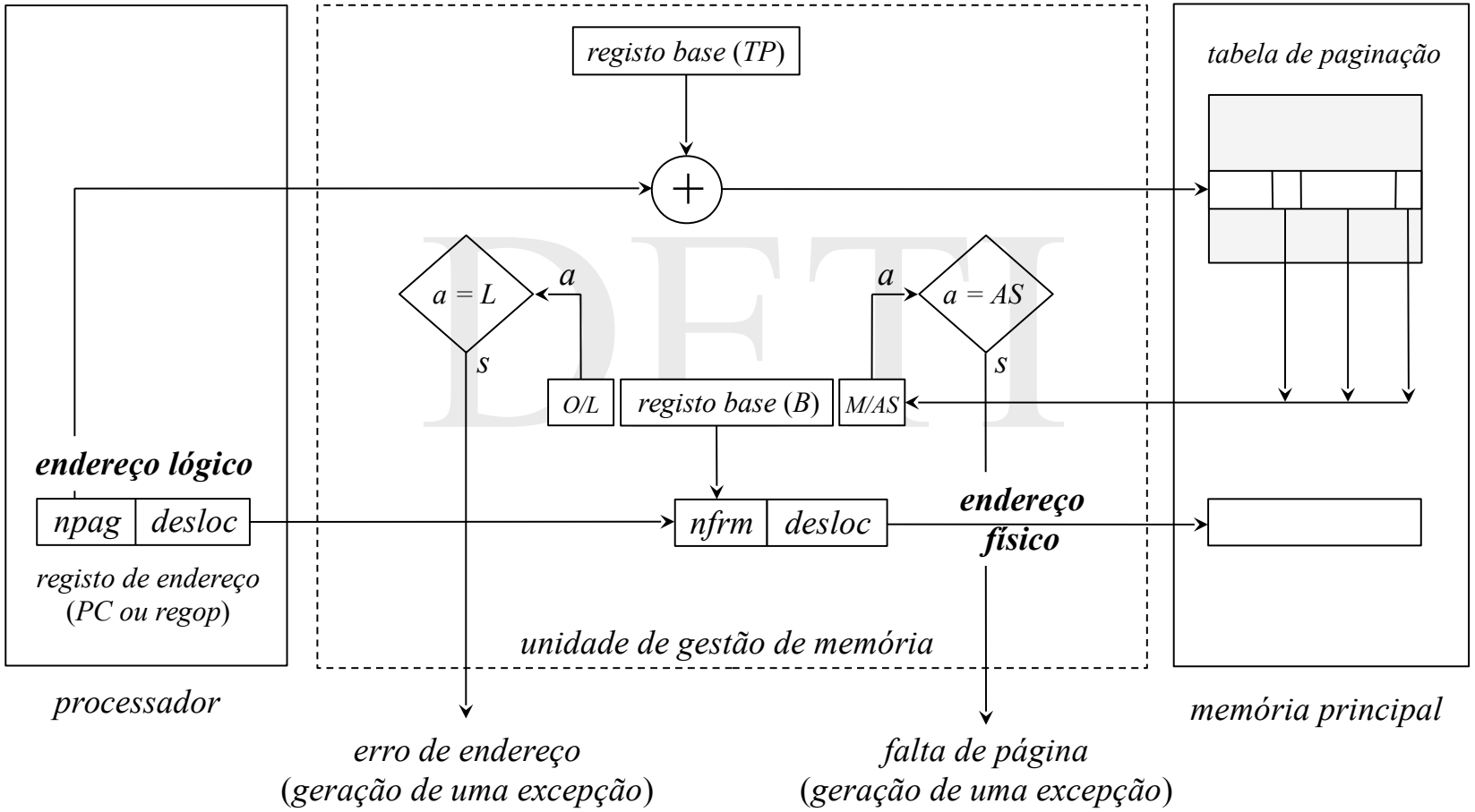


- Os blocos, designados aqui de *páginas*, são todos iguais e têm um tamanho correspondente a uma potência de 2, tipicamente 4 ou 8 KB
- O mecanismo de divisão do espaço de endereçamento lógico do processo é meramente operacional: os bits mais significativos do endereço definem o *número da página* e os menos significativos o *deslocamento*
- A memória principal, por seu lado, é vista também como dividida em blocos, os chamados *frames*, de tamanho igual às *páginas*
- É comum o *linker* organizar o espaço de endereçamento lógico do processo atribuindo o início de uma nova *página* a cada uma das regiões funcionalmente distintas (no caso da *stack*, o fim)

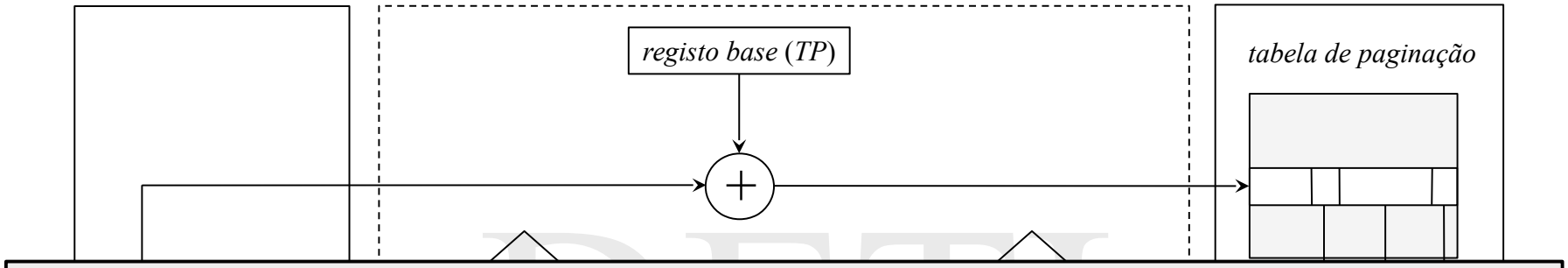
Arquitetura paginada



Arquitetura paginada



Arquitetura paginada



- Estruturando o espaço de endereçamento lógico do processo de modo a mapear a totalidade, ou pelo menos uma fração, do espaço de endereçamento fornecido pelo processador (em qualquer caso, sempre maior ou igual ao tamanho da memória principal existente), torna-se possível eliminar a necessidade do *registo limite* associado com a *tabela de paginação*
- Uma segunda vantagem daqui decorrente, é que a reserva de espaço na zona de definição dinâmica e no *stack* pode atingir a máxima amplitude possível

Arquitetura paginada

Conteúdo de cada entrada da tabela de paginação

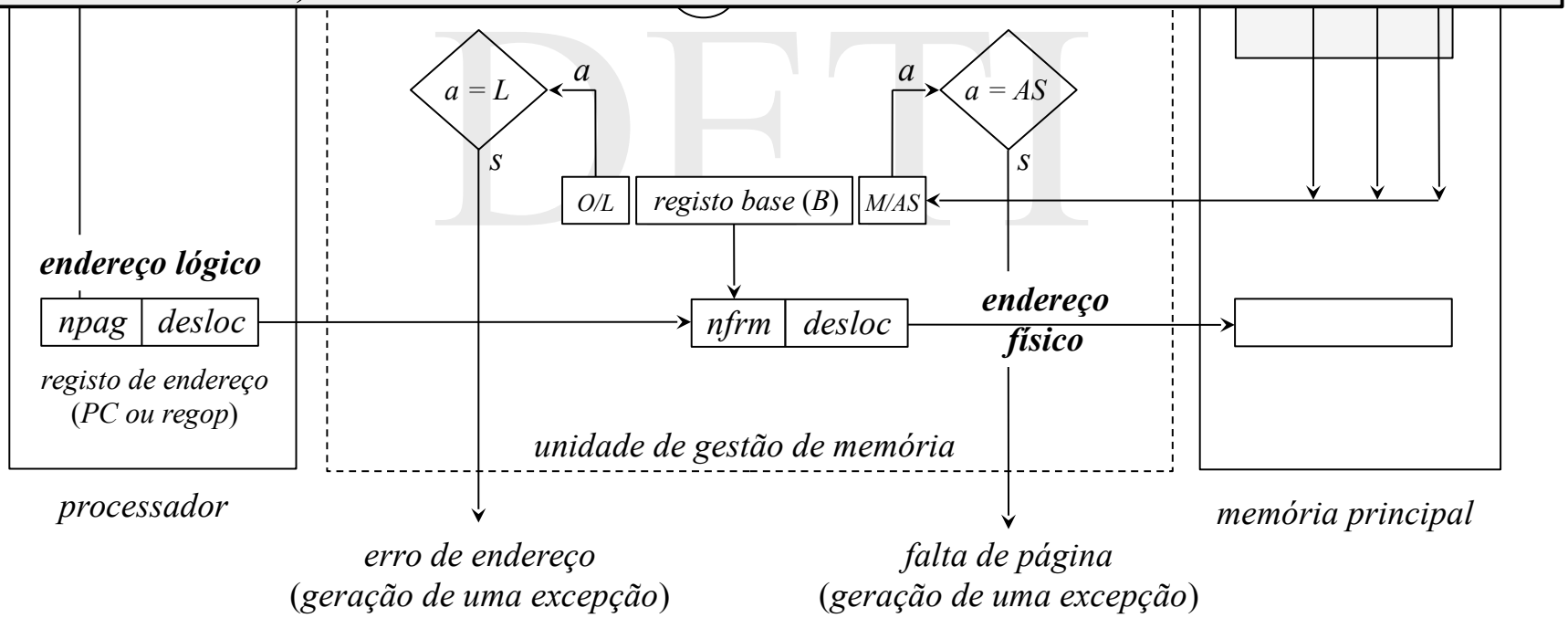
O/L	*M/AS*	*Ref*	*Mod*	*Perm*	*N. do frame em memória*	*N. do bloco na área de swapping*

- *O/L* – bit que sinaliza a ocupação ou não desta entrada (a não ocupação significa que ainda não foi reservado espaço na *área de swapping* para esta página)
- *M/AS* – bit que sinaliza se a página está ou não residente em memória principal
- *Ref* – bit que sinaliza se a página foi ou não referenciada para leitura e/ou escrita
- *Mod* – bit que sinaliza se a página foi ou não referenciada para escrita
- *Perm* – indicação do tipo de acesso permitido (*ronly* ou *read*/*write*, no caso mais simples; ou discriminação mais detalhada *rwx*, com sinalização em separado de acesso para leitura e/ou escrita (operandos) ou de execução (instruções)
- *Número do frame em memória* – localização da página, se residente em memória principal
- *Número do bloco na área de swapping* – localização da página na *área de swapping*, se já lhe foi atribuído espaço

(*geração de uma excepção*) (*geração de uma excepção*)

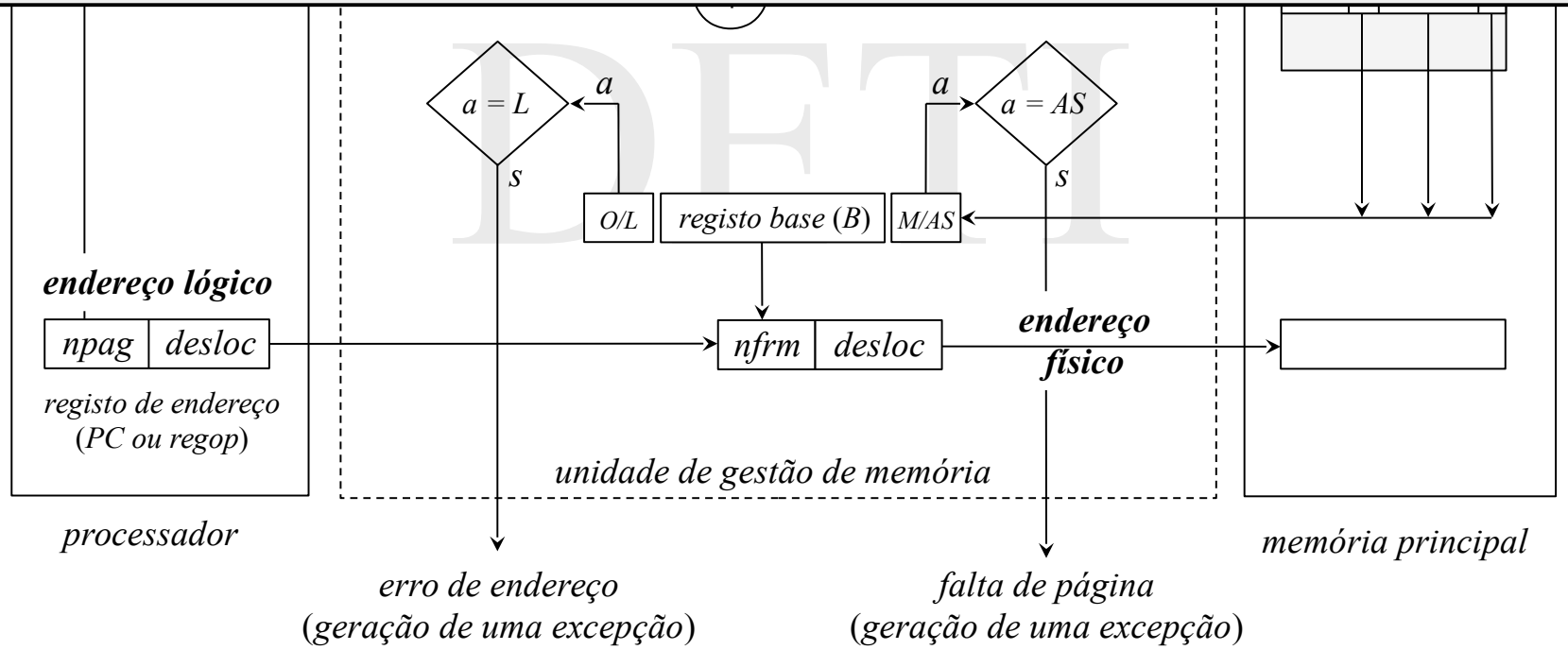
Arquitetura paginada

- O *registo limite* associado com uma página particular não existe nunca, pois o *endereço físico* é formado pela concatenação entre o campo *nfrm*, que identifica o *frame* de memória principal onde a página está localizada, e o campo *desloc*, que identifica o deslocamento dentro da página (ou do *frame*, já que eles têm o mesmo tamanho)



Arquitetura paginada

- As páginas correspondentes à zona de definição dinâmica e à *stack* só são criadas quando se torna necessário, o que permite poupar espaço na área de *swapping* e originar um erro de endereço sempre que se tenta aceder a uma página ainda não existente.



Arquitetura paginada

Vantagens

- *geral* – o âmbito da sua aplicação é independente do tipo de processos que vão ser executados (número e tamanho do seu espaço de endereçamento);
- *grande aproveitamento da memória principal* – não conduz a fragmentação externa e a fragmentação interna é praticamente desprezável;
- *não exige requisitos especiais de hardware* – a unidade de gestão de memória existente nos processadores atuais de uso geral está já preparada para a sua implementação.

Arquitetura paginada

Desvantagens

- *acesso à memória mais longo* – cada acesso à memória transforma-se num duplo acesso por consulta prévia da *tabela de paginação*
  - Este aspecto pode ser minimizado se a unidade de gestão de memória contiver *translation lookaside buffer* (*TLB*), para armazenamento das entradas da *tabela de paginação* recentemente mais referenciadas
- *operacionalidade muito exigente* – a sua implementação exige por parte do sistema de operação a existência de um conjunto de operações de apoio que são complexas e que têm que ser cuidadosamente estabelecidas para que não haja uma perda acentuada de eficência.

Arquitetura segmentada

- A *arquitetura paginada* divide o espaço de endereçamento lógico do processo de uma forma cega, praticamente sem usar qualquer informação sobre a estrutura subjacente. A exceção é, como foi referido atrás, a prática habitual de colocar sempre no início de uma nova página (no fim, no caso do *stack*) cada uma das regiões funcionalmente distintas
- Duas consequências:
  - A estrutura modular que está na base do desenvolvimento de uma aplicação com alguma complexidade, não é tida em conta e, em consequência, não é possível usar o *princípio da localidade de referência* de modo a minimizar o número de páginas que tenham que estar residentes em memória principal em cada etapa de execução do processo;
  - A gestão do espaço disponível entre a zona de definição dinâmica e o *stack* torna-se complicada e pouco eficiente, sobretudo, quando há a possibilidade de surgirem em *run time* múltiplas regiões de dados partilhados de tamanho variável, ou estruturas de dados de crescimento contínuo.

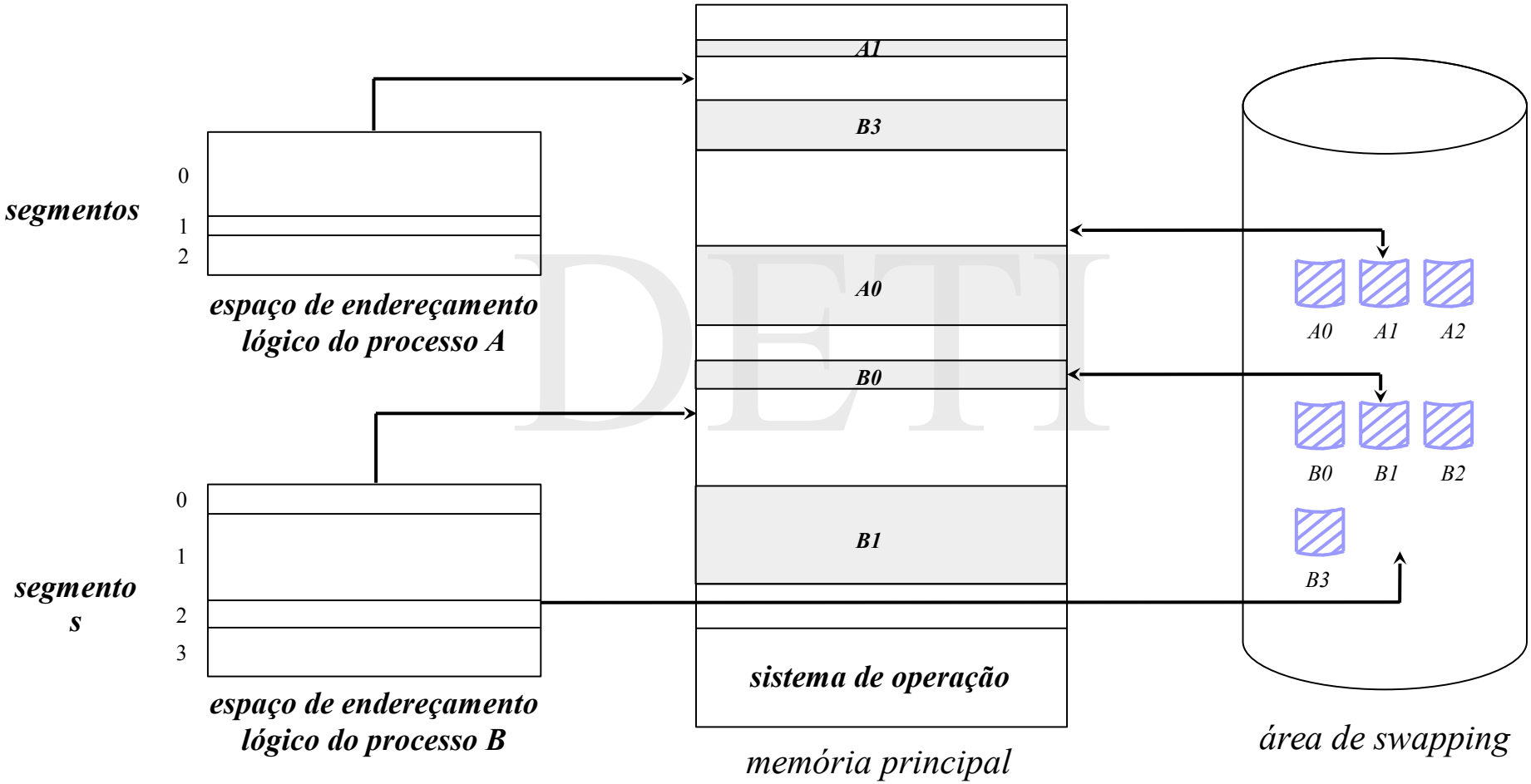
Arquitetura segmentada

- Uma solução para o problema é desdobrar-se o espaço de endereçamento lógico do processo, que constitui um espaço de endereçamento linear único no caso da *arquitetura paginada*, numa multiplicidade espaços de endereçamento lineares autónomos definidos na fase de *linkagem*
- Assim, cada *módulo* da aplicação (ficheiro em código fonte de compilação separada) vai originar dois espaços de endereçamento autónomos: um para o código e outro, na zona de definição estática, para as variáveis globais à aplicação (definidas localmente) e para as variáveis localmente globais (internas ao módulo)
- Cada um destes espaços de endereçamento autónomos designa-se de *segmento* e uma organização de memória virtual, baseada neste tipo de decomposição, constitui a chamada *arquitetura segmentada*
- Como é evidente, os blocos (ou *segmentos*), resultantes da divisão do espaço de endereçamento lógico do processo, podem ser de comprimento variável.

Arquitetura segmentada

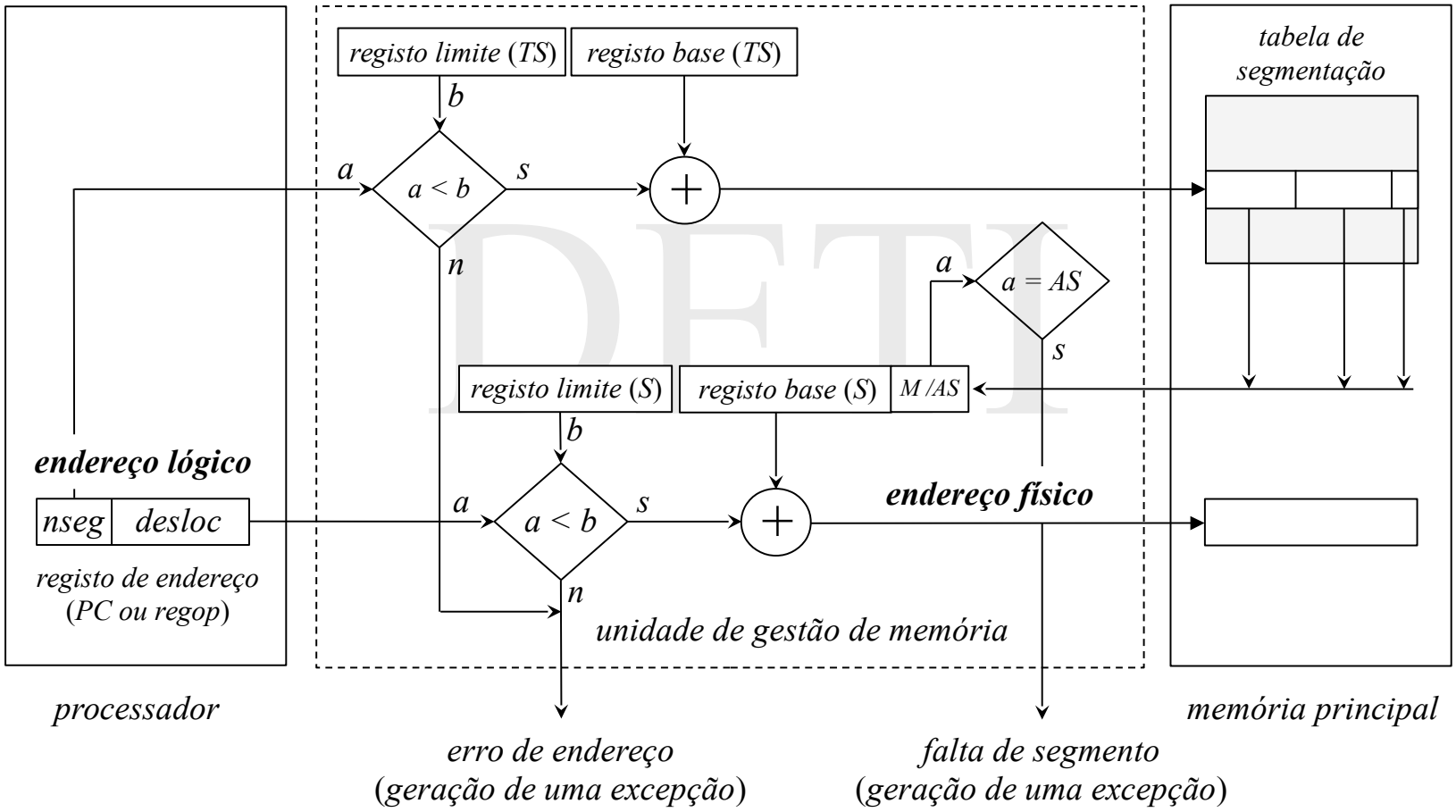
- Tipicamente, o *espaço de endereçamento lógico* do processo será formado pelos segmentos seguintes
  - *região de código* – um segmento por cada módulo que contenha código, quer seja local, quer global
  - *zona de definição estática* – um segmento por cada módulo que contenha a definição de variáveis globais à aplicação ou ao módulo
  - *zona de definição dinâmica local (heap)* – um segmento
  - *zona de definição dinâmica global* – um segmento por cada região de memória partilhada de dados;
  - *stack* – um segmento

Arquitetura segmentada



Arquitetura segmentada

Tradução de um endereço lógico num endereço físico



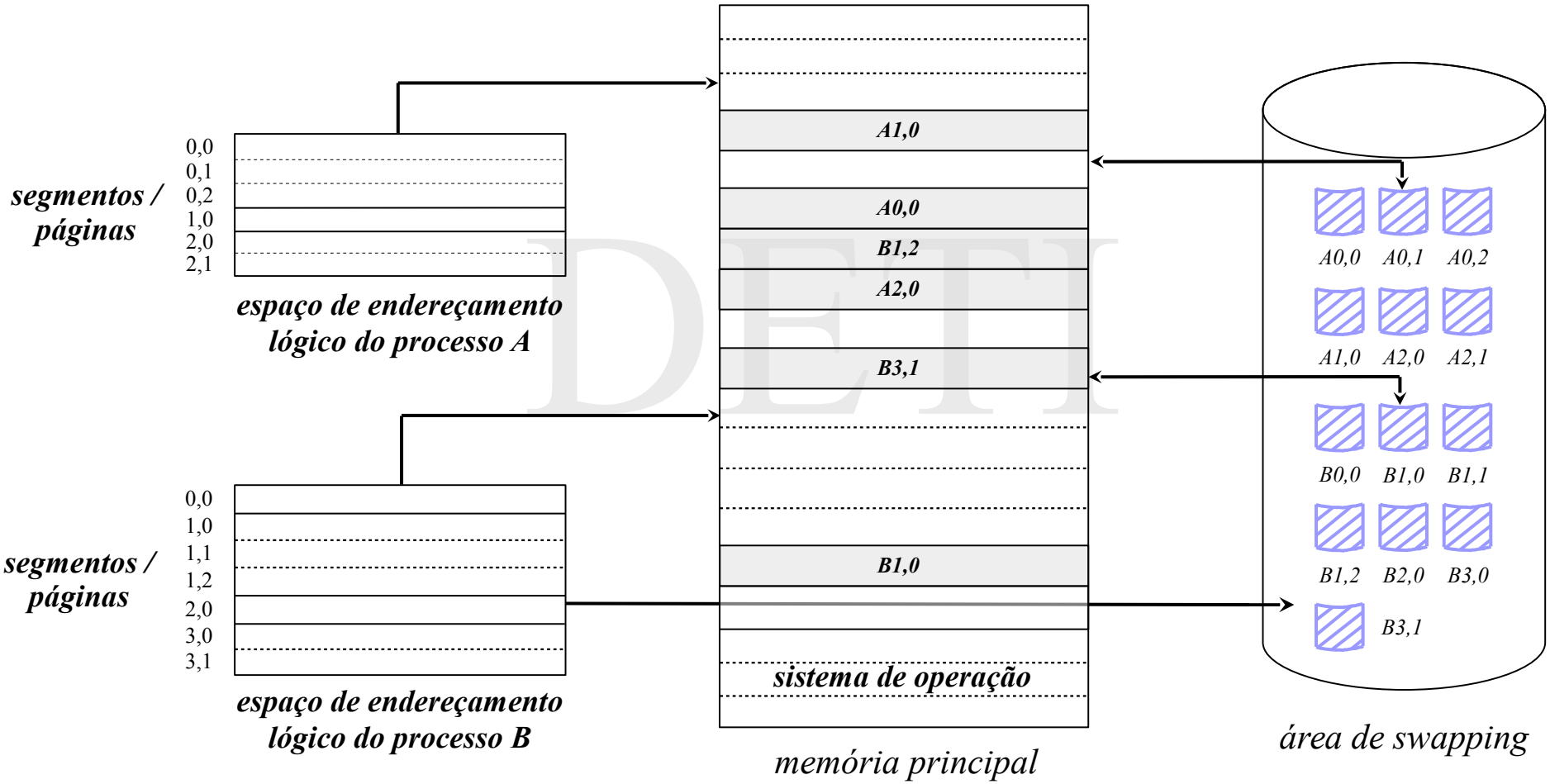
Arquitetura segmentada

- A *arquitetura segmentada* na sua versão pura tem pouco interesse prático porque, tratando a memória principal como um espaço contínuo, exige a aplicação de técnicas de reserva de espaço para carregamento de um *segmento* em memória que são em tudo semelhantes às usadas em organizações de memória real de partições variáveis
- Daqui resulta uma enorme fragmentação externa da memória principal com o consequente desperdício de espaço
- Além disso, segmentos de dados de crescimento contínuo conduzem a problemas adicionais: são facilmente concebíveis situações em que um acréscimo de tamanho do segmento não poderá ser realizado na sua localização presente, originando a sua transferência na totalidade para outra região da memória, ou, num caso limite em que não há espaço suficiente, ao bloqueio, ou suspensão, do processo, com a remoção do segmento, ou de todo o espaço de endereçamento residente, para a *área de swapping*

Arquitetura segmentada / paginada

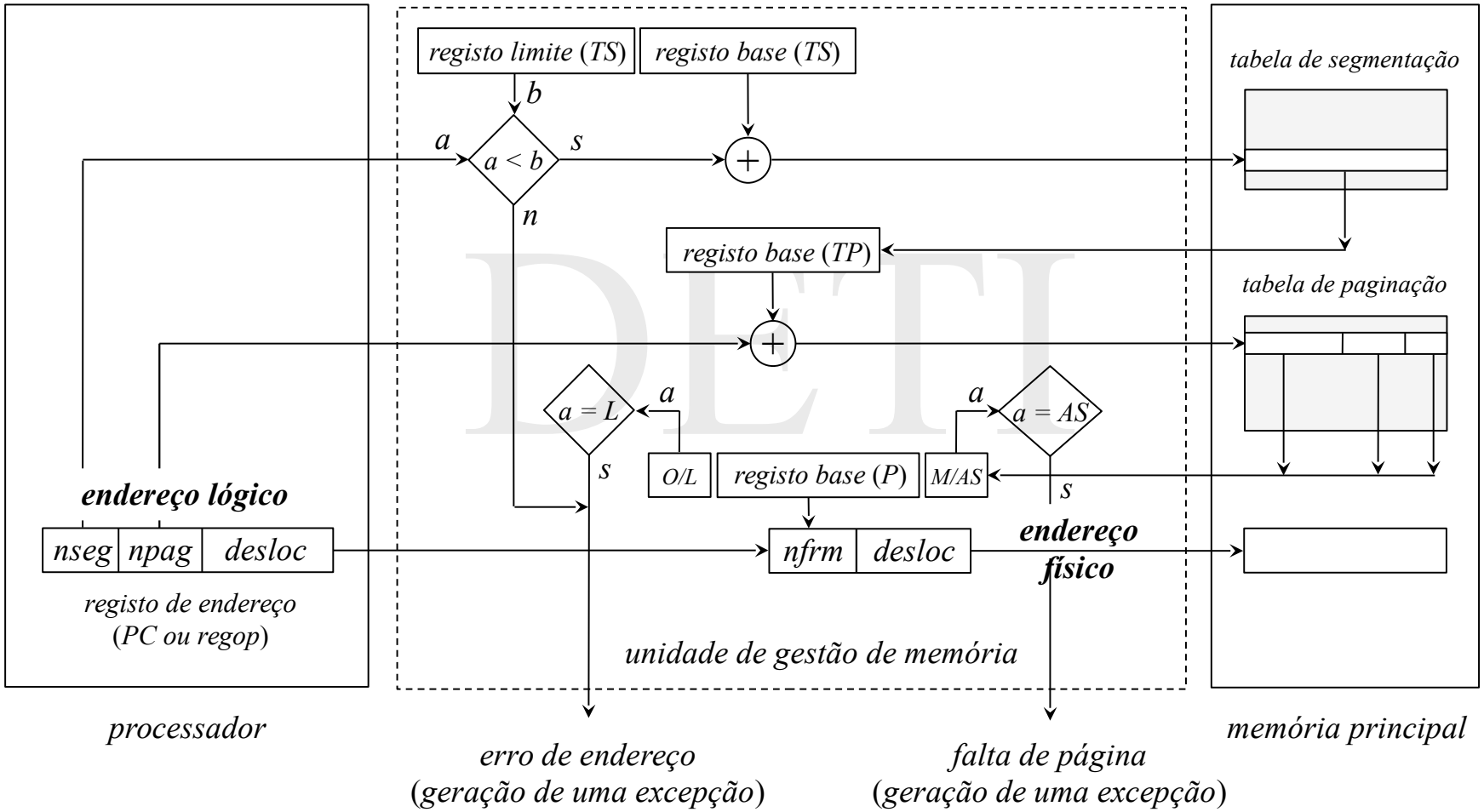
- Uma alternativa potencialmente interessante é uma arquitetura mista, designada de *arquitetura segmentada / paginada*, que é constituída pela combinação das características desejáveis das duas arquiteturas anteriores.
- Assim, tem-se que
  - A divisão do espaço de endereçamento lógico é primariamente segmentada, com a atribuição na fase de *linkagem* de múltiplos espaços de endereçamento lineares autónomos
  - Cada um destes espaços de endereçamento lineares é dividido em páginas, originando um mecanismo de carregamento de blocos em memória principal com todas as características da arquitetura paginada.

Arquitetura segmentada / paginada



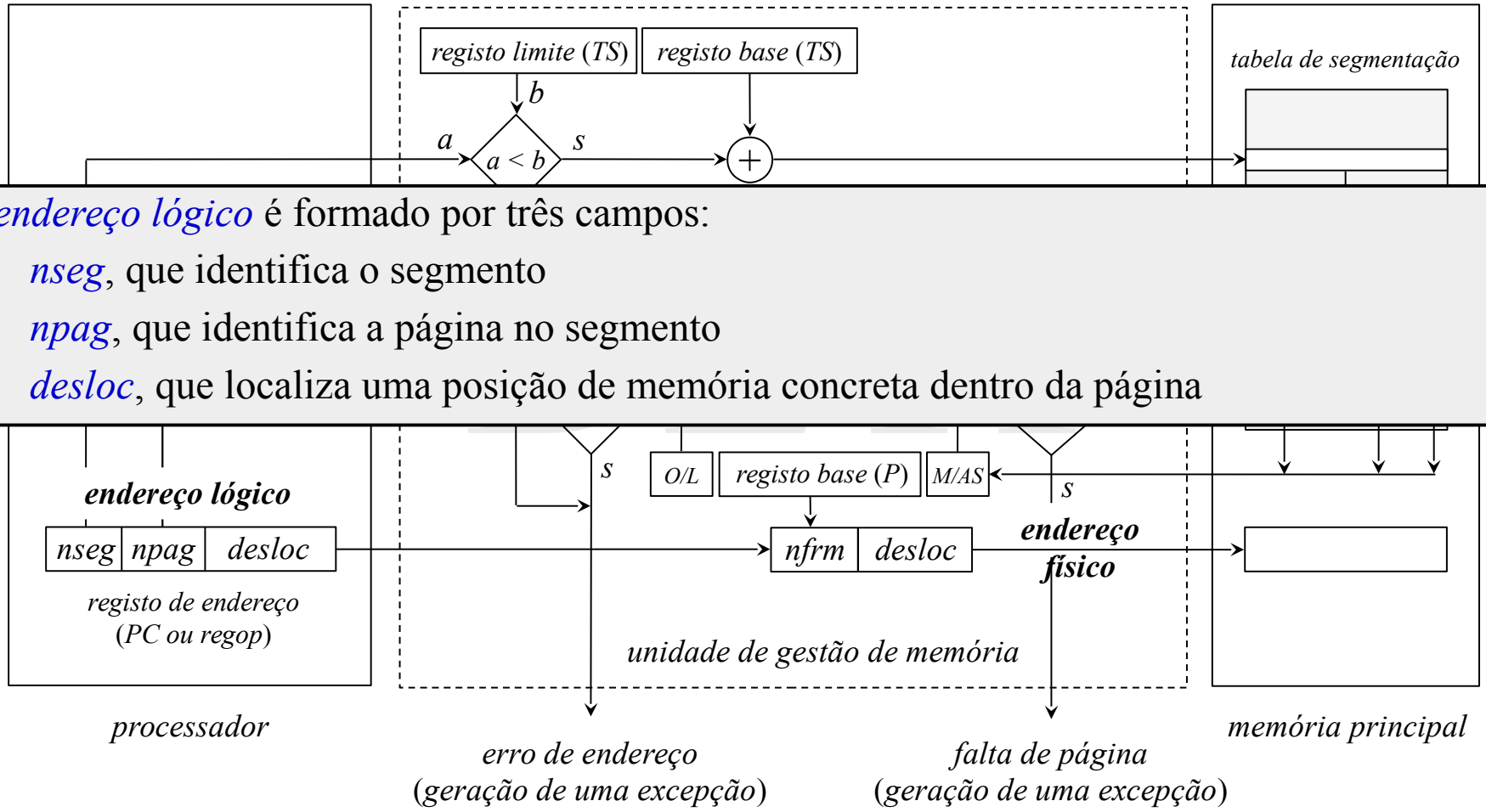
Arquitetura segmentada / paginada

Tradução de um endereço lógico num endereço físico



Arquitetura segmentada / paginada

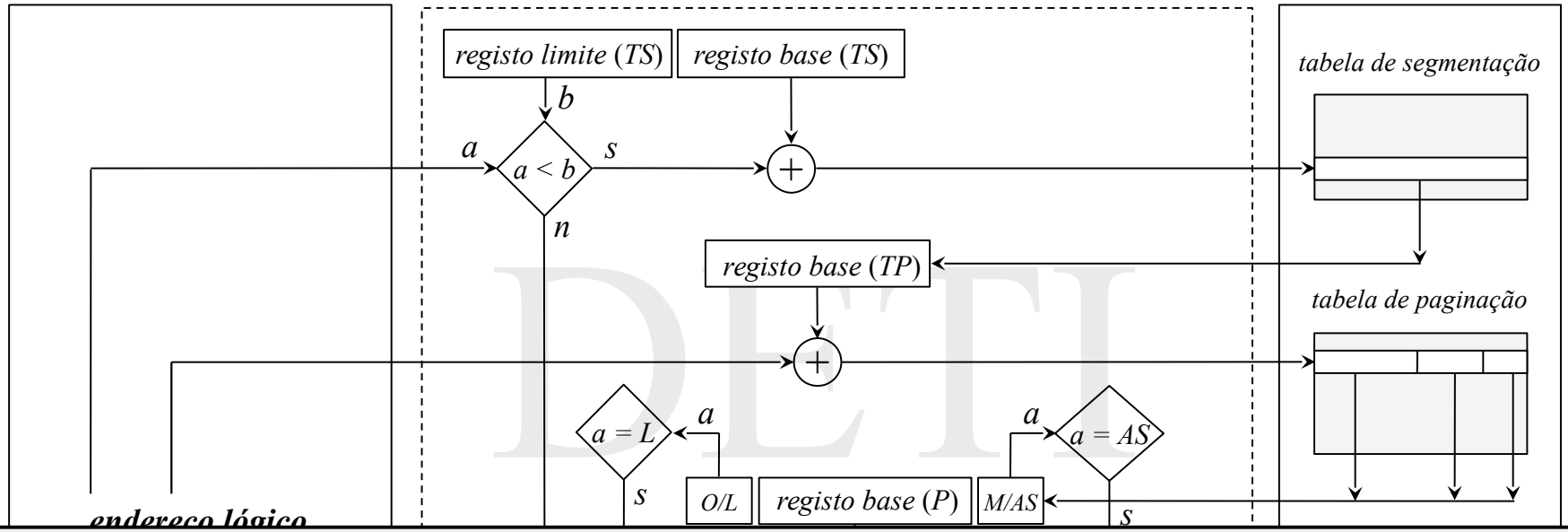
Tradução de um endereço lógico num endereço físico



- o *endereço lógico* é formado por três campos:
  - *nseg*, que identifica o segmento
  - *npag*, que identifica a página no segmento
  - *desloc*, que localiza uma posição de memória concreta dentro da página

Arquitetura segmentada / paginada

Tradução de um endereço lógico num endereço físico

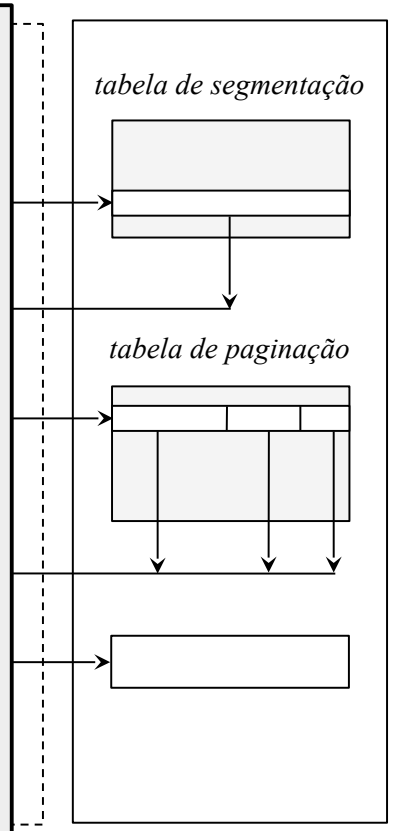


- A unidade de gestão de memória contém *três registos base e um registo limite* associados, respetivamente, com:
  - o endereço da *tabela de segmentação* do processo (o registo base TS)
  - o número de entradas da *tabela de segmentação* (o registo limite)
  - o endereço da *tabela de paginação* do segmento que está a ser referenciado (o registo base TP)
  - o *frame* de memória principal onde a página está localizada (o registo base P)

Arquitetura segmentada / paginada

Tradução de um endereço lógico num endereço físico

- Cada acesso à memória transforma-se em três acessos
  - No primeiro, é referenciada a entrada da *tabela de segmentação* do processo, associada com o segmento descrito no campo *nseg* do endereço lógico, para se obter o endereço da tabela de paginação do segmento;
  - No segundo, é referenciada a entrada da *tabela de paginação* do segmento, associada com a página descrita no campo *npag* do endereço lógico, para se obter o *frame* de memória principal onde a página está localizada;
  - No terceiro, é referenciada a posição de memória específica, sendo o cálculo do seu endereço feito concatenando o número do *frame* de memória principal, ocupado pela página, com *desloc*.



processador

memória principal

erro de endereço
(*geração de uma excepção*)

falta de página
(*geração de uma excepção*)

Arquitectura segmentada / paginada

Conteúdo de cada entrada da tabela de segmentação

Perm	*Endereço em memória da tabela de paginação do segmento*

Conteúdo de cada entrada da tabela de paginação de cada segmento

O/L	*M/AS*	*Ref*	*Mod*	*N. do frame em memória*	*N. do bloco na área de swapping*

- o aspecto mais saliente é o deslocamento do campo *Perm* da descrição de cada página para a descrição de cada segmento – o tipo de acesso é, assim, tratado de uma maneira global.

Arquitectura segmentada / paginada

Vantagens

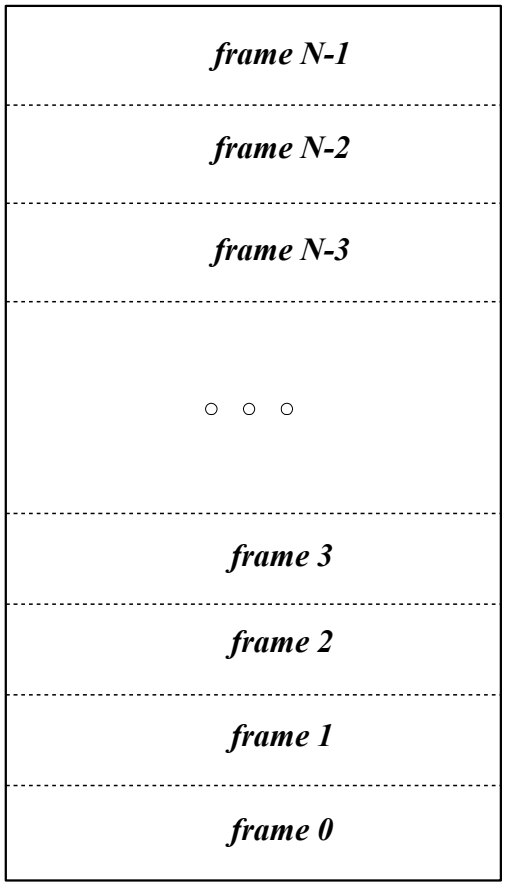
- *geral* – o âmbito da sua aplicação é independente do tipo de processos que vão ser executados (número e tamanho do seu espaço de endereçamento)
- *grande aproveitamento da memória principal* – não conduz a fragmentação externa e a fragmentação interna é praticamente desprezável
- *gestão mais eficiente da memória no que respeita a regiões de crescimento dinâmico*
- *minimização do número de páginas que têm que estar residentes em memória principal em cada etapa de execução do processo*

Arquitectura segmentada / paginada

Desvantagens

- *Exige requisitos especiais de hardware* – ao contrário do *Pentium* da Intel, nem todos os processadores atuais de uso geral estão preparados para a sua implementação
- *Acesso à memória mais longo* – cada acesso à memória transforma-se num triplo acesso por consulta prévia das *tabelas de segmentação e de paginação*
  - Este aspecto pode ser minimizado se a unidade de gestão de memória contiver *translation lookaside buffer* (*TLB*), para armazenamento das entradas da *tabela de paginação* recentemente mais referenciadaseste aspe
- *Operacionalidade muito exigente* – a sua implementação por parte do sistema de operação é mais exigente do que a *arquitetura paginada*.

Políticas de substituição de páginas em memória

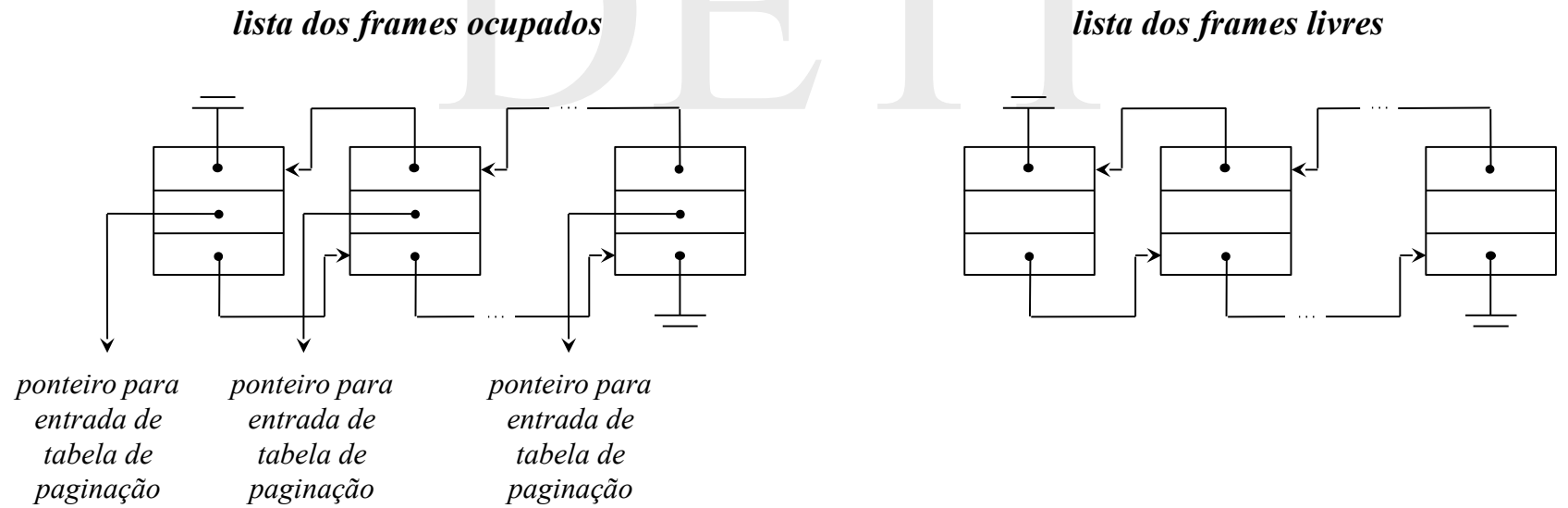


memória principal

- Numa arquitetura paginada ou segmentada / paginada, a memória principal é vista como dividida operacionalmente em *frames* do tamanho de cada página.
- Cada *frame* vai, em princípio, permitir o armazenamento do conteúdo de uma página do espaço de endereçamento lógico de um processo.
- As páginas podem estar em dois estados diferentes
  - *locked* – quando não podem ser removidas de memória; é o caso das páginas associadas com o *kernel* do sistema de operação, do *buffer cache* do sistema de ficheiros ou de um ficheiro mapeado em memória;
  - *unlocked* – quando podem ser removidas de memória; é o caso das páginas associadas com os processos convencionais.

Políticas de substituição de páginas em memória

- Os *frames* ocupados, associados com páginas *unlocked*, estão organizados numa lista biligada que descreve os *frames* passíveis de substituição.
- Os *frames* livres estão igualmente organizados numa lista biligada.
- O tipo de memória implementado pela *lista dos frames ocupados* depende do algoritmo de substituição utilizado.



Políticas de substituição de páginas em memória

- Quando ocorre uma *falta de página*, a situação mais comum é a *lista dos frames livres* estar vazia e, por isso, torna-se necessário selecionar um *frame* para substituição da *lista dos frames ocupados*.
- Alternativamente, pode manter-se sempre na *lista dos frames livres* alguns *frames*, sendo um deles usado para carregar a página em falta, e proceder-se em seguida à substituição de um *frame* ocupado. Como as operações decorrem em paralelo, este segundo método é mais eficiente.
- O problema que se coloca em qualquer caso, é que *frame* escolher para substituição? Teoricamente, deverá ser *um frame que não irá mais ser referenciado ou, sendo-o, sê-lo-á o mais tarde possível* – ***princípio da otimalidade***. Minimiza-se deste modo a ocorrência de outras *faltas de página*.
- O *princípio da otimalidade* é, porém, um princípio não causal e não pode ser diretamente implementado. Procura-se, assim, encontrar estratégias de substituição que sejam realizáveis e que, ao mesmo tempo, se aproximem tanto quanto possível do *princípio da otimalidade*.

Algoritmo LRU

- Uma aproximação excelente é a estratégia designada *LRU (least recently used)*, que visa encontrar o *frame* que não é referenciado há mais tempo
  - Assumindo o *princípio da localidade de referência*, se um *frame* não é referenciado há muito tempo, é fortemente provável que também não o venha a ser no futuro próximo
- Cada referência à memória tem que ser sinalizada com o instante da sua ocorrência (conteúdo de um *timer* ou de um contador)
  - A unidade de gestão de memória tem de ter capacidade para o fazer (pouco provável) ou hardware específico terá que ser acoplado
- Quando ocorre uma *falta de página*, a *lista ligada dos frames ocupados* tem que ser percorrida para determinar aquele cujo último acesso foi realizado há mais tempo.
  - Custo de implementação elevado e pouco eficiente

Algoritmo NRU

- Uma aproximação possível ao *LRU*, de implementação menos exigente e relativamente eficiente, é a estratégia designada *NRU (not recently used)*
  - São apenas usados os bits *Ref* e *Mod* que são processados tipicamente por uma unidade de gestão de memória convencional (*Ref* é atualizado em cada acesso à página, *Mod* é atualizado em cada escrita na página).
- Periodicamente, o sistema de operação percorre a *lista dos frames ocupados* e coloca a zero o bit *Ref*. Assim, quando ocorre uma *falta de página*, os *frames* da *lista dos frames ocupados* enquadram-se numa das classes seguintes

	Ref	*Mod*
classe 0	0	0
classe 1	0	1
classe 2	1	0
classe 3	1	1

- A seleção da página a substituir será feita entre aquelas pertencentes à classe de ordem mais baixa existente atualmente na *lista dos frames ocupados*

Algoritmo FIFO

- Um critério baseado no tempo de estadia das páginas em memória principal pode ser usado.
- Baseia-se no pressuposto de que quanto mais tempo as páginas residirem em memória, menos provável será que elas sejam referenciadas a seguir
- Considera-se que a *lista dos frames ocupados* está organizada num FIFO que espelha a ordem de carregamento das páginas correspondentes em memória principal
- Quando ocorre uma *falta de página*:
  - retira-se do *FIFO* o elemento correspondente à página há mais tempo em memória

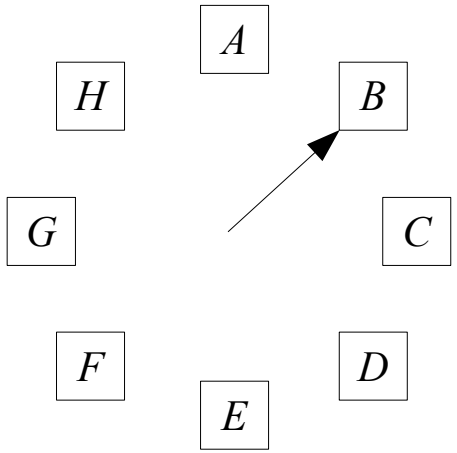
- O pressuposto por si só é extremamente falível (veja-se, por exemplo, as páginas associadas com o código do editor de texto ou do compilador num ambiente de desenvolvimento, ou com bibliotecas de sistema).
  - Contudo, se se lhe juntar um refinamento, este pressuposto pode tornar-se bastante robusto.

Algoritmo da segunda oportunidade

- Considera-se, tal como no anterior, que a *lista dos frames ocupados* está organizada num FIFO que espelha a ordem de carregamento das páginas correspondentes em memória principal
- Quando ocorre uma *falta de página*:
  - retira-se do *FIFO* o elemento correspondente à página há mais tempo em memória
  - se o seu bit *Ref* estiver a zero, a página associada é escolhida para substituição
  - caso contrário, coloca-se a zero o bit *Ref*, o nó é reintroduzido no fim do *FIFO* e o processo repete-se
- O nome do algoritmo advém da segunda oportunidade dada a uma página antes de ser substituída.

Algoritmo do relógio

- A estratégia subjacente ao *algoritmo da segunda oportunidade* pode ser tornada mais eficiente se a *lista dos frames ocupados* que implementa o *FIFO* for convertida numa lista circular em vez de linear
- As operações *fifoOut* e *fifoIn* transformam-se em meros incrementos de um ponteiro (módulo o número de elementos da lista)



- Quando ocorre uma *falta de página*:
- Enquanto o bit *Ref* do *frame* apontado pelo ponteiro não for zero
  - O bit Ref é colocado a zero
  - O ponteiro avança uma posição
- A página apontada é escolhida para substituição
- O ponteiro avança uma posição

Working Set

- Considere-se que, quando um processo é colocado pela primeira vez na fila de espera dos processos *READY-TO-RUN*, apenas a primeira e a última páginas do seu espaço de endereçamento (correspondentes, respetivamente, ao início do código e ao topo do *stack*) são colocadas em memória principal
- Quando o processador for atribuído ao processo, suceder-se-ão inicialmente *faltas de página* a um ritmo relativamente rápido e, depois, o processo entrará numa fase mais ou menos longa onde a execução decorrerá sem mais sobressaltos
- Está-se então perante uma situação em que, de acordo com o *princípio da localidade de referência*, as páginas associadas com a fração do espaço de endereçamento que o processo está atualmente a referenciar estão todas presentes em memória principal.
- Este conjunto de páginas é designado o ***working set*** do processo.

Working Set

- Ao longo do tempo o *working set* do processo vai variar, não só no que respeita ao número, como às páginas concretas que o definem
- Se o número de *frames* em memória principal atribuído ao processo é fixo e inferior à dimensão do seu *working set* atual, o processo está continuamente a gerar *faltas de página* e o seu ritmo de execução é muito lento
  - diz-se então que está em ***thrashing***
- Caso contrário, o processo alternará períodos curtos em que sofrerá um conjunto de *faltas de página* a um ritmo muito elevado, com períodos mais ou menos longos em que elas quase não ocorrem
- Tentar manter o *working set* do processo sempre presente em memória principal constitui, assim, o objectivo prioritário de qualquer política de substituição. Um meio de conseguir garanti-lo é atribuir novos *frames* ao processo sempre que ele se encontre num período de ritmo elevado de *faltas de página* e ir-lhe retirando os *frames* recentemente não referenciados em períodos de acalmia

Demand paging vs. prepaging

- Quando um processo é introduzido na fila de espera dos processos *READY-TO-RUN*, pela primeira vez ou, mais tarde, em resultado de uma suspensão, é preciso decidir que páginas deslocar para a memória principal
- Duas estratégias básicas podem ser seguidas
  - *demand paging* – constitui a estratégia minimalista e menos eficiente, nenhuma página é em princípio colocada e joga-se com o mecanismo de geração de *faltas de página* para formar o *working set* do processo
  - *prepaging* – constitui a estratégia mais eficiente em que se procura adivinhar o *working set* do processo para minimizar a geração de *faltas de página*
    - na primeira vez, são colocadas as duas páginas atrás referidas
    - nas vezes seguintes, o conjunto de páginas residente no momento em que ocorreu a suspensão

Substituição global vs. substituição local

- Um último aspecto a considerar nas políticas de substituição de páginas é o âmbito de aplicação dos algoritmos de substituição. Podem ser de aplicação
  - *local* – se a escolha for efetuada entre o conjunto de *frames* atribuído ao processo;
  - *global* – se a escolha for efetuada entre todos os *frames* que constituem a *lista dos frames ocupados*.
- O âmbito de aplicação *global* é habitualmente preferível, porque permite enquadrar grandes variações no *working set* dos processos, com parte do seu espaço de endereçamento residente em memória principal, sem que daqui resulte desperdício de memória ou *thrashing*.
- Note-se, porém, que o *thrashing* não é completamente eliminado. Pode sempre ocorrer uma situação em que o somatório dos *working sets* dos processos é superior aos número de *frames unlocked* disponíveis em memória principal. Neste caso, a solução é ir sucessivamente suspendendo processos até que o problema desapareça.

Bibliografia

Operating Systems Concepts, Silberschatz, Galvin, Gagne, John Wiley & Sons, 9th Ed

- Capítulo 8: *Memory Management* (secções 8.1 a 8.5)
- Capítulo 9: *Virtual Memory* (secções 9.1 a 9.6)

Modern Operating Systems; Tanenbaum & Bos; Prentice-Hall International Editions, 4rd Ed

- Capítulo 3: *Memory Management* (secções 3.1 a 3.7)

Operating Systems, Stallings, Prentice-Hall International Editions, 7th Ed

- Capítulo 7: *Memory management* (secções 7.1 a 7.4)
- Capítulo 8: *Virtual memory* (secções 8.1 a 8.2)