

SO-resumo-dcrvl-0.8.0

Notas:

Notem que o resumo começou como tal mas quando comecei a ver o tempo a passar ficou copy paste. Quem quiser melhorar o resumo é bem vindo , é favor reenviar o resumo melhorado para o fórum de ECT.

Histórico:

0.8.0 – 2010-01-14 – versão inicial.

Resumo de SO – em português decorável

Índice de conteúdos

Resumo de SO – em português decorável.....	1
Conceitos Introdutórios.....	6
Sistema de Operação: visão simplificada.....	6
Perspectiva 'bottom-up' (ou do construtor).....	6
Perspectiva 'top-down' (ou do programador).....	6
Exemplos das funcionalidades criadas pelo sistema de operação.....	6
Evolução dos sistemas de operação.....	7
Processamento em série.....	7
Sistema de tipo batch simples.....	7
Objectivo.....	7
Método.....	8
Sistema de tipo batch multiprogramado.....	8
Objectivo.....	8
Método.....	8
Sistema de tipo interactivo ('time-sharing').....	8
Objectivo.....	9
Método.....	9
Sistema de operação de tempo real.....	9
Objectivo.....	9
Método.....	9
Sistema de operação de rede.....	9
Objectivo.....	10
Serviços.....	10
Sistema de operação distribuído.....	10
Objectivo.....	10
Metodologia.....	10
Multiprocessamento vs. Multiprogramação.....	11
Paralelismo.....	11
Concorrência.....	11
Características comuns aos sistemas de operação actuais.....	12
Arquitectura interna.....	12
abordagem monolítica (tradicional).....	12
abordagem modular (actual).....	12
arquitectura em camadas.....	13
kernel minimalista.....	13
microkernel.....	13
módulos carregados dinamicamente.....	13
Arquitectura interna tradicional do Unix.....	14
Arquitectura interna contemporânea do Unix.....	15
Gestão do Processador.....	16
Programa vs. Processo.....	16
Tracing de execução num ambiente multiprogramado.....	16
Modelação dos processos.....	17
Diagrama de estados de um processo.....	17
Unix – Lançamento da shell.....	22
Criação de um processo em Unix.....	23
Ambiente de execução de um programa em C.....	24

Utilização das funções de limpeza.....	25
Espaço de endereçamento de um processo.....	26
Argumentos da linha de comando e variáveis de ambiente.....	27
Tabela de controlo de processos.....	28
Níveis de funcionamento do processador.....	29
Processamento de uma excepção genérica.....	30
Comutação de processos.....	31
Políticas de scheduling.....	33
Non-preemptive scheduling.....	33
Preemptive scheduling.....	33
Critérios a satisfazer pelos algoritmos de scheduling.....	33
Valorizando o critério de justiça.....	34
Definição de prioridades.....	35
Prioridade estática.....	35
Prioridade dinâmica.....	36
Diagrama de estados de um processo em Unix.....	39
Scheduling em Linux	40
Processos vs. Threads.....	41
Vantagens de um ambiente multithreaded.....	42
Organização de um programa multithreaded.....	42
Diagrama de estados de um thread.....	43
Suporte à implementação de um ambiente multithreaded.....	43
user threads.....	43
kernel threads.....	43
Threads em Linux.....	44
Comunicação entre Processos.....	45
Princípios gerais.....	45
Relação de competição por um recurso.....	47
Relação de partilha de dados.....	47
Relação produtor-consumidor	47
Acesso a uma região crítica com exclusão mútua.....	49
Tipo de soluções.....	49
soluções software.....	49
soluções hardware.....	49
Alternância estrita.....	50
Análise Crítica.....	50
Etapas da construção de uma solução	51
Análise Crítica.....	51
Análise Crítica.....	52
Análise Crítica.....	53
Algoritmo de Dekker (1965)	54
Análise Crítica.....	54
Algoritmo de Dijkstra (1966)	55
Algoritmo de Peterson (1981).....	56
Análise Crítica.....	56
Extensão do algoritmo de Peterson.....	57
Instruções de activação e de inibição das interrupções.....	58
Sistemas Computacionais Monoprocessador.....	58
Sistemas Computacionais Multiprocessador com Memória Partilhada.....	58
Instruções de tipo read-modify-write.....	58
Busy waiting.....	59
perda a eficiência.....	59

constrangimentos no estabelecimento do algoritmo de scheduling.....	59
BLA BLA BLA BLA:.....	60
Semáforos.....	60
sem_down.....	61
sem_up.....	61
Problema dos produtores / consumidores (semáforos).....	62
Crítica ao uso de semáforos.....	63
vantagens.....	63
desvantagens.....	64
Monitores.....	64
BLA BLA BLA BLA:.....	64
wait.....	65
signal.....	65
Monitor de Hoare.....	67
Monitor de Brinch Hansen	68
Monitor de Lampson / Redell.....	69
Problema dos produtores / consumidores (monitores).....	70
Passagem de mensagens.....	72
sincronização não bloqueante.....	72
sincronização bloqueante.....	72
rendez-vous.....	73
remota.....	73
Problema dos produtores / consumidores (pass. de mens.).....	76
Jantar dos filósofos.....	77
Jantar dos filósofos – semáforos.....	79
Jantar dos filósofos – monitor.....	81
Jantar dos filósofos – mensagens.....	83
Recursos.....	86
recursos preemptable.....	86
recursos non-preemptable.....	86
Caracterização esquemática de deadlock.....	87
Condições necessárias à ocorrência de deadlock	87
condição de exclusão mútua.....	87
condição de espera com retenção.....	87
condição de não libertação.....	88
condição de espera circular (ou ciclo vicioso).....	88
Exemplo: Jantar dos filósofos.....	88
Análise Crítica.....	88
exclusão mútua.....	88
espera com retenção.....	88
não libertação.....	89
formou-se uma cadeia circular de filósofos e.....	89
Prevenção de deadlock no sentido estrito.....	90
Negando a condição de espera com retenção.....	90
Impondo a condição de libertação de recursos.....	91
Negando a condição de espera circular.....	92
Prevenção de deadlock no sentido estrito.....	93
Análise Crítica.....	93
negação da condição de exclusão mútua.....	93
negação da condição de espera com retenção.....	93
imposição da condição de não libertação.....	94
negação da condição de espera circular.....	94

Prevenção de deadlock no sentido lato.....	94
Algoritmo dos banqueiros de Dijkstra.....	95
Exemplo: Jantar dos filósofos.....	95
Algoritmo dos banqueiros de Dijkstra.....	96
Análise Crítica.....	97
Detecção e recuperação de deadlock	98
estratégia da avestruz.....	98
detecção de deadlock.....	98
libertação forçada de um recurso.....	98
rollback.....	98
morte de processos.....	98
Sinais em Unix.....	98
ignorá-lo.....	99
bloqueá-lo.....	99
executar uma acção associada.....	99
kill.....	99
raise.....	99
sigaction.....	99
sigprocmask.....	99
sigpending.....	99
sigsuspend.....	99
Tabela dos sinais mais comuns (standard Posix.1).....	100
Exemplo de processamento de um sinal.....	101
Semáforos em Unix.....	102
tratamento em bloco de grupos de semáforos.....	102
operações básicas diferentes.....	102
Memória partilhada em Unix.....	102
zona de definição dinâmica.....	102
zona de código.....	102
criação ou ligação a uma região previamente criada.....	103
anexação da região ao espaço de endereçamento do processo.....	103
desanexação da região do espaço de endereçamento do processo.....	103
destruição de uma região existente.....	103
Threads e monitores em Unix	104
pthread_create.....	104
pthread_exit.....	104
pthread_join.....	104
pthread_self.....	104
pthread_once, pthread_mutex_*, pthread_cond_*.....	104
Mensagens em Unix.....	104
Pipes em Unix.....	105
o canal de comunicação é unidireccional.....	105
só pode ser usado entre processos que estão relacionados entre si.....	105
Exemplo de comunicação pai – filho.....	107
Gestão de memória.....	109
Introdução.....	109
Papel da gestão de memória em multiprogramação.....	111
Etapas da produção de um programa.....	112
Organização de memória real.....	117
Tradução de um endereço lógico num endereço físico.....	119
Arquitectura de partições fixas.....	125
Vantagens.....	126

Desvantagens.....	126
Arquitectura de partições variáveis.....	126
Vantagens.....	128
Desvantagens.....	129
Organização de memória virtual.....	129
Organização de memória virtual: acesso à memória.....	131
Organização de memória virtual: estado CREATED.....	133
Organização de memória real - continuação.....	133
Organização de memória virtual: estado READY-TO-RUN.....	136
Organização de memória virtual: excepção por falta de bloco.....	137
Organização de memória virtual: análise crítica.....	137
Organização de memória virtual: optimização.....	137
Arquitectura paginada.....	138
Arquitectura paginada: acesso à memória.....	139
Arquitectura paginada: entrada da tabela de paginação.....	141
Arquitectura paginada: vantagens.....	141
Vantagens.....	141
Desvantagens.....	142
Arquitectura segmentada.....	142
Arquitectura segmentada / paginada	144
Conteúdo de cada entrada da tabela de segmentação.....	148
Conteúdo de cada entrada da tabela de paginação de cada segmento.....	148
Vantagens.....	148
Desvantagens.....	148
Política de substituição de páginas em memória.....	149
Algoritmo LRU.....	150
Algoritmo da segunda oportunidade.....	151
Algoritmo do relógio.....	152
Working Set.....	152
Demand paging vs. prepaging.....	153
Substituição global vs. substituição local.....	153

Conceitos Introdutórios

Sistema de Operação: visão simplificada

- é o programa base executado pelo sistema computacional;
- permite que o hardware interaja com os utilizadores para a realização de trabalho útil
- este ambiente pode ser de dois tipos:
 - gráfico – uso de janelas no monitor que agem como modelos de interacção entre o utilizador e aplicações, através de diversos artefactos.
 - linguagem de comandos (shell) – uso de texto como modelo de interacção entre o utilizador e aplicações; podem ser conjugados para realização de tarefas mais complexas, formando uma metalinguagem.

Perspectiva 'bottom-up' (ou do construtor)

Sistema computacional – sistema formado por um conjunto de recursos (processador(es), memória principal, memória de massa e diferentes tipos de controladores de dispositivos de entrada / saída) destinados ao processamento e armazenamento de informação.

Sistema de operação

- o programa que gera o sistema computacional
- faz a atribuição dos seus diferentes recursos aos programas que por eles competem;
- **objectivo:** garantir uma utilização eficiente dos recursos existentes, rentabilizando ao máximo o sistema computacional

Perspectiva 'top-down' (ou do programador)

Sistema de operação

- uma abstracção do sistema computacional
- fornece, assim, uma máquina virtual e consequentemente um ambiente uniforme de programação;
- o trabalho sobre hardware é efectuado através de chamadas ao sistema uniformes entre diversos sistemas computacionais [diferentes], aumentando assim a portabilidade de aplicações

Exemplos das funcionalidades criadas pelo sistema de operação

- estabelece o ambiente base de interacção com o utilizador;

- disponibiliza facilidades para o desenvolvimento e execução de programas;
- fornece mecanismos para a execução controlada de programas, sua comunicação e sincronização mútuas;
- permite dissociar as limitações impostas pela memória principal do espaço de endereçamento do programa
- organiza a memória de massa em sistemas de ficheiros
- define um modelo geral de acesso aos dispositivos de entrada/saída;
- detecta e trata adequadamente situações de erro.

Evolução dos sistemas de operação

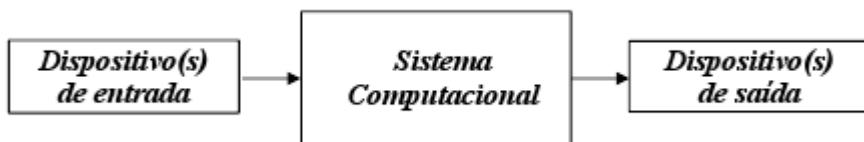
(não deve valer a pena saber)

Processamento em série



- o operador tem um controlo completo sobre o sistema computacional
- o sistema operativo é elementar - um conjunto de rotinas que estabelecem a comunicação com os dispositivos de entrada / saída (IOCS – input / output control system).

Sistema de tipo batch simples



Objectivo

manter o processador ocupado [por minimização da intervenção humana]

Método

- job - agrupamento de todas as operações relativas ao tratamento de um programa
- fornecimento em bloco de um conjunto (batch) de jobs a serem processados sequencialmente
- aparecimento do sistema de operação como programa monitor que :
 - controla o sistema computacional
 - interpreta comandos escritos numa linguagem própria (job control language):

\$JOB #####, \$FORTRAN #####, \$RUN, \$DATA #####, \$END.

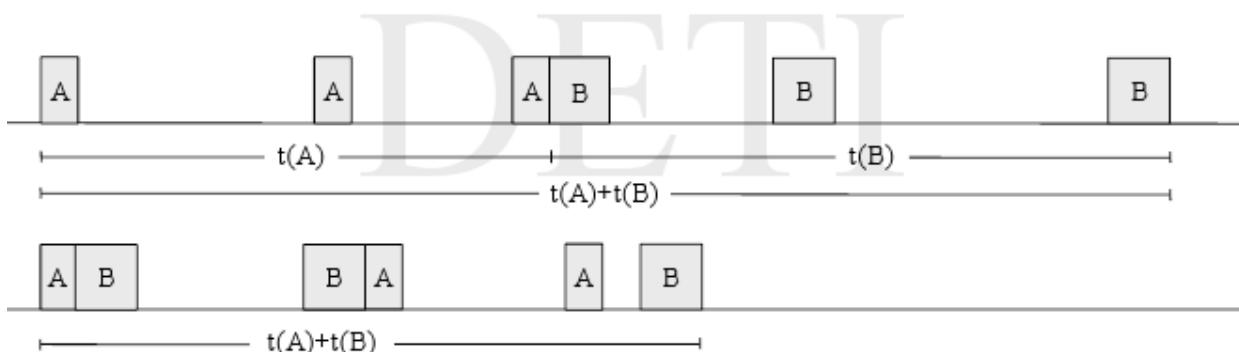
Sistema de tipo batch multiprogramado

Objectivo

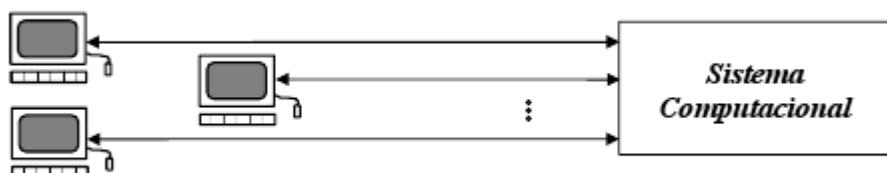
manter o processador ocupado [optimizando a ocupação do processador]

Método

enquanto o processador aguarda pela realização das operações de entrada / saída:
aproveitamento desses tempos para a execução de outro programa.



Sistema de tipo interactivo ('time-sharing')



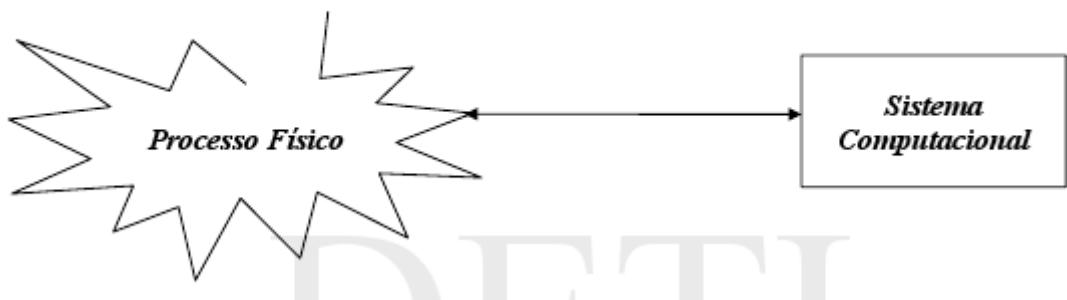
Objectivo

diminuição do tempo de resposta do sistema a solicitações externas – adequação do sistema a uso interativo por utilizadores

Método

- associados ao sistema computacional: múltiplos terminais (dispositivos de entradas/saída) para múltiplos utilizadores;
- atribuição sucessiva do processador aos múltiplos programas em execução (usando multiprogramação), durante intervalos de tempo muito curtos (time-slots)
- como a alternância entre processos é mais rápida que o tempo de resposta humano, cria-se a ilusão de existência de um sistema totalmente dedicado em cada terminal.

Sistema de operação de tempo real



Objectivo

utilizar o computador na

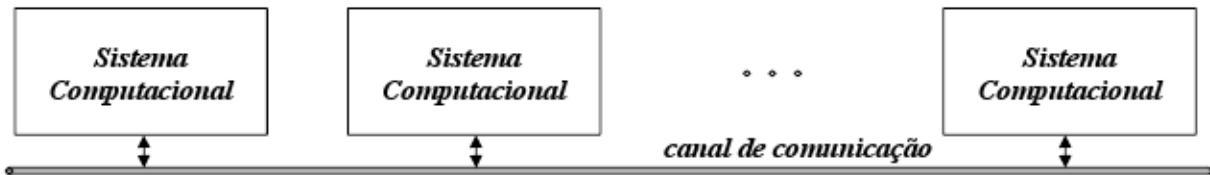
- monitorização e
- controlo on-line

de processos físicos.

Método

É um sistema interativo, mas impõe-se limites máximos aos tempos de resposta do sistema computacional a solicitações externas, usando diferentes classes [de prioridade dessas solicitações].

Sistema de operação de rede



Objectivo

estabelecimento de um conjunto de serviços comuns a toda uma comunidade (aproveitando as facilidades actuais de interligação de sistemas computacionais)

Serviços

- Browsers - acesso à Internet.
- E-mail - correio electrónico;
- autenticação remota;
- telnet, remote login - acesso a sistemas computacionais remotos;
- FTP - transferência de ficheiros entre sistemas computacionais;
- NFS - partilha de sistemas de ficheiros remotos, criando a ilusão de que são locais;
- impressoras, etc. - partilha de dispositivos remotos;

Sistema de operação distribuído

Objectivo

- estabelecer um ambiente de interacção com o(s) utilizador(es) que encara o sistema computacional paralelo como uma entidade única
- tirar partido das facilidades actuais de
 - interligação de sistemas computacionais distintos, ou
 - construção de sistemas computacionais com processadores múltiplos.

Metodologia

garantir um acesso transparente aos diferentes recursos dos sistemas computacionais interligados para obter

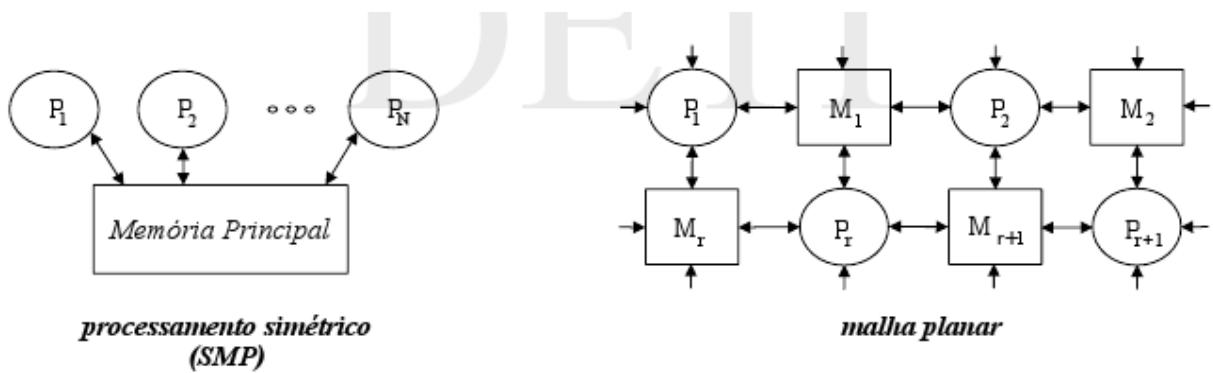
- a paralelização de aplicações;
- a detecção e tratamento de falhas;
- o balanceamento estático e/ou dinâmico de carga;
- o aumento da velocidade de processamento por incorporação de novos recursos.

Multiprocessamento vs. Multiprogramação

Paralelismo

- capacidade de um sistema computacional de poder executar em simultâneo dois ou mais programas;
- exige que o sistema computacional seja formado por dois ou mais processadores (um por cada programa em execução simultânea).

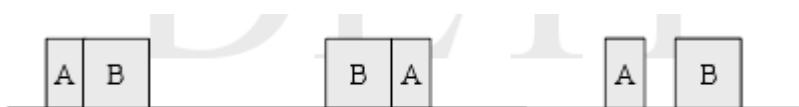
Quando tal acontece, diz-se que o sistema operativo associado tem características de **multiprocessamento**.



Concorrência

- ilusão criada por um sistema computacional de aparentemente poder executar em simultâneo mais programas do que o número de processadores existentes;
- exige que a atribuição do(s) processador(es) seja multiplexada no tempo entre os diferentes programas presentes.

Quando tal acontece, diz-se que o sistema operativo associado tem características de **multiprogramação**.



Num monoprocessador (sistema computacional com um único processador), os programas A e B estão a ser executados em concorrência.

Paralelismo VS Concorrência
Multiprocessamento VS Multiprogramação

Características comuns aos sistemas de operação actuais

- apresentam um ambiente gráfico de interacção com o utilizador;
- são sistemas
 - interactivos
 - multiutilizador;
 - multitarefa
 - de rede
 - acesso indistinto a
 - sistemas de ficheiros e
 - dispositivos de entrada / saída locais e remotos;
 - aplicações que permitem fazer o que se faz em sistemas de operação de rede [ver acima]
- implementam uma organização de memória virtual;
- têm uma colecção muito grande de device drivers para suportar variados dispositivos de entrada / saída;
- suportam a ligação dinâmica de dispositivos (plug and play).

Arquitectura interna

Assim, um S.O.

- é programa muito complexo
- tem um tamanho muito grande (pode atingir as dezenas de milhão de instruções)
- exige cuidados enormes na sua concepção e implementação para funcionar correctamente.
- ~~exige um Borges para ser lecionado~~

Abordagens mais comuns:

abordagem monolítica (tradicional)

o núcleo (kernel) é construído como um programa único, em que todas as

- funcionalidades, e
- estruturas de dados associadas,
- são directamente acessíveis;

resulta numa implementação

- muito eficiente,
- mas muito difícil de testar;

abordagem modular (actual)

o núcleo (kernel) é construído numa perspectiva top-down, em que as diferentes funcionalidades são

- identificadas e
- isoladas umas das outras através da especificação de um interface de comunicação;

resulta numa implementação

- menos eficiente do que a anterior,

- **mas** onde se torna possível o teste separado dos diversos componentes e
- efectuar alterações, com um mínimo de riscos.

A **abordagem modular** pode ser enquadrada sob diferentes perspectivas que não são necessariamente mutuamente exclusivas:

arquitectura em camadas

decomposição hierárquica em que cada novo módulo

- constrói-se sobre os módulos anteriores
 - usa as operações fornecidas pela camada anterior;
- embora conceptualmente seguro,
- exige uma cuidadosa definição das funcionalidades de cada camada
 - pode criar um overhead de comunicações que não é desprezável;
-

kernel minimalista

o kernel

- é reduzido a um número mínimo de funcionalidades,
- fornece tipicamente uma gestão mínima
 - do processador e
 - da memória principal e
 - um mecanismo base de comunicação

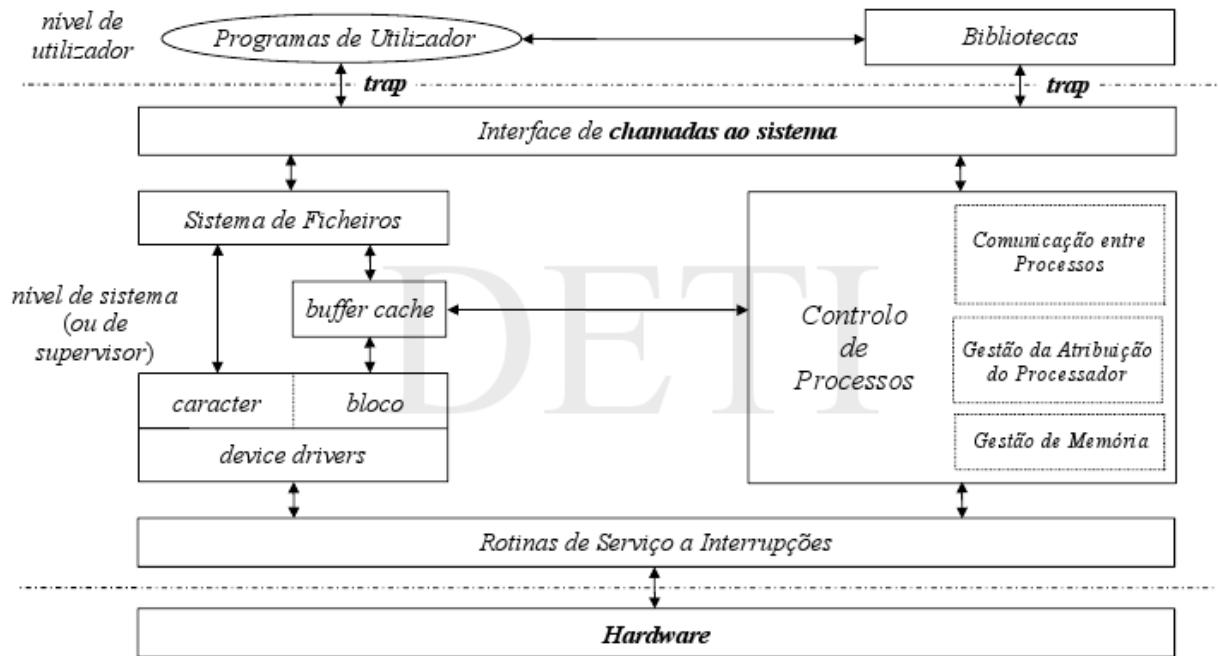
microkernel

- as restantes funcionalidades do sistema de operação são implementadas por processos executados em modo utilizador
- Estes comunicam entre si e com as aplicações usando o mecanismo de comunicação do kernel;

módulos carregados dinamicamente

- as restantes funcionalidades constituem módulos que são carregados e ligados dinamicamente ao kernel com este já em execução
- alguns destes módulos podem também ser removidos a qualquer momento (os associados a dispositivos plug-and-play, por exemplo).

Arquitectura interna tradicional do Unix



Nível de utilizador:

- programas de utilizador
- bibliotecas

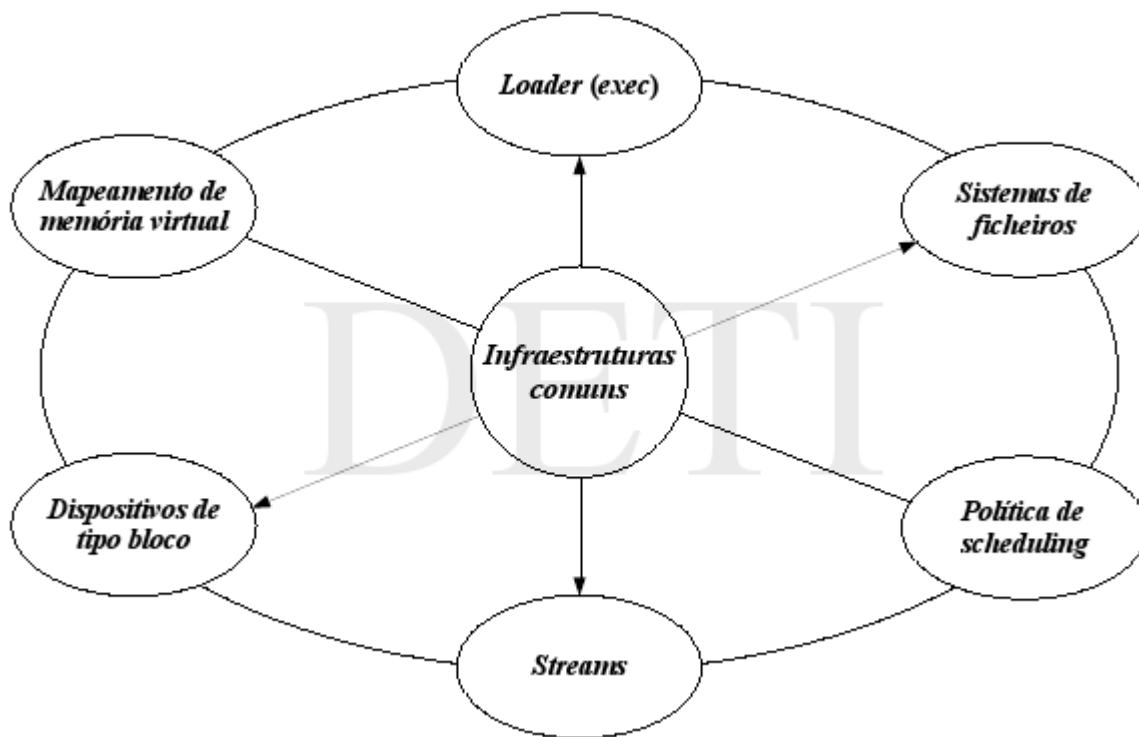
Comunicação feita com o nível de sistema (ou de supervisor) com **traps**

Nível de sistema (ou de supervisor)

- interface de chamadas ao sistema
 - sistemas de ficheiros
 - buffer cache
 - bloco
 - device drivers
 - carácter
 - controlo de processos
 - comunicação entre processos
 - gestão da atribuição do processador
 - gestão de memória
 - rotina de serviço ás interrupções

Hardware

Arquitectura interna contemporânea do Unix



Infraestruturas comuns

- Loader (exec)
- Streams
- Política de Scheduling
- Mapeamento de memória virtual
- Sistemas de Ficheiros
- Dispositivos de tipo bloco

Gestão do Processador

Programa vs. Processo

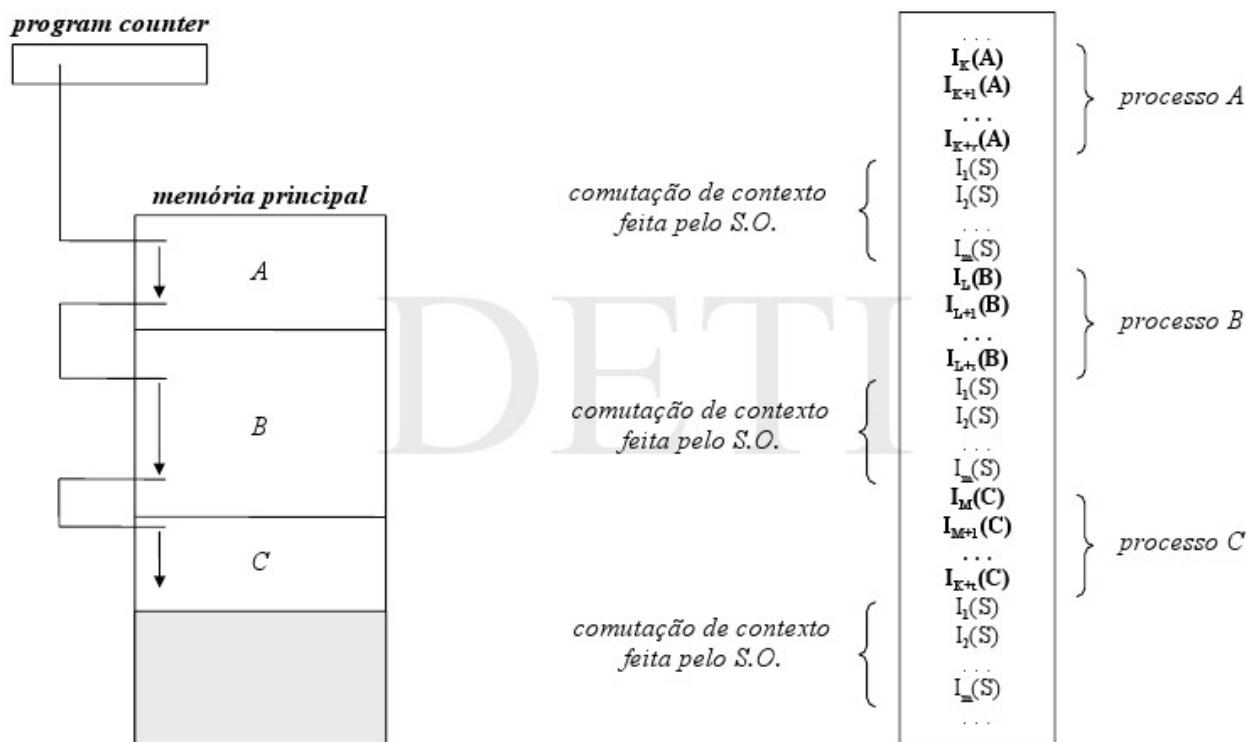
Programa – conjunto de instruções que descreve a realização de uma tarefa por um computador.

A execução de um programa designa-se de **processo** – realiza efectivamente a tarefa

Tratando-se da representação de uma actividade em curso, o processo caracteriza-se em cada instante por

- o código do programa respetivo
- o valor actual de todas as suas variáveis (**espaço de endereçamento**);
- o valor actual de todos os registos internos do processador;
- os dados que estão a ser transferidos de e para os dispositivos de entrada/saída;
- o seu estado de execução.

Tracing de execução num ambiente multiprogramado



Modelação dos processos

Para simplificar o modelo da multiprogramação, conceptualiza-se que um sistema computacional é constituído por tantos processadores quanto o número de processos que nele correm mas que apenas um pode estar ligado num dado momento (ou, no caso de um sistema dual-core, dois).

Para tal ser possível:

- a execução dos processos não é afectada pelo instante, ou local no código, onde ocorre a comutação;
- não há quaisquer restrições relativamente aos tempos de execução dos processos (totais ou parciais).

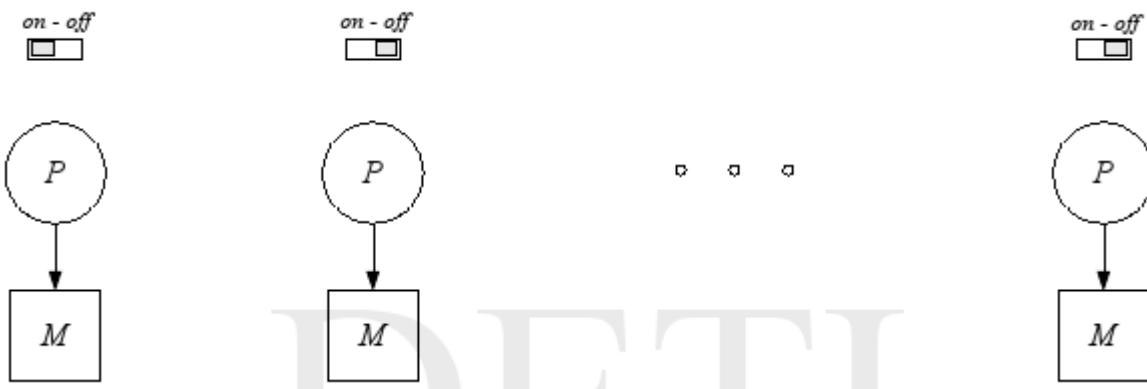
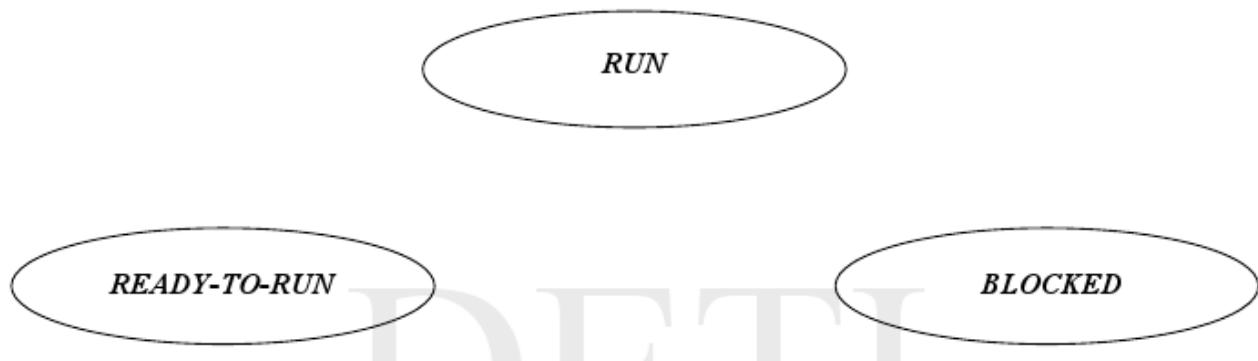
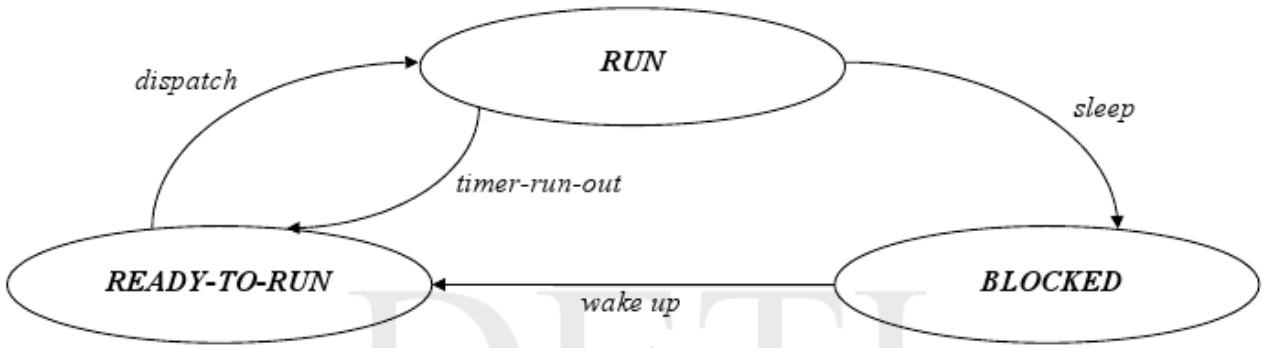


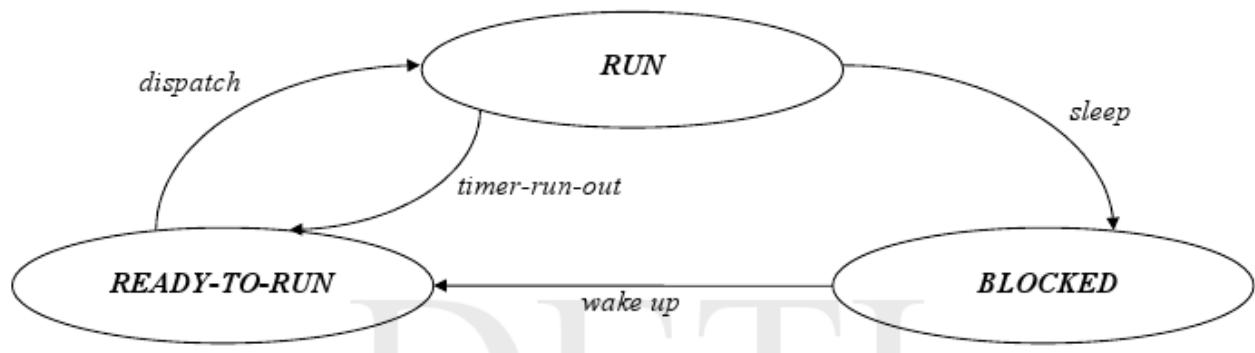
Diagrama de estados de um processo



- run – quando detém a posse do processador e está, por isso, em execução;
- ready-to-run – quando aguarda a atribuição do processador para começar ou continuar a sua execução;
- blocked – quando está impedido de continuar até que um acontecimento externo ocorra (acesso a um recurso, completamento de uma operação de entrada / saída, etc.).



- dispatch – um dos processos da fila de espera dos processos prontos a serem executados é seleccionado para execução;
- timer-run-out – o processo em execução esgotou o intervalo de tempo de processador que lhe tinha sido atribuído;
- sleep – o processo está impedido de prosseguir, aguardando a ocorrência de um acontecimento externo;
- wake up – ocorreu entretanto o acontecimento externo que o processo aguardava.



- As transições entre estados resultam normalmente de uma intervenção externa, mas podem nalguns casos ser despoletadas pelo próprio processo.
- A parte do sistema de operação que lida com estas transições, chama-se scheduler [neste caso, do processador].

O scheduler constitui parte integrante do kernel e é responsável

- pelo tratamento de excepções
- agendar atribuição
 - do processador e
 - de muitos outros recursos
- do sistema computacional

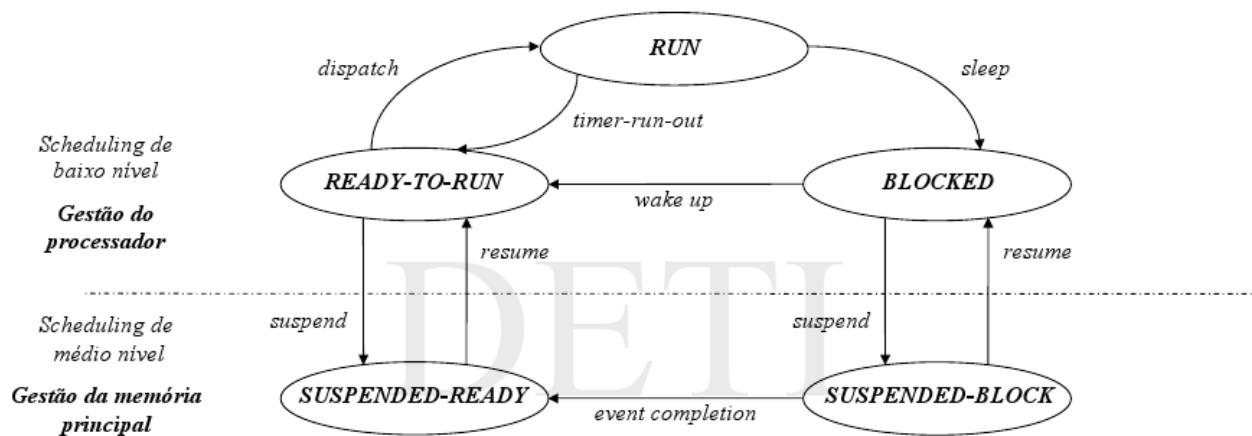
Como a memória principal é finita, quando o número de processos a correr concorrentemente é muito elevado, a memória torna-se um factor limitativo.

Assim cria-se em memória de massa uma extensão à memória principal – área de swapping – que funciona como uma região secundária de armazenamento.

Assim, liberta-se memória principal para a execução de processos – aumento da taxa de utilização do processador.

Surgem então dois novos estados, associados à transferência do espaço de endereçamento do processo para a área de swapping

- suspended-ready – correspondente ao estado ready-to-run;
- suspended-blocked – correspondente ao estado blocked.



- suspend – suspensão de um processo, por transferência do seu espaço de endereçamento para a área de swapping, o processo é **swapped out**;
- resume – retomada eventual da execução de um processo, por transferência do seu espaço de endereçamento para a memória principal, o processo é **swapped in**;
- event completion – o acontecimento externo que o processo aguardava, ocorreu.

Resume – swapped in
Suspend – swapped out

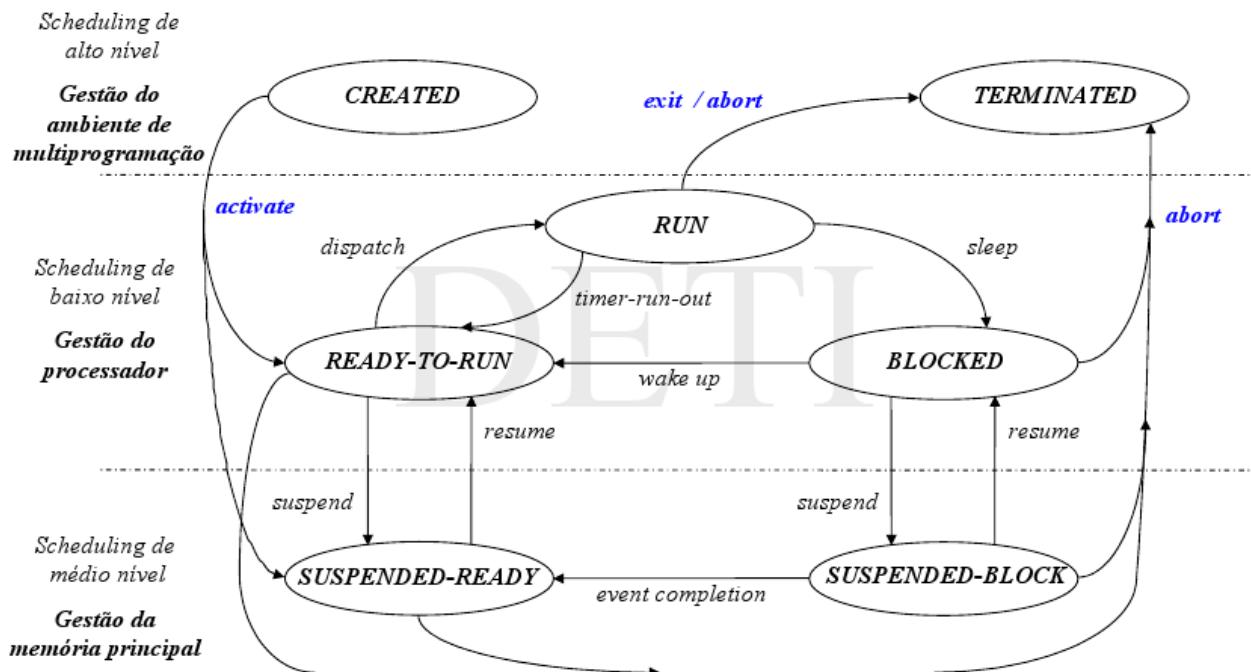
Os processos são em geral criados, têm um tempo de vida mais ou menos longo e terminam.

A aplicação desta ideia exige a inclusão de dois estados novos

- created – associado com
 - a atribuição de um espaço de endereçamento e
 - a inicialização das estruturas de dados destinadas a gerir o processo, operações que têm que ser realizadas antes que a execução possa ter lugar;
- terminated – associado com
 - a gestão da informação relativa à história da execução do processo, após a sua terminação;

esta informação pode vir a ser recolhida por programas específicos relacionados com a contabilização

- do tempo de uso
 - do processador e
 - de outros recursos do sistema computacional, e
- com a análise de desempenho.



- activate – lançamento do processo em execução; o processo pode ser colocado:
 - na fila de espera dos processos prontos a serem executados,
 - ou suspenso, em caso contrário;
 dependendo se há ou não espaço em memória principal para carregar o seu espaço de endereçamento
- exit – terminação normal do processo;
- abort – terminação anormal do processo, provocada:
 - pela ocorrência de um erro fatal
 - ou por acção de um processo com autoridade suficiente para isso.

Scheduling de alto nível – gestão do ambiente de multiprogramação

- created
- terminated

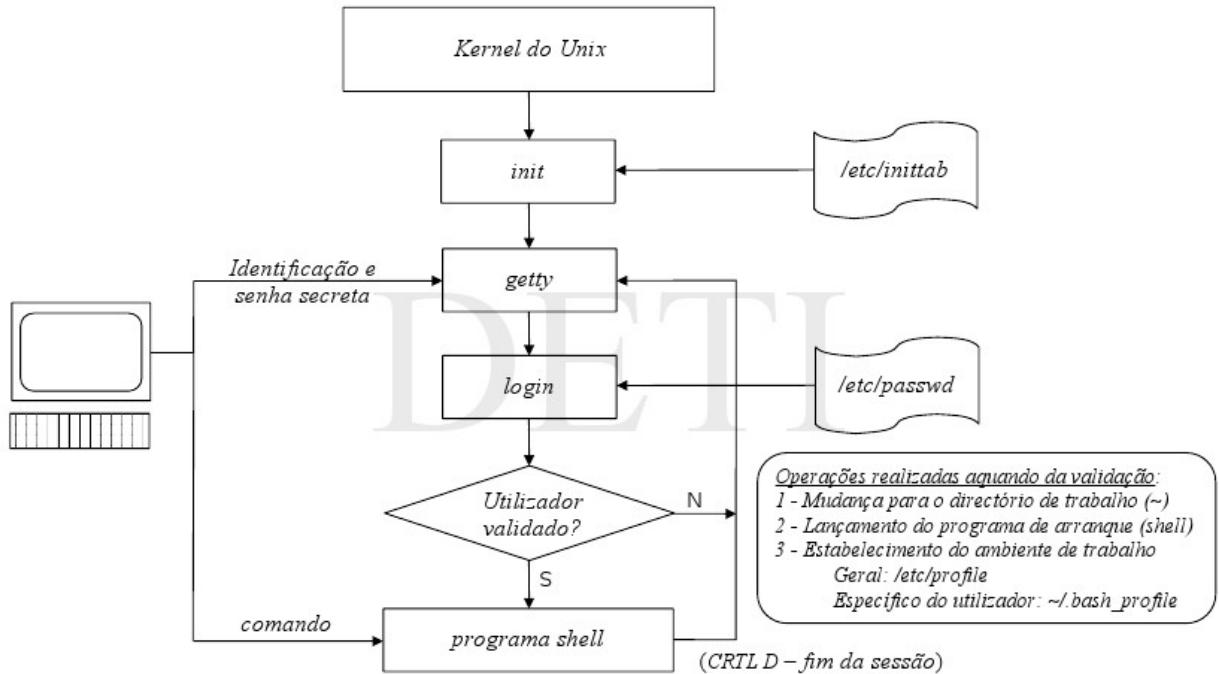
Scheduling de baixo nível – gestão do processador

- run
- ready-to-run
- blocked

Scheduling de médio nível – gestão da memória

- suspended-ready
- suspended-block

Unix – Lançamento da shell



Kernel do Unix → init (/etc/inittab) → getty → login (/etc/passwd) → Utilizador validado → (Sim) → programa shell

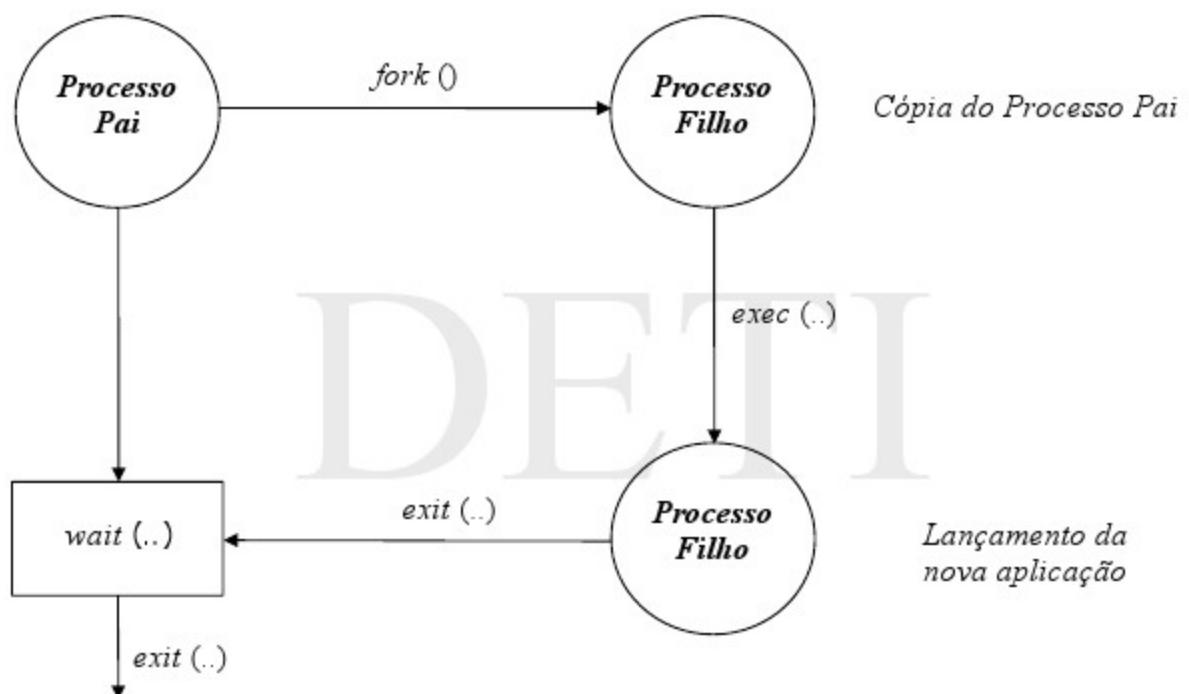
Terminal fornece:

- identificação e senha secreta ao **getty**
- comando à **shell**
- CTRL+D para terminar sessão

Utilizador validado:

- Mudança para o directório de trabalho (~)
- Lançamento do programa de arranque (shell)
- Estabelecimento do ambiente de trabalho
 - geral: /etc/profile
 - específico do utilizador: ~/.bash_profile

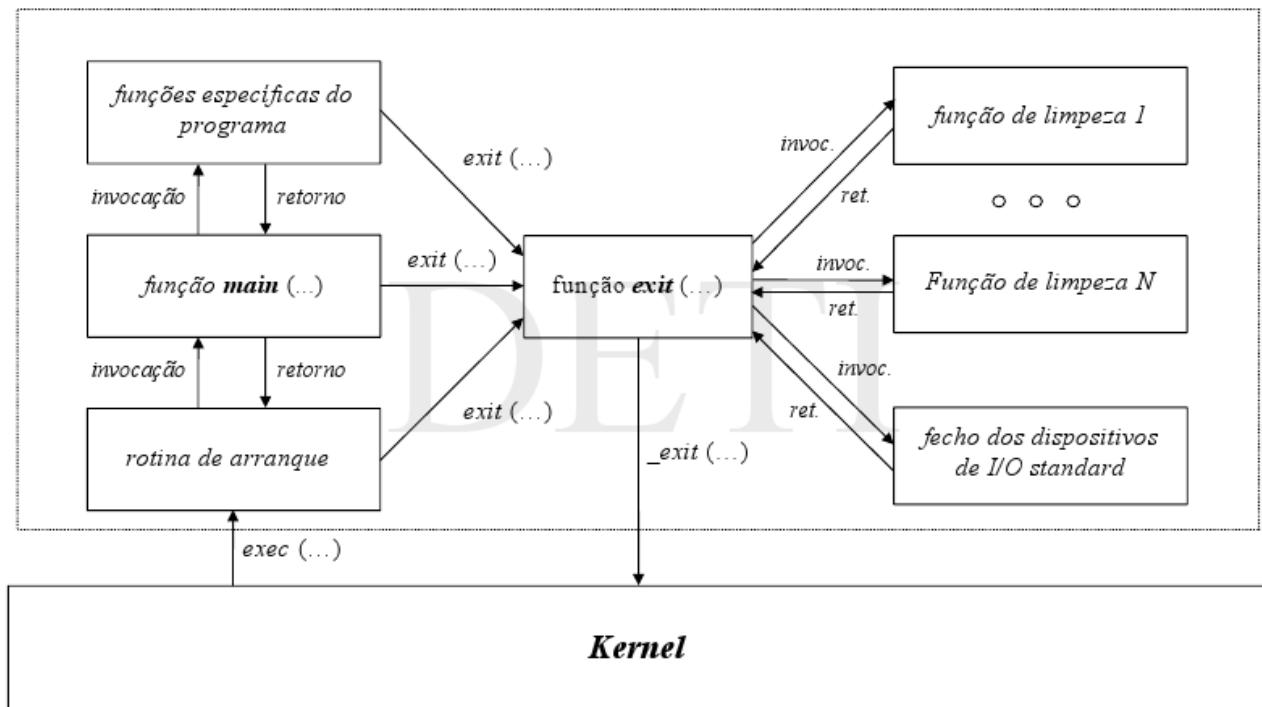
Criação de um processo em Unix



```
if (pid != 0)
    /* processo pai - aguarda a terminação do processo filho */
    { if (wait (&status) != pid)
        { perror ("erro na espera pelo processo filho");
          exit (EXIT_FAILURE);
        }
      printf ("o meu filho, com id %d, já terminou\n", pid);
      if (WIFEXITED (status))
        printf ("o seu status de saída foi %d\n", WEXITSTATUS (status));
      printf ("o meu id é %d\n", getpid ());
    }

  else /* processo filho - lança a aplicação */
  {
    if (execl (aplic, aplic) < 0)
      { perror ("erro no lançamento da aplicaccao");
        exit (EXIT_FAILURE);
      }
    exit (EXIT_SUCCESS);
  }
  exit (EXIT_FAILURE);
}
```

Ambiente de execução de um programa em C



1. Kernel – exec():

1. rotina de arranque → pode invocar exit()
 1. função main → pode invocar exit()
 1. funções específicas do programa → pode invocar exit()

2. função exit → volta a kernel (chamando _exit()) após:

1. função de limpeza 1
2. função de limpeza N
3. fecho dos dispositivos de I/O

Utilização das funções de limpeza

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

#define      OK          0

/* Variáveis localmente globais */

static int a = 0;

/* Alusão às funções de processamento de saída */

static void prep_saida_1 (void),
          prep_saida_2 (void);

void main (void)
{
    /* registo das funções de processamento de saída */

    if (atexit (prep_saida_2) != OK)
        perror ("impossível registar prep_saida_2");
        exit (EXIT_FAILURE);
    }
    if (atexit (prep_saida_1) != OK)
        perror ("impossível registar prep_saida_1");
        exit (EXIT_FAILURE);
    }

    /* trabalho útil */

    printf ("hello world!\n");

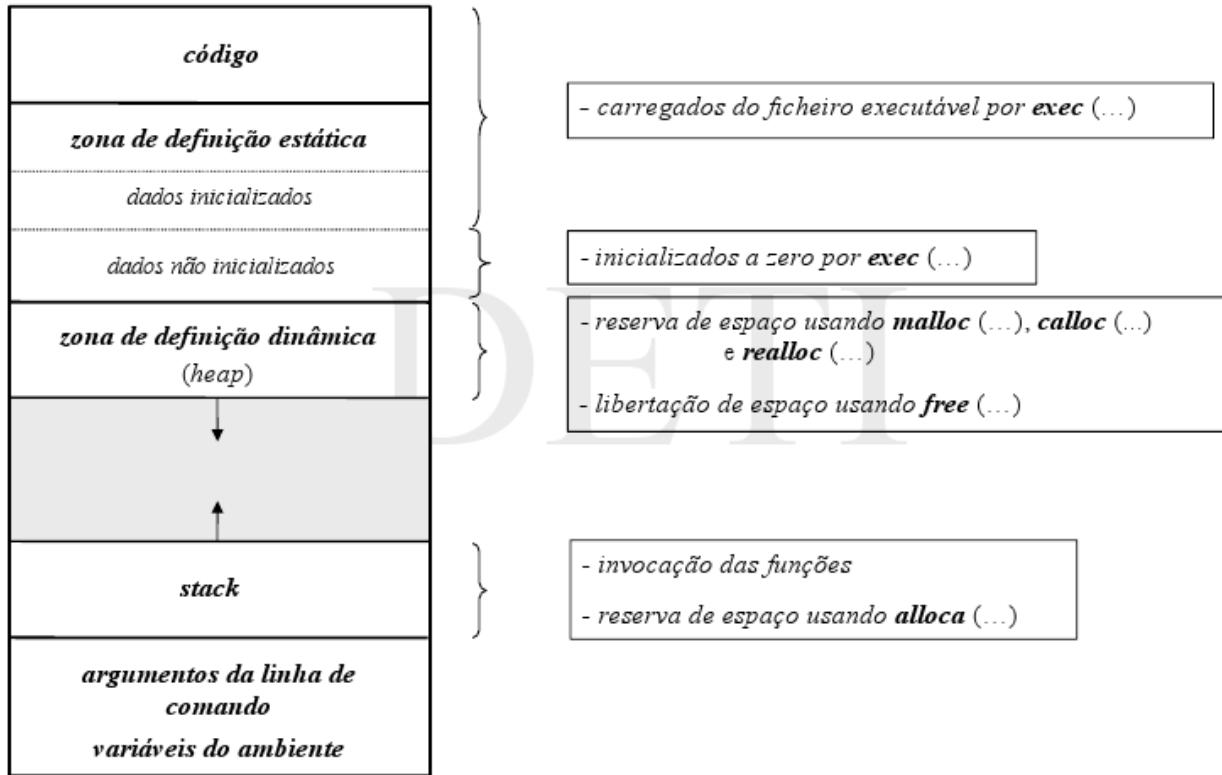
    /* terminação do programa */

    exit (EXIT_SUCCESS);
}

static void prep_saida_1 (void)
{
    printf ("saída 1: %d\n", ++a);
}

static void prep_saida_2 (void)
{
    printf ("saída 2: %d\n", ++a);
}
```

Espaço de endereçamento de um processo



- carregados do ficheiro executável por exec()
 - código
 - zona de definição estática
 - dados inicializados
- inicializados a zero por exec()
 - dados não inicializados
- reserva de espaço usando malloc(), calloc(), realloc() / libertação usando free()
 - zona de definição dinâmica (heap) – cresce para baixo
- invocação das funções / reserva de espaço usando alloca()
 - stack – cresce para cima
- coiso
 - argumentos da linha de comandos / variáveis de ambiente

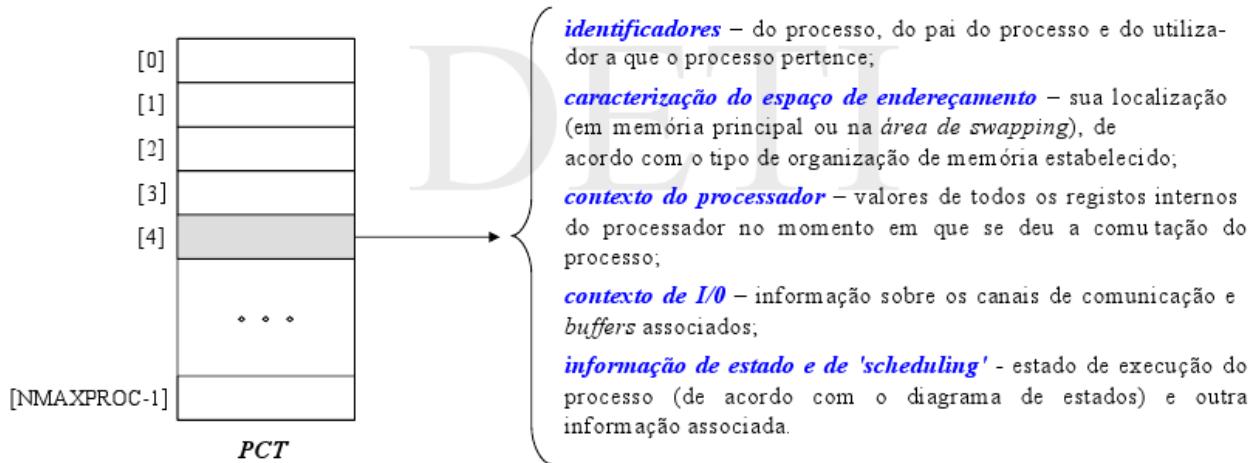
Argumentos da linha de comando e variáveis de ambiente

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

extern char **environ;           /* ponteiro para o descritor do ambiente */
void main (int argc, char *argv[])
{ unsigned int i;
  /* impressão dos diferentes campos da linha de comando */
  printf ("\n\nCampos da linha de comando:\n");
  i = 0;
  while (i < argc)
  { printf ("%s\n", argv[i]);
    i += 1;
  }
  printf ("\n");
  /* impressão das variáveis que caracterizam o ambiente */
  printf ("Variáveis que caracterizam o ambiente:\n");
  i = 0;
  while (environ[i] != NULL)
  { printf ("%s\n", environ[i]);
    i += 1;
  }
  printf ("\n");
  exit (EXIT_SUCCESS);
}
```

Tabela de controlo de processos

A implementação de um ambiente multiprogramado implica a existência de uma gama variada de informação acerca de cada processo.



Informação mantida numa tabela, Tabela de Controlo de Processos (PCT), usada intensivamente pelo *scheduler* para fazer a gestão

- do processador e
- de outros recursos do sistema computacional:

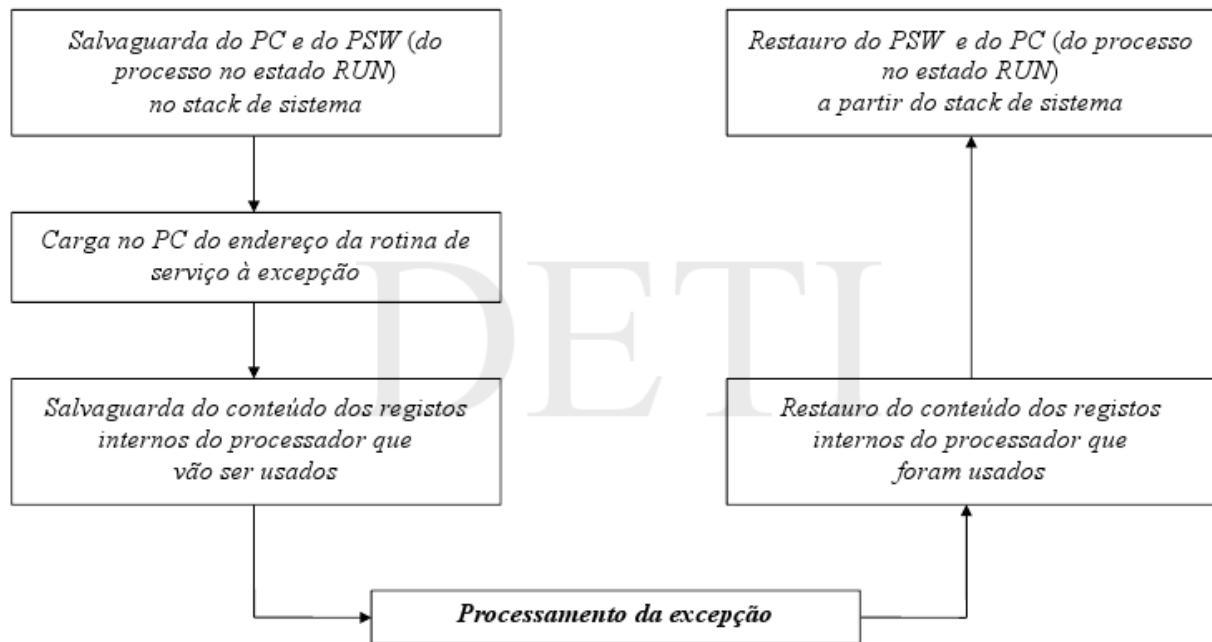
Conteúdo da PCT:

- identificadores
 - do processo,
 - do pai do processo e
 - do utilizador a que o processo pertence
- informação de estado e de 'scheduling'
 - estado de execução (de acordo com o diagrama de estados) e
 - outra informação associada
- contexto do processador
 - valores de todos os registos internos do processador (no momento em que se deu a comutação do processo)
- caracterização do espaço de endereçamento
 - localização (em memória principal ou na área de swapping)
- contexto de I/O
 - informação sobre os canais de comunicação e buffers associados

Níveis de funcionamento do processador

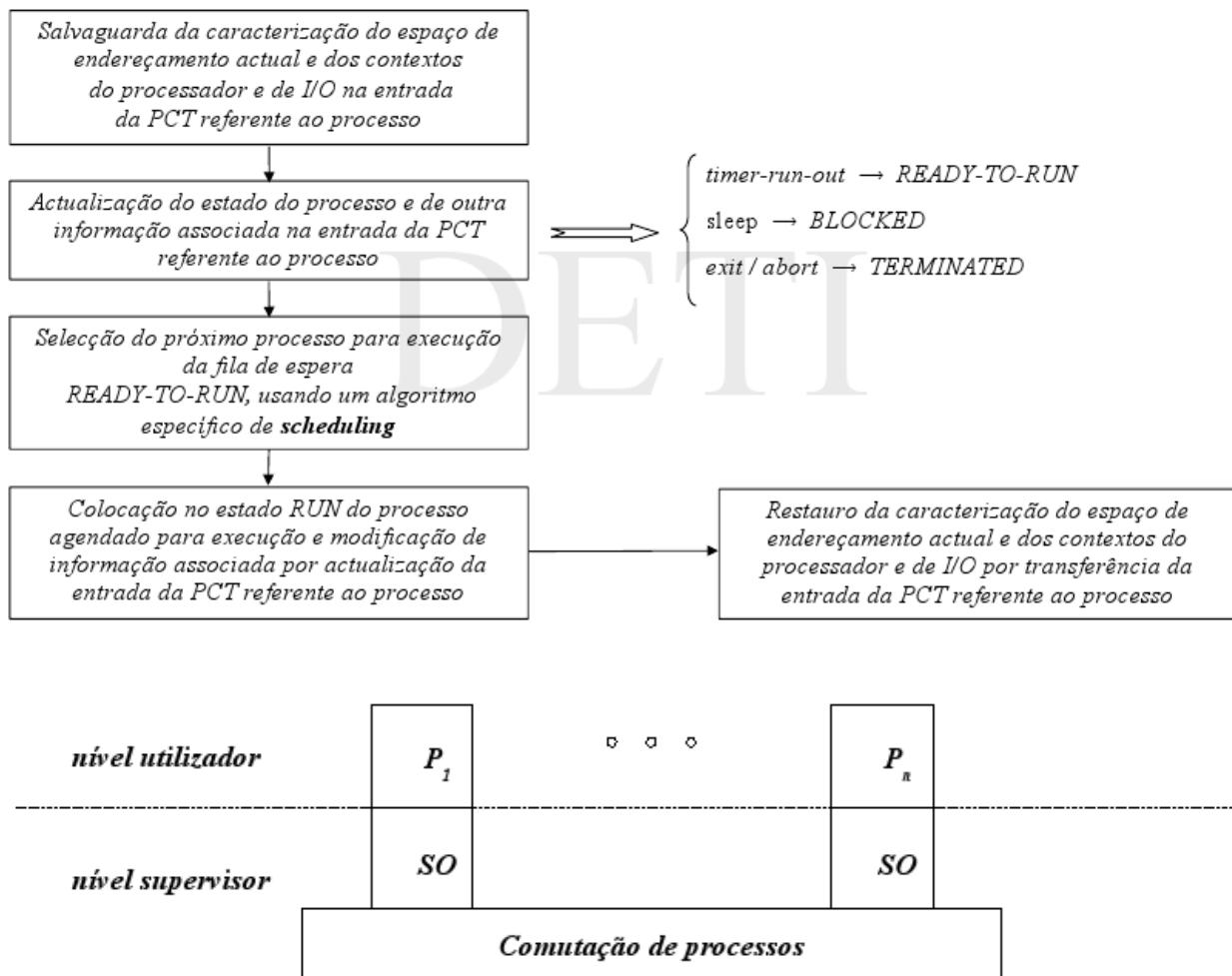
- Os processadores actuais têm basicamente dois níveis de funcionamento
 - **nível supervisor**
 - todo o conjunto de instruções do processador (*instruction set*) pode ser executado;
 - trata-se de um modo de funcionamento privilegiado: reservado para o sistema de operação;
 - **nível utilizador**
 - só uma parte do conjunto de instruções do processador pode ser executada: estão excluídas
 - as instruções de entrada / saída e
 - quase todas as instruções que permitem modificar o conteúdo dos registos da unidade de controlo;
 - constitui o modo normal de funcionamento.
- A passagem do nível supervisor para o nível utilizador efectua-se por alteração de um *bit* do *program status word*.
- Não há instruções que possibilitem directamente a passagem inversa. Ela só é realizada quando o processador processa uma **excepção**.
- Uma **excepção** é, no fundo, algo que interrompe a normal execução de instruções. Pode ser provocada por:
 - **interrupção** – um dispositivo externo;
 - **interrupção por software** – a execução de uma instrução de tipo trap;
 - **a execução de uma instrução ilegal**, ou que conduz a um erro (divisão por zero, por ex.).
- As **chamadas ao sistema**, quando não despoletadas pelo próprio hardware, são implementadas a partir de instruções trap (para que o sistema de operação funcione no modo privilegiado, com total acesso a toda a funcionalidade do processador);
- todo o processamento pode ser encarado como o serviço de excepções, neste ambiente operacional uniforme criado.
- Assim, a comutação de processos pode ser visualizada globalmente como uma vulgar rotina de serviço à excepção. Apresenta, porém, uma característica peculiar que a distingue de todas as outras: a instrução que vai ser executada, após o serviço da excepção, é, normalmente, **diferente** daquela cujo endereço foi salvaguardado ao dar-se início ao processamento da excepção.

Processamento de uma excepção genérica



- Salvaguarda do **PC** e do **PSW** (do processo no estado RUN) no stack do sistema
- **Carga no PC** do endereço da rotina de serviço à excepção
- Salvaguarda do **conteúdo dos registos internos** do processador que vão ser usados
- **Processamento da excepção**
- Restauro do **conteúdo dos registos internos** do processador que foram usados
- Restauro do **PSW e do PC** (do processo no estado RUN) a partir da stack de sistema

Comutação de processos



- Salvaguarda da
 - caracterização do espaço de endereçamento actual e
 - dos contextos
 - do processador e
 - de I/O
 na entrada da PCT referente ao processo
- Actualização
 - do estado do processo e
 - de outra informação associada
 na entrada da PCT referente ao processo
- Seleção do próximo processo para execução da fila de espera READY-TO-RUN, usando um algoritmo de *scheduling*
- Colocação no estado RUN do processo agendado para execução e
- modificação de informação associada por actualização da entrada da PCT referente ao processo
- Restauro da

- caracterização do espaço de endereçamento actual e
- dos contextos
 - do processador e
 - de I/O da entrada da PCT referente ao processo.

A maioria das funções do sistema de operação são executadas no contexto do processo utilizador que detém na altura o processador.

Quando ocorre uma interrupção produzida por um dispositivo externo, ou quando o processo executa uma **chamada ao sistema (interrupção por software)**, o processador é colocado no nível supervisor e o controlo é passado à rotina de serviço à excepção correspondente.

Note-se a importância da existência de um segundo *stack*, o chamado *stack* de sistema, para armazenar o PC e PSW do processador, quando ocorre a mudança de nível de execução, e o PC, quando se invoca rotinas no nível supervisor.

A comutação de processos é um caso à parte. Ocorre logicamente fora do contexto dos processos que coexistem.

Políticas de scheduling

Non-preemptive scheduling

- quando, após a atribuição do processador a um dado processo, este o mantém na sua posse até bloquear ou terminar.
- A transição timer-run-out não existe neste caso.
- É característico dos sistemas operativos de **tipo batch**. Porquê?

Preemptive scheduling

- quando o processador pode ser retirado ao processo que o detém,
 - tipicamente por esgotamento do intervalo de tempo de execução que lhe foi atribuído,
 - ou por necessidade de execução de um processo de prioridade mais elevada.
- É característico dos sistemas operativos de **tipo interactivo**. Porquê?

Que política de scheduling deverá ser utilizada nos sistemas de operação de tempo real?
Porquê?

Critérios a satisfazer pelos algoritmos de *scheduling*

- **eficiência** – manter o processador o mais possível ocupado com a execução dos processos dos utilizadores.
- **throughput** – maximizar o número de processos terminados por unidade de tempo;
- **deadlines** – garantir o máximo de cumprimento possível das metas temporais impostas pelos processos em execução;
- **tempo de resposta** – minimizar o tempo de resposta às solicitações feitas pelos processos interactivos;
- **tempo de turnaround** – minimizar o tempo de espera pelo completamento de um job no caso de utilizadores num sistema batch;
- **justiça** – todo o processo, ao longo de um intervalo de tempo considerado de referência, deve ter direito à sua fração de tempo de processador;
- **previsibilidade** – o tempo de execução de um processo deve ser razoavelmente constante (e independente da sobrecarga **pontual** a que o sistema computacional possa estar sujeito);

Os critérios a serem satisfeitos pelos algoritmos de *scheduling* do processador podem ser enquadrados segundo duas perspectivas

- **perspectiva sistémica**
 - **critérios orientados para o utilizador** – estão relacionados com o comportamento do sistema de operação na perspectiva dos processos ou dos utilizadores;
 - **critérios orientados para o sistema** – estão relacionados com o uso

eficiente dos recursos do sistema de operação;

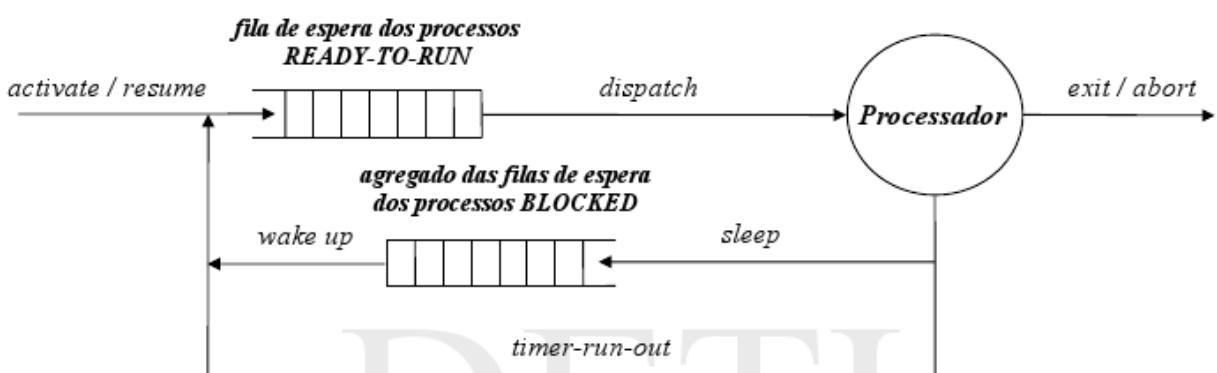
- **perspectiva comportamental**
 - **critérios orientados para o desempenho** – são quantitativos e, portanto, passíveis de serem medidos;
 - **outro tipo de critérios** – são qualitativos e difíceis de serem medidos de uma maneira directa.

Como é evidente, os critérios são em muitos casos interdependentes e não podem ser todos optimizados em simultâneo.

O desenho de uma disciplina de *scheduling* particular envolve compromissos entre requisitos contraditórios.

A natureza destes compromissos é naturalmente função do tipo e do enquadramento operacional pretendido para o sistema de operação.

Valorizando o critério de justiça



Trata-se da disciplina de atribuição mais justa. Todos os processos são colocados em pé de igualdade e são servidos por ordem de chegada.

Em políticas de *scheduling non-preemptive*, é normalmente designada pelo nome *first-come, first served* (FCFS);

Em políticas de *scheduling preemptive*, pelo nome *round robin*.

É simples de implementar, mas privilegia o tratamento dos processos CPU-intensivos sobre os processos I/O-intensivos.

Em sistemas de tipo interactivo, o intervalo de tempo da janela de execução, *time slot*, tem que ser cuidadosamente escolhido para garantir um compromisso aceitável entre o **tempo de resposta** e a **eficiência**.

Definição de prioridades

Em muitas circunstâncias concretas, considerar todos os processos em pé de igualdade não é o procedimento mais adequado. Por exemplo:

- A minimização do tempo de resposta exige, por exemplo, que seja dada preferência a processos I/O-intensivos sobre processos CPU-intensivos.
- Em sistemas de tempo real há processos cuja execução tem que ser mantida dentro de intervalos de tempo bem definidos.

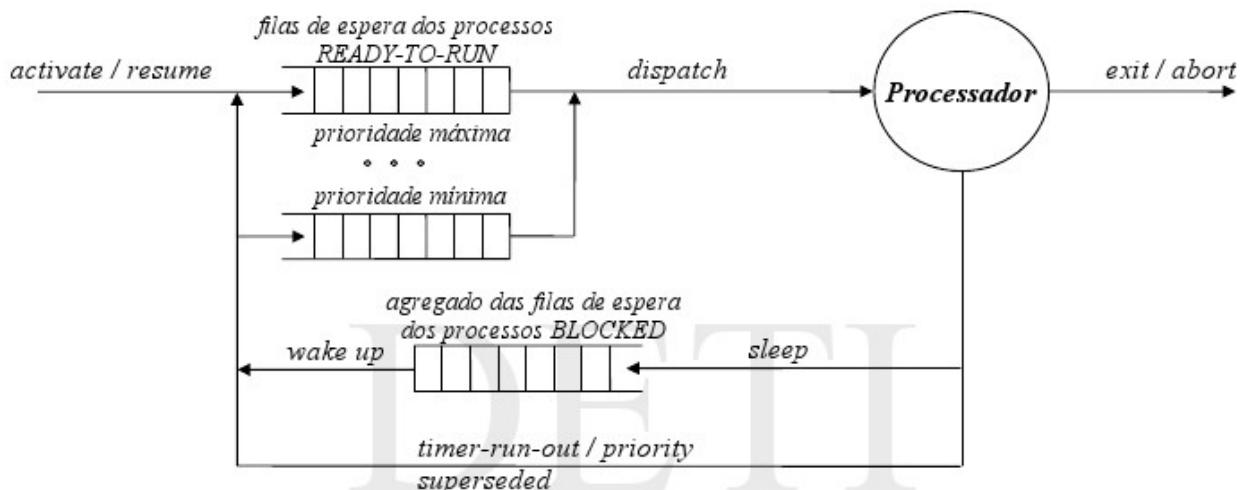
Os processos podem, por isso, ser agrupados em níveis de prioridade distinta no que respeita ao acesso ao processador. Assim, aquando da selecção do próximo processo a ser executado, o primeiro critério a ser seguido é o grau de prioridade relativa.

Processos de prioridade mais baixa só serão seleccionados quando na lista de espera READY-TO-RUN não existirem processos de prioridade mais elevada.

As prioridades podem ser de dois tipos:

- **prioridades estáticas** – quando o método de definição é determinístico
- **prioridades dinâmicas** – quando o método de definição depende da história passada de execução do processo.

Prioridade estática



Os processos são agrupados em classes de prioridade fixa, de acordo com a sua importância relativa.

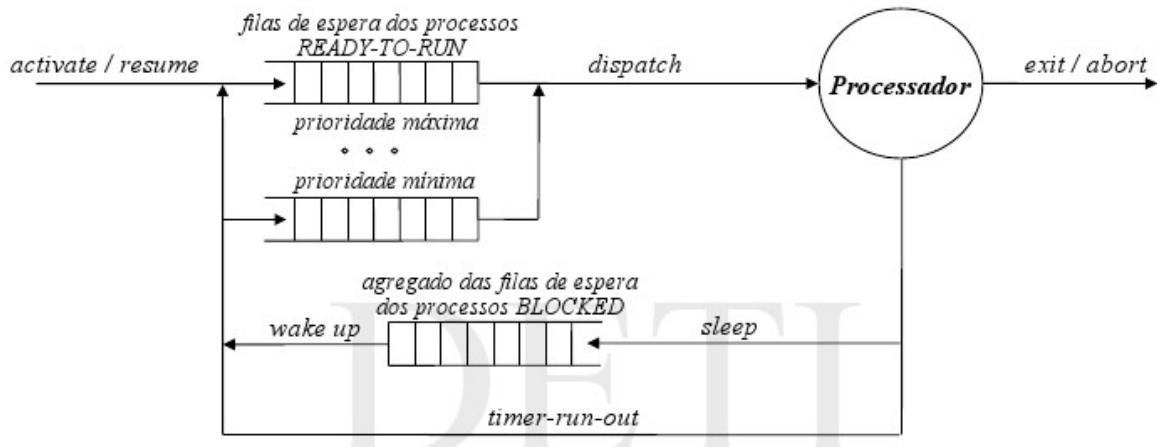
A comutação pode ocorrer sempre que seja necessário executar um processo de prioridade mais elevada.

Trata-se da disciplina de atribuição mais **injusta**, existindo um risco claro de ocorrência de **adiamento indefinido** na calendarização para execução dos processos de **prioridade**

mais baixa.

Disciplina característica dos **sistemas de tempo real**.

Nota minha: notar que o esquema acima assume **preemption**.



Nota minha: este esquema já não tem *priority superseeded*.

Aquando da sua criação, é atribuído a cada processo um dado nível de prioridade.

Funcionamento:

À medida que o processo é calendarizado para execução:

- quando esgota a janela de execução atribuída → a sua prioridade é decrementada de uma unidade;
- se bloqueia antes de esgotar a janela → incrementada de uma unidade.

sendo reposta no valor inicial quando atinge o valor mínimo

Disciplina característica de **sistemas multiutilizador**. O Unix SVR4 usa uma disciplina de scheduling deste tipo para os processos utilizador.

Prioridade dinâmica

Um método alternativo de privilegiar os processos interactivos consiste na definição de classes de prioridade com carácter funcional. Os processos transitam entre elas de acordo com a ocupação da(s) última(s) janelas de execução.

Por exemplo:

- **Nível 1 (mais prioritário): terminais** – classe para que transitam os processos após 'acordarem' no seguimento de espera por informação do dispositivo de entrada standard;
- **Nível 2: I/O genérico** – classe para que transitam os processos após 'acordarem'

no seguimento de espera por informação de um dispositivo distinto do dispositivo de entrada standard;

- **Nível 3: janela pequena** – classe para que transitam os processos quando completaram a sua última janela de execução;
- **Nível 4 (menos prioritário): janela longa** – classe para que transitam os processos quando completaram em sucessão um número pré-definido de janelas de execução; trata-se de processos CPU-intensivos, o objectivo é atribuir-se-lhes no futuro uma janela de execução de duração mais longa, mas menos vezes.

Um problema que se coloca em sistemas de tipo *batch*, é a redução do tempo de *turnaround* (somatório dos tempos de espera e de execução) dos *jobs* que constituem a fila de processamento.

Desde que se conheçam estimativas dos tempos de execução de todos os processos na fila, é possível estabelecer-se uma ordenação para a execução dos processos que minimiza o tempo médio de *turnaround* do grupo.

Com efeito, admita-se que a fila contém N *jobs*, cujas estimativas dos tempos de execução são, respectivamente, te_n , com $n = 1, 2, \dots, N$. Então, o tempo médio de *turnaround* do grupo vem dado por

$$tm_{\text{taround}} = te_1 + (N-1)/N \cdot te_2 + \dots + 1/N \cdot te_N ,$$

que é mínimo quando a ordenação dos processos é feita por ordem crescente do tempo de execução.

Este método de selecção designa-se por *shortest job first* (SJF) ou *shortest process next* (SPN).

Uma abordagem semelhante pode ser usada em sistemas interactivos para se estabelecer a prioridade dos processos que competem pela posse do processador. O princípio consiste em procurar estimar-se a fracção de ocupação da janela de execução seguinte em termos da ocupação das janelas passadas, atribuindo-se o processador ao processo para o qual esta estimativa é menor.

Seja fe_1 a estimativa da fracção de ocupação da primeira janela de execução de um dado processo e, f_1 , a fracção de ocupação efectivamente verificada. Então, a estimativa para a fracção de ocupação da próxima janela vem dada por

$$fe_2 = af_1 + (1-a)fe_1 , \quad \text{com } a \in [0, 1] ;$$

21 e, para a N -ésima janela, por

$$fe_N = af_{N-1} + (1-a)fe_{N-1} , \quad \text{com } a \in [0, 1]$$

$$= a^{N-1}fe_1 + a^{N-2}(1-a)f_1 + \dots + (1-a)f_{N-1} .$$

O parâmetro a é utilizado para controlar o grau com que a história passada de execução vai influenciar a estimativa presente.

Quando o sistema computacional tem uma carga muito grande de processos I/O-intensivos, os processos CPU-intensivos correm o risco de **adiamento indefinido** se a disciplina de *scheduling* anterior for aplicada sem qualquer correção.

Um meio de alterar a situação é incorporar no cálculo da prioridade o tempo que o processo aguarda a atribuição do processador no estado READY-TO-RUN.

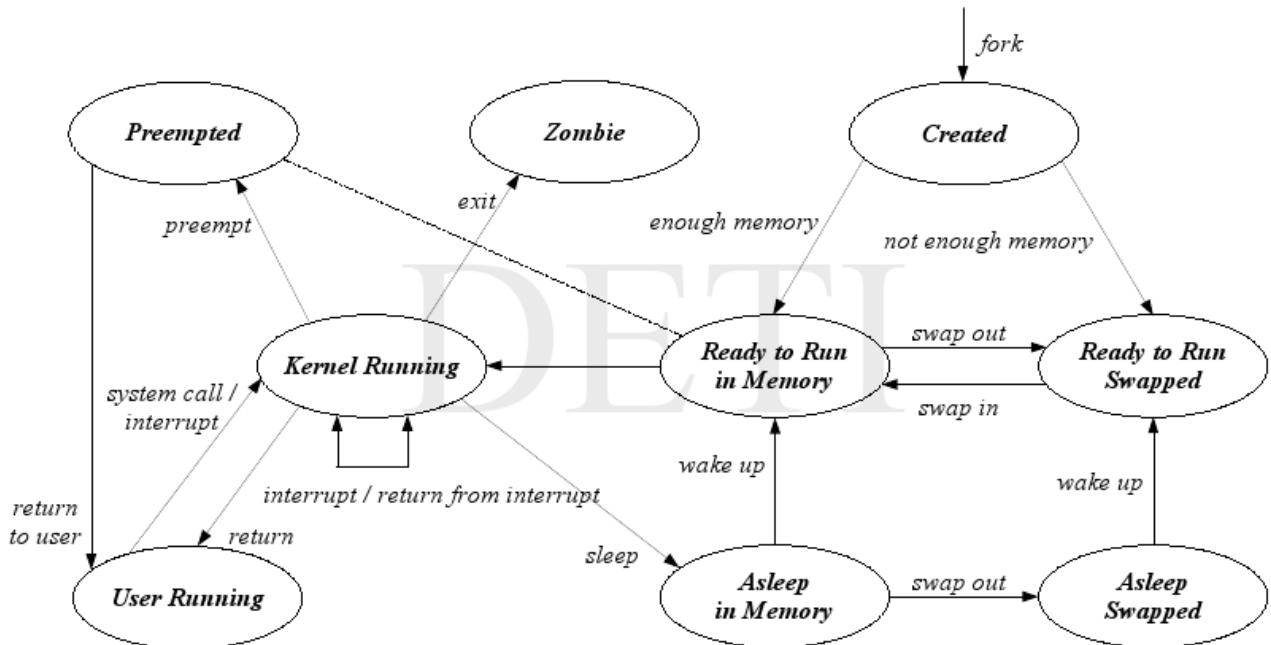
Seja R esse tempo, que é normalizado em termos da duração do intervalo de execução, então a prioridade p de cada processo pode ser definida por

$$p = \frac{1 + b \cdot R}{fe_N} ,$$

em que o parâmetro b controla o peso com que o tempo de espera afecta a definição da prioridade.

Disciplinas de *scheduling* com esta propriedade **promovem o aging dos processos**.

Diagrama de estados de um processo em Unix



- o Unix considera dois estados *run*, o *kernel running* e o *user running*, associados aos níveis do processador, supervisor e utilizador, respectivamente;
- o estado *ready-to-run* está também dividido formalmente em dois estados, *ready to run in memory* e *preempted*, embora no fundo eles formem o mesmo estado, como é indiciado pela linha tracejada;
- sempre que um processo utilizador sai do nível supervisor, o *scheduler* tem a possibilidade de calendarizar para execução um processo de prioridade mais alta, fazendo então transitar o processo actual para o estado *preempted* (em termos práticos, contudo, os processos nos estados *ready to run in memory* e *preempted* são colocados nas mesmas filas de espera e são, por isso, tratados de forma idêntica pelo *scheduler*);
- de facto, é desta maneira que é tratada a situação de esgotamento da janela de execução, a transição *timer-run-out* está, assim, incluída em *preempt*;

Tradicionalmente, a execução no nível supervisor não podia ser interrompida, donde resultava que o Unix não era adequado para processamento em tempo real;

Nas versões actuais do sistema de operação, nomeadamente a partir do SVR4, o problema foi resolvido dividindo o código numa sucessão de regiões atómicas entre as quais as estruturas de dados internas estão garantidamente num estado seguro e permitem, portanto, que a execução seja interrompida;

Surge então uma nova transição, estabelecida entre os estados *preempted* e *kernel running*, que pode ser designada de *return to system*.

Scheduling em Linux

- o Linux considera três grandes classes de *scheduling*:
 - **SCHED_FIFO** – classe formada por processos cuja atribuição do processador só lhes é retirada quando processos da mesma classe, com prioridade mais alta, estão prontos a serem executados (*priority superseded*);
 - **SCHED_RR** – classe formada por processos cuja atribuição do processador está condicionada a uma janela de execução, a atribuição do processador é-lhes retirada mais cedo quando processos da classe **SCHED_FIFO**, ou da mesma classe com prioridade mais alta, estão prontos a serem executados (*priority superseded*);
 - **SCHED_OTHER** – classe formada pelos processos restantes, o processador só é atribuído a processos desta classe se não houver outro tipo de processos prontos a serem executados;
- as classes **SCHED_FIFO** e **SCHED_RR** estão associadas a processamento de tempo real e a processos de sistema e o valor das suas prioridades é **fixo**;
- a classe **SCHED_OTHER** está associada aos processos utilizador;
- para a classe **SCHED_OTHER**, o Linux usa um algoritmo baseado em créditos no estabelecimento da sua prioridade;
- no instante de recreditação i , a prioridade de cada processo (equivalente ao número de créditos de execução que lhe são atribuídos) é calculada pela fórmula seguinte

$$CPU_j(i) = \frac{CPU_j(i-1)}{\tau} + PBase_j + nice_j$$

em que $CPU_j(i)$ representa a prioridade do processo j (o número de créditos que lhe são atribuídos) no instante de recreditação i , $CPU_j(i-1)$ o número de créditos não usados pelo processo j no intervalo de recreditação $i-1$, $Pbase_j$ a prioridade base do processo j e $nice_j$ o valor de alteração de prioridade dependente do utilizador (valor no intervalo -20 a 19);

- funcionamento:
 - o *scheduler* calendariza para execução o processo com mais créditos (maior prioridade)
 - sempre que ocorre uma interrupção do RTC o processo perde um crédito;
 - quando o número de créditos atinge o valor zero, o processo perde a posse do processador por esgotamento da janela de execução e outro processo é calendarizado;

- quando já não há processos na fila de espera dos processos prontos a serem executados com créditos não nulos, procede-se a uma nova operação de recreditação que envolve todos os processos da classe, mesmo aqueles que estão bloqueados;
- o algoritmo de *scheduling* combina, assim, dois factores, a história passada de execução do processo e a sua prioridade, e maximiza o tempo de resposta dos processos I/O-intensivos sem produzir adiamento indefinido para os processos CPU-intensivos.

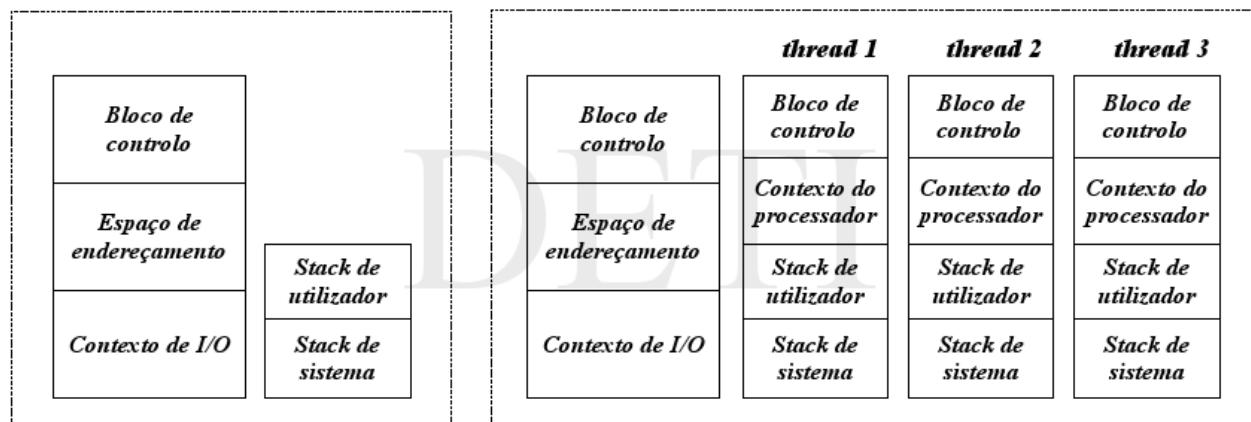
Processos vs. Threads

O conceito de processo corporiza as propriedades seguintes

- **posse de recursos**
 - um espaço de endereçamento próprio e
 - um conjunto de canais de comunicação com os dispositivos de entrada / saída;
- **fio de execução (thread)**
 - um *program counter* que sinaliza a localização da instrução que deve ser executada a seguir
 - um conjunto de registos internos do processador que contêm os valores actuais das variáveis em processamento e
 - um *stack* que armazena a história de execução.

Estas propriedades, embora surjam reunidas num processo, podem ser tratadas separadamente pelo sistema de operação. Quando tal acontece, os processos dedicam-se a agrupar um conjunto de recursos e os *threads*, também conhecidos por *light weight processes*, constituem entidades executáveis independentes dentro do contexto de um mesmo processo.

Multithreading representa então a situação em que é possível criar-se *threads* múltiplos de execução no contexto de um processo.



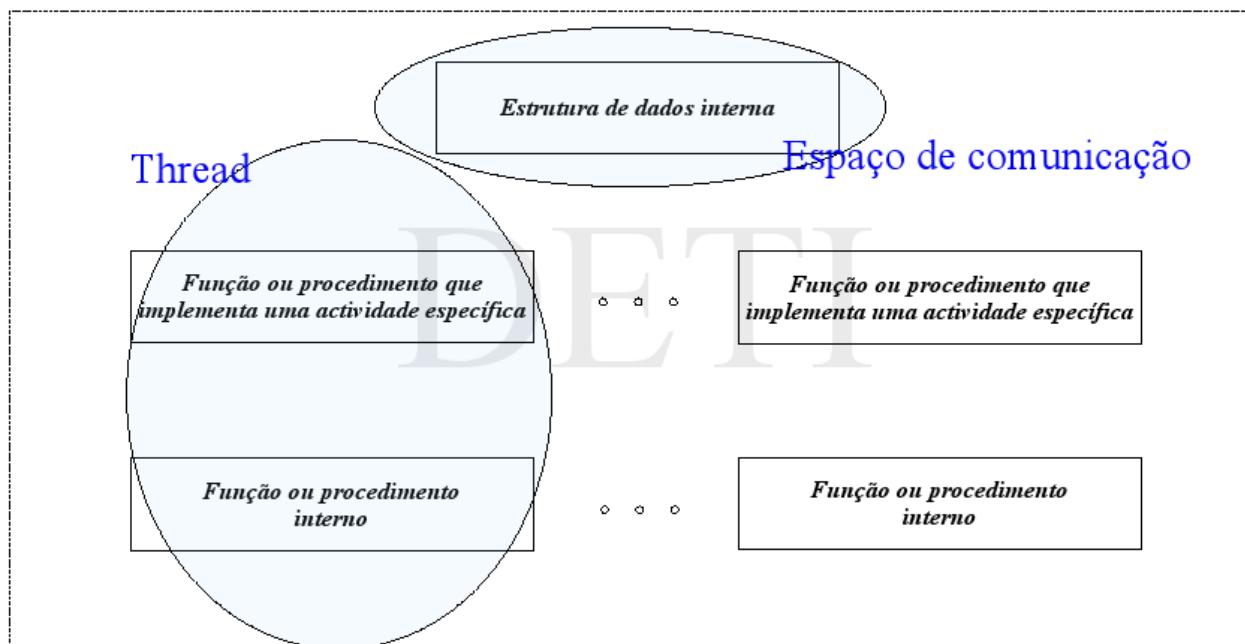
Single threading

Multithreading

Vantagens de um ambiente multithreaded

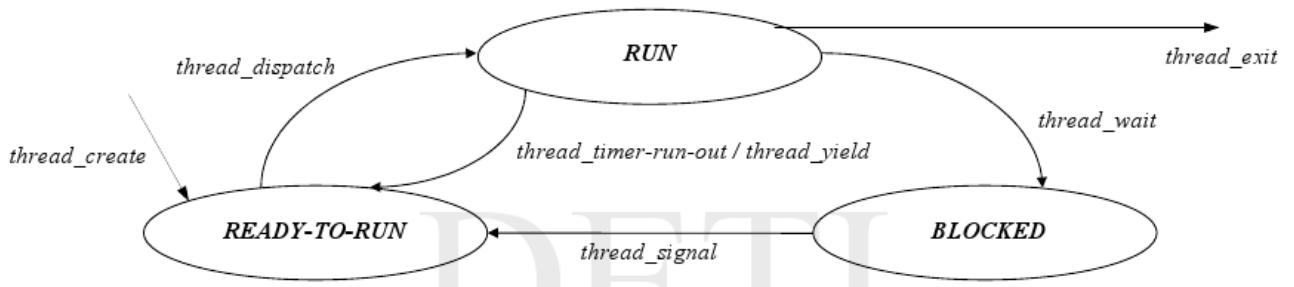
- **maior simplicidade na decomposição da solução** – programas que envolvem múltiplas actividades são mais fáceis de conceber numa perspectiva concorrencial do que numa perspectiva sequencial;
- **melhor gestão de recursos do sistema computacional** – torna-se mais simples gerir a ocupação da memória principal e o acesso aos dispositivos de entrada / saída de uma maneira eficaz, devido a haver uma partilha do espaço de endereçamento e do contexto de I/O entre os *threads* em que uma aplicação é dividida;
- **eficiência e velocidade de execução**
 - operações como a criação e destruição de *threads* e a mudança de contexto, tornam-se menos pesadas e, portanto, mais eficientes;
 - em multiprocessadores simétricos torna-se possível executar em paralelo múltiplos *threads* da mesma aplicação, aumentando assim a sua velocidade de execução.

Organização de um programa multithreaded



- cada *thread* está tipicamente associado à execução de uma função que implementa uma actividade específica;
- a comunicação entre os múltiplos *threads* é materializada pelo acesso à estrutura de dados interna, que é global, e onde está definido um espaço de partilha de informação em termos de variáveis e de canais de comunicação com os dispositivos de entrada / saída;
- o programa principal constitui o primeiro *thread* a ser criado e, tipicamente, o último *thread* a ser concluído.

Diagrama de estados de um thread



Descrição rápida

- `thread_create`
 - `thread_dispatch`
 - `thread_exit`
 - `thread_timer-run-out / thread_yield`
 - `thread_wait`
 - `thread_signal`
-
- o diagrama contém apenas os estados correspondentes à gestão do processador (*scheduling* de baixo nível);
 - a gestão da memória principal (*scheduling* de médio nível) é um problema que se coloca apenas ao nível do processo (visto que o espaço de endereçamento é partilhado)
 - a gestão do ambiente de multiprogramação (*scheduling* de alto nível) tem sobretudo a ver com as restrições impostas ao número máximo de *threads* que podem coexistir no âmbito de um processo.

Suporte à implementação de um ambiente multithreaded

user threads

os *threads* são implementados por uma biblioteca específica ao *nível utilizador* que fornece apoio à

- criação,
- gestão e
- *scheduling* de threads

sem interferência do kernel;

isto significa que a sua implementação é muito eficiente, mas, como o kernel vê apenas o processo a que eles pertencem, quando um *thread* particular executa uma chamada ao sistema bloqueante, todo o processo é bloqueado, mesmo que existam *threads* que estejam prontos a serem executados;

kernel threads

os *threads* são implementados directamente ao *nível do kernel* que providencia as operações de

- criação,

- gestão e
- scheduling de threads;

a sua implementação é menos eficiente do que no caso anterior, mas o bloqueio de um *thread* particular **não** afecta a calendarização para execução dos restantes e torna-se possível a sua execução paralela num multiprocessador.

Threads em Linux

O Linux lida com a questão da implementação de *threads* de um modo muito artifioso

fork – cria um novo processo a partir dum já existente por cópia integral do seu contexto alargado:

- contexto do processador
- espaço de endereçamento e
- contexto de I/O.

clone, que cria um novo processo a partir de um já existente por cópia apenas do seu contexto restrito (**contexto do processador**), partilhando

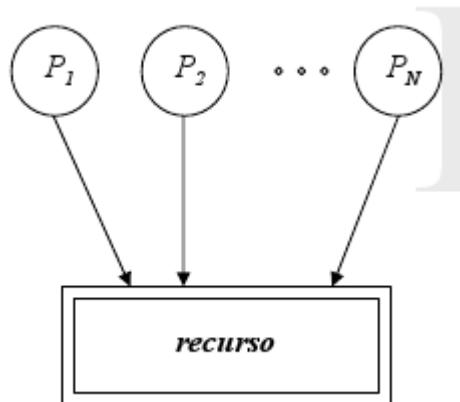
- o espaço de endereçamento e
- o contexto de I/O e

iniciando a sua execução pela invocação de uma função que é passada como parâmetro.

Assim, não há distinção efectiva entre processos e *threads*, que o Linux designa indiferentemente de *tasks*, e eles são tratados pelo *kernel* da mesma maneira.

Comunicação entre Processos

Princípios gerais

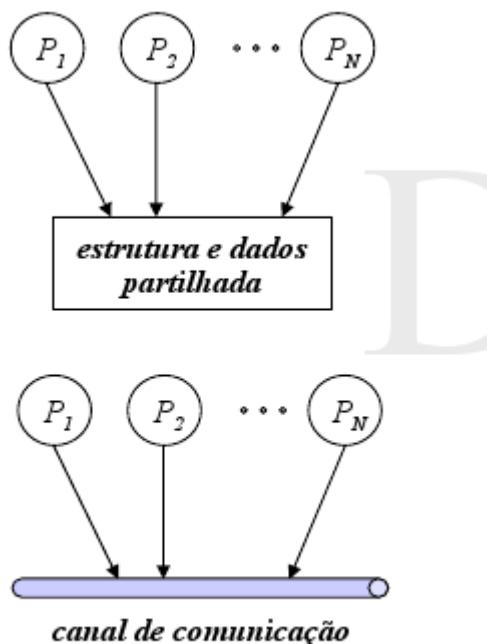


Num ambiente multiprogramado, os processos que coexistem podem ter comportamentos diversos em termos de interacção:

Processos independentes – quando são criados, têm o seu 'tempo de vida' e terminam sem interactarem de um modo explícito:

- a interacção que ocorre é **implícita** devido à competição pelos recursos do sistema computacional:
 - a atribuição dos recursos tem de ser feita forma controlada para que não haja perda de informação (responsabilidade do SO);
 - em geral, só um processo de cada vez pode ter acesso ao recurso (**exclusão mútua**).

Processos cooperantes – quando partilham informação ou comunicam entre si de um modo **explícito**:



- A partilha exige um espaço de endereçamento comum;
- A comunicação pode ser feita tanto através
 - da partilha de um espaço de endereçamento
 - ou de um canal de comunicação
- o canal de comunicação é tipicamente um recurso do sistema computacional, logo o acesso a ele está enquadrado na competição por acesso a um recurso comum;
- o acesso à região partilhada tem de ser feito de uma forma controlada para que não haja perda de informação (**responsabilidade dos processos envolvidos**)
- em geral, impõe **exclusão mútua** no acesso.

Região crítica: código executado de forma a evitar condições de corrida que conduzem à perda de informação, ou seja, execução por parte do processador do **código de acesso** a um recurso [ou região partilhada] A.K.A. “acesso por parte de um processo a um recurso [ou região partilhada]”



A imposição de exclusão mútua no acesso a um recurso [ou região partilhada], pode ter, pelo seu carácter restritivo, duas consequências indesejáveis

- **deadlock** – dois ou mais processos ficam a aguardar eternamente o acesso às **regiões críticas** respectivas, esperando acontecimentos que nunca irão acontecer: o resultado é o bloqueio das operações;
- **adiamento indefinido** – quando o acesso por parte de um processo a uma região crítica é sucessivamente adiado, devido a uma conjunção de circunstâncias fora do seu alcance, ficando efectivamente “bloqueada” a sua continuação [notar as aspas]

Relação de competição por um recurso

```
/* processos que competem pelo recurso - pid = p, p = 0, 1, ..., N-1 */  
void process(unsigned int p)  
{  
    forever  
    { qualquer_coisa();  
        acesso_ao_recurso(p); → { entrada_em_RC(p);  
            outra_coisa_qualquer();  
            manipulação_do_recurso();  
            saída_de_RC(p); }  
    }  
}
```

região crítica

Relação de partilha de dados

```
/* estrutura de dados partilhada */  
shared DATA d;  
  
/* processos que competem pelo recurso - pid = p, p = 0, 1, ..., N-1 */  
void process(unsigned int p)  
{  
    forever  
    { qualquer_coisa();  
        acesso-aos_dados_partilhados(p); → { entrada_em_RC(p);  
            outra_coisa_qualquer();  
            manipulação_dos_dados();  
            saída_de_RC(p); }  
    }  
}
```

região crítica

Relação produtor-consumidor



Processos produtores

Processos consumidores

```

/* área de comunicação: memória FIFO */
shared FIFO *fid;

/* processos produtores - pid = p, p = 0, 1, ..., N-1 */
void process(unsigned int p)
{
    DATA val;
    BOOLEAN transf_done;

    forever
    { produz_valor(&val);
        transf_done = FALSE;
        do
        { entrada_em_RC (p);
            if (! fifo_full(fid))
            { fifo_in(fid, val); /* armazenamento */
                transf_done = TRUE;
            }
            saida_de_RC(p);
        } while (!transf_done);
        outra_coisa_qualquer_p();
    }
}

/* área de comunicação: memória FIFO */
shared FIFO *fid;

/* processos consumidores - pid = p, p = 0, 1, ..., N-1 */
void process(unsigned int p)
{
    DATA val;
    BOOLEAN transf_done;

    forever
    { transf_done = FALSE;
        do
        { entrada_em_RC (p);
            if (! fifo_empty(fid))
            { fifo_out(fid, &val); /* recolha */
                transf_done = TRUE;
            }
            saida_de_RC(p);
        } while (!transf_done);
        consome_valor(val);
        outra_coisa_qualquer_p();
    }
}

```

região crítica

região crítica

Acesso a uma região crítica com exclusão mútua

Propriedades desejadas [são 5]:

- **garantia** [efectiva de imposição] **de exclusão mútua** – o acesso à região crítica associada a um mesmo recurso [ou região partilhada] só pode ser permitido a um processo de cada vez, de entre todos os processos que competem pelo acesso;
- **um processo fora da região crítica não pode impedir outro de lá entrar;**
- **independência do número ou da velocidade de execução** [relativa] **dos processos** [intervenientes] – nada deve ser presumido acerca destes factores;
- **o tempo de permanência de um processo na região crítica é** [necessariamente] **finito;**
- **inexistência de adiamento indefinido** – não pode ser adiada indefinidamente a possibilidade de acesso à região crítica a qualquer processo que o requeira.

Tipo de soluções

soluções software

são soluções que supõem o recurso em última instância ao conjunto de **instruções básico** do processador [quer sejam implementadas num monoprocessador, quer num multiprocessador com memória partilhada] ou seja:

- as instruções de transferência de dados de e para a memória são de tipo **standard**:
 - leitura e
 - escritade um valor;
- a única suposição adicional diz respeito ao caso do multiprocessador, em que a tentativa de acesso simultâneo a uma mesma posição de memória por parte de diferentes processadores é necessariamente serializada por intervenção de um árbitro;

soluções hardware

são soluções que supõem o recurso a **instruções especiais** do processador para garantir a **atomicidade** na leitura e subsequente escrita de uma mesma posição de memória:

- são muitas vezes suportadas pelo próprio sistema de operação e
- podem mesmo estar integradas na linguagem de programação utilizada.

Alternância estrita

```
/* estrutura de dados de controlo */
#define R ... /* n° de processos (pid = 0, 1, ..., R-1) */

shared unsigned int vez_de_acesso = 0;

/* primitiva de entrada na região crítica */
void entrada_em_RC (unsigned int pid_proprio)
{
    while (pid_proprio != vez_de_acesso);
}

/* primitiva de saída da região crítica */
void saida_de_RC (unsigned int pid_proprio)
{
    if (pid_proprio == vez_de_acesso)
        vez_de_acesso = (vez_de_acesso + 1) % R;
}
```

Análise Crítica

A proposta anterior, ao supor a entrada na região crítica dos processos intervenientes num regime de alternância estritamente sucessiva, não constitui uma solução geral para o problema de acesso a uma região crítica com exclusão mútua.

Duas propriedades, anteriormente apresentadas como desejáveis, são violadas

- independência da velocidade de execução relativa dos processos intervenientes – o ritmo de execução desenvolve-se ao ritmo do processo que faz menos acessos por unidade de tempo à região crítica;
- um processo fora da região crítica não pode impedir outro de lá entrar - se não for a sua vez de entrada, um processo terá que aguardar, mesmo que nenhum outro processo esteja correntemente a aceder à região crítica.

Etapas da construção de uma solução

```
/* estrutura de dados de controlo */  
#define R 2 /* n° de processos (pid = 0, 1) */  
  
shared BOOLEAN p_entrou[R] = {FALSE, FALSE};  
  
/* primitiva de entrada na região crítica */  
void entrada_em_RC (unsigned int pid_proprio)  
{  
    unsigned int pid_outro = (pid_proprio + 1) % 2;  
    while (p_entrou[pid_outro]);  
    p_entrou[pid_proprio] = TRUE;  
}  
  
/* primitiva de saída da região crítica */  
void saida_de_RC (unsigned int pid_proprio)  
{  
    p_entrou[pid_proprio] = FALSE;  
}
```

Análise Crítica

Uma análise cuidada permite verificar que a exclusão mútua não é garantida em todas as circunstâncias.

Por exemplo, quando se tem em sucessão

- o processo P0 invoca a função entrada em_RC e testa a variável p_entrou[1] que tem nesse momento o valor FALSE;
- o processo P1 invoca a função entrada em_RC e testa a variável p_entrou[0] que ainda mantém o valor FALSE;
- o processo P1 altera o valor da variável p_entrou[1] para TRUE e acede à sua região crítica;
- o processo P0 altera o valor da variável p_entrou[0] para TRUE e acede à sua região crítica;

ambos os processos entram simultaneamente na região crítica!

Aparentemente, o problema resulta do facto de se proceder ao teste da variável do outro e só depois se alterar o valor da variável própria.

```

/* estrutura de dados de controlo */
#define R 2           /* nº de processos (pid = 0, 1) */

shared BOOLEAN p_quer_entrar[R] = {FALSE, FALSE};

/* primitiva de entrada na região crítica */
void entrada_em_RC (unsigned int pid_proprio)
{
    unsigned int pid_outro;
    p_quer_entrar[pid_proprio] = TRUE;
    pid_outro = (pid_proprio + 1) % 2;
    while (p_quer_entrar[pid_outro]);
}

/* primitiva de saída da região crítica */
void saida_de_RC (unsigned int pid_proprio)
{
    p_quer_entrar[pid_proprio] = FALSE;
}

```

Análise Crítica

A exclusão mútua passou a ser garantida em todas as circunstâncias, mas surgiu uma consequência perversa: possibilidade de ocorrência de deadlock.

O que acontece, por exemplo, quando se tem em sucessão

- o processo P_0 invoca a função `entrada_em_RC` e altera o valor da variável `p_quer_entrar[0]` para `TRUE`;
- o processo P_1 invoca a função `entrada_em_RC` e altera o valor da variável `p_quer_entrar[1]` para `TRUE`;
- o processo P_1 testa a variável `p_quer_entrar[0]` e, como ela tem o valor `TRUE`, fica a aguardar acesso à sua região crítica;
- o processo P_0 testa a variável `p_quer_entrar[1]` e, como ela tem o valor `TRUE`, fica a aguardar acesso à sua região crítica.

Numa situação de contenção, como esta, pelo menos um dos processos terá que recuar temporariamente para que o impasse possa ser resolvido!

```

/* estrutura de dados de controlo */
#define R 2 /* n° de processos (pid = 0, 1) */
shared BOOLEAN p_quer_entrar[R] = {FALSE, FALSE};

/* primitiva de entrada na região crítica */
void entrada_em_RC (unsigned int pid_proprio)
{
    unsigned int pid_outro (pid_proprio + 1) % 2;
    p_quer_entrar[pid_proprio] = TRUE;
    while (p_quer_entrar[pid_outro])
    { p_quer_entrar[pid_proprio] = FALSE;
        esperar_um_intervalo_de_tempo_aleatório ();
        p_quer_entrar[pid_proprio] = TRUE;
    }
}

/* primitiva de saída da região crítica */
void saída_de_RC (unsigned int pid_proprio)
{
    p_quer_entrar[pid_proprio] = FALSE;
}

```

Análise Crítica

Trata-se de uma solução quase válida e, como tal, é por vezes usada na prática. Um exemplo disso é o protocolo Ethernet que utiliza uma variante deste algoritmo para estabelecer a exclusão mútua no acesso ao canal de comunicação.

Sob o ponto de vista teórico, no entanto, é incorrecta. Pode sempre considerar-se a ocorrência de uma combinação de circunstâncias, por pouco provável que seja, que conduz inevitavelmente a situações de deadlock ou de adiamento indefinido.

Procure encontrar exemplos que ilustrem cada uma destas situações!

A situação de contenção tem que ser resolvida de uma maneira determinística e não de uma maneira aleatória.

Algoritmo de Dekker (1965)

```
/* estrutura de dados de controlo */

#define R 2 /* número de processos (pid = 0, 1) */

shared BOOLEAN p_quer_entrar[R] = {FALSE, FALSE};
shared unsigned int p_c_prioridade = 0;

/* primitiva de entrada na região crítica */

void entrada_em_RC (unsigned int pid_proprio)
{
    unsigned int pid_outro = (pid_proprio + 1) % 2;
    p_quer_entrar[pid_proprio] = TRUE;
    while (p_quer_entrar[pid_outro])
        if (pid_proprio != p_c_prioridade)
            { p_quer_entrar[pid_proprio] = FALSE;
              while (pid_proprio != p_c_prioridade);
              p_quer_entrar[pid_proprio] = TRUE;
            }
    }
}

/* primitiva de saída da região crítica */

void saida_de_RC (unsigned int pid_proprio)
{
    unsigned int pid_outro;
    pid_outro = (pid_proprio + 1) % 2;
    if (p_quer_entrar[pid_outro]) p_c_prioridade = pid_outro;
    p_quer_entrar[pid_proprio] = FALSE;
}
```

Análise Crítica

O algoritmo de Dekker usa um mecanismo de alternância para resolver o conflito resultante da situação de contenção de dois processos. Não é muito complicado demonstrar que efectivamente garante a exclusão mútua no acesso à região crítica, evita deadlock e adiamento indefinido e não presume o que quer que seja quanto à velocidade de execução relativa dos processos intervenientes.

A sua generalização a N processos não é, contudo, fácil. De facto, não se conhece qualquer algoritmo que o faça, respeitando todas as propriedades desejáveis para a solução.

O algoritmo apresentado a seguir, devido a Dijkstra (1966), não elimina o risco de adiamento indefinido. Porquê?

Algoritmo de Dijkstra (1966)

```
/* estrutura de dados de controlo */
#define R 2 /* número de processos (pid = 0, 1) */
shared unsigned int p_quer_entrar[R] = {NAO, NAO};
shared unsigned int p_c_prioridade = 0;

/* primitiva de entrada na região crítica */
void entrada_em_RC (unsigned int pid_proprio)
{
    unsigned int n;
    do
    { p_quer_entrar[pid_proprio] = INTERESSADO;
      while (pid_proprio != p_c_prioridade)
          if (p_quer_entrar[p_c_prioridade] == NAO)
              p_c_prioridade = pid_proprio;
      p_quer_entrar[pid_proprio] = DECIDIDO;
      for (n=0; n < 2; n++)
          if (n != pid_proprio && p_quer_entrar[n] == DECIDIDO) break;
    }
    while (n < 2);
}

/* estrutura de dados de controlo */
#define R ... /* número de processos (pid = 0, 1, ..., R-1) */
shared unsigned int p_quer_entrar[R] = {NAO, NAO, ..., NAO};
shared unsigned int p_c_prioridade = 0;

/* primitiva de entrada na região crítica */
void entrada_em_RC (unsigned int pid_proprio)
{
    unsigned int n;
    do
    { p_quer_entrar[pid_proprio] = INTERESSADO;
      while (pid_proprio != p_c_prioridade)
          if (p_quer_entrar[p_c_prioridade] == NAO)
              p_c_prioridade = pid_proprio;
      p_quer_entrar[pid_proprio] = DECIDIDO;
      for (n=0; n < R; n++)
          if (n != pid_proprio && p_quer_entrar[n] == DECIDIDO) break;
    } while (n < R);
}

/* primitiva de saída da região crítica */
void saida_de_RC (unsigned int pid_proprio)
{
    p_c_prioridade = (pid_proprio + 1) % R;
    p_quer_entrar[pid_proprio] = NAO;
}
```

Algoritmo de Peterson (1981)

```
/* estrutura de dados de controlo */

#define R      2      /* numero de processos (pid = 0, 1) */

shared BOOLEAN p_quer_entrar[R] = {FALSE, FALSE};
shared unsigned int ultimo;

/* primitiva de entrada na região crítica */

void entrada_em_RC (unsigned int pid_proprio)
{
    p_quer_entrar[pid_proprio] = TRUE;
    ultimo = pid_proprio;
    unsigned int pid_outro = (pid_proprio + 1) % 2;
    while ((p_quer_entrar[pid_outro]) && (ultimo == pid_proprio));
}

/* primitiva de saída da região crítica */

void saida_de_RC (unsigned int pid_proprio)
{
    p_quer_entrar[pid_proprio] = FALSE;
}
```

Análise Crítica

O algoritmo de Peterson usa a seriação por ordem de chegada para resolver o conflito resultante da situação de contenção de dois processos. Isto é conseguido forçando cada processo a escrever a sua identificação na mesma variável. Assim, uma leitura subsequente permite por comparação determinar qual foi o último que aí escreveu. De novo, não é muito complicado demonstrar que este algoritmo garante efectivamente a exclusão mútua no acesso à região crítica, evita deadlock e adiamento indefinido e não presume o que quer que seja quanto à velocidade de execução relativa dos processos intervenientes.

A sua generalização a N processos é quase directa, se se tiver em conta a analogia de formação de uma fila de espera. Existe, contudo, uma diferença subtil entre a disciplina implementada no algoritmo e a da formação de uma fila de espera convencional.

Veja se consegue descobri-la!

Extensão do algoritmo de Peterson

```
/* estrutura de dados de controlo */
#define R ... /* numero de processos (pid = 0, 1, ..., R-1) */
shared int p_quer_entrar[R] = {-1, -1, ... , -1};
shared unsigned int ultimo[R-1];

/* primitiva de entrada na região critica */
void entrada_em_RC (unsigned int pid_proprio)
{
    unsigned int i, j;
    for (i = 0; i < R-1; i++)
    { p_quer_entrar[pid_proprio] = i;
        ultimo[i] = pid_proprio;
        do
        { boolean oqe = FALSE;
            for (j = 0; j < R; j++)
                if (j != pid_proprio) oqe = oqe || (p_quer_entrar[j] >= i);
            } while (oqe && (ultimo[i] == pid_proprio));
    }
}

/* primitiva de saída da região critica */
void saida_de_RC (unsigned int pid_proprio)
{
    p_quer_entrar[pid_proprio] = -1;
}
```

Instruções de activação e de inibição das interrupções

Sistemas Computacionais Monoprocessador

Dado que a comutação de processos num ambiente de multiprogramação é, sempre que existe *preemption*, despoletada por acção de um dispositivo externo:

- **relógio de tempo real (RTC)** – quando se esgota a janela de execução que foi atribuída, originando a transição *timer-run-out*;
- **controlador associado a um dispositivo específico** – que acorda um processo de prioridade mais elevada, originando a transição *priority superseded*;

ela é condicionada por **interrupções ao processador**.

Assim, o acesso com exclusão mútua a uma região crítica é garantido se as interrupções ao processador forem inibidas, antes do início de execução do código respectivo, sendo reposta a situação no final.

Esta abordagem, contudo, só é válida em termos do *kernel*.

A sua generalização aos processos utilizador conduziria facilmente ao bloqueio completo do sistema:

- um *bug*, que provocasse um ciclo infinito dentro da região crítica,
- ou que pusesse o processador a aguardar uma transacção de entrada / saída que ainda não tivesse ocorrido, impediria qualquer tipo de recuperação;
- além disso, um utilizador de má fé poderia sempre apropriar-se sem grande esforço de todos os recursos do sistema.

Sistemas Computacionais Multiprocessador com Memória Partilhada

Sendo aqui a contenção estabelecida entre processadores distintos, o recurso à inibição e activação das interrupções não tem qualquer efeito.

Instruções de tipo read-modify-write

Idealmente, se na função *entrada_em_RC()* fosse possível garantir a atomicidade nas leitura e escrita sucessivas da variável ocupado, o acesso com exclusão mútua a uma região crítica poderia ser implementado com uma construção do tipo abaixo.

Este tipo de construção designa-se habitualmente de *flag de locking*.

```

/* dados de controlo */
shared bool ocupado = FALSE;

/* entrada na região crítica */
void entrada_em_RC (bool
ocupado)
{
    bool estava_ocupado;
    do
    { estava_ocupado = ocupado;
      ocupado = TRUE;
    } while (estava_ocupado);
}

```

operação atómica (a sua execução não pode ser interrompida)

A implementação de uma *flag* de *locking* não é possível com as instruções convencionais de um processador, porque estas apenas efectuam a leitura ou a escrita de uma posição de memória na mesma instrução.

Os processadores actuais têm uma instrução especial, designada normalmente por *test-and-set (tas)*, que permite em sucessão [de uma maneira atómica portanto] a leitura de uma posição de memória, a afectação do registo de status em função do seu conteúdo e a escrita na mesma posição de um valor diferente de zero.

Assumindo que, quando a região crítica não está a ser acedida, a posição de memória reservada para a variável ocupado tem o valor zero (FALSE), tem-se então que

```

void entrada_em_RC (BOOLEAN * end_ocupado)
{
    lea    ad_reg, end_ocupado
    loop: tas  (ad_reg)          }
          bnz   loop
}

```

↔ { **while** (!lock) (end_ocupado); }

Busy waiting

Um problema comum às soluções *software* e ao uso de flags de *locking*, implementadas a partir de instruções de tipo *read-modify-write*, é que os processos intervenientes aguardam entrada na região crítica no estado activo – *busy waiting*.

Este facto é indesejável em sistemas computacionais monoprocessador, já que conduz a **perda a eficiência**

a atribuição do processador a um processo que pretende acesso a uma região crítica, associada a um recurso [ou região partilhada] em que um segundo processo se encontra na altura no seu interior, faz com que o intervalo de tempo de atribuição do processador se esgote sem que qualquer trabalho útil tenha sido realizado;

constrangimentos no estabelecimento do algoritmo de scheduling

numa política *preemptive* de *scheduling* onde os processos que competem por um mesmo

recurso [ou partilham uma mesma região de dados] têm prioridades diferentes, existe o risco de **deadlock** se for possível ao processo de mais alta prioridade interromper a execução do outro.

BLA BLA BLA BLA:

Assim, torna-se conveniente procurar soluções em que um processo bloqueie quando é impedido de entrar na região crítica respectiva.

Em sistemas computacionais multiprocessador com memória partilhada, e mais concretamente no caso de multiprocessamento simétrico, o problema de *busy waiting* não é tão crítico.

Quando a execução do código das regiões críticas tem uma duração relativamente curta, a alternativa de bloquear o processo enquanto aguarda uma oportunidade de acesso à região crítica, exige uma mudança de contexto para calendarizar outro processo para execução nesse processador e pode tornar-se, por isso, **menos eficiente**.

É neste contexto que as *flags* de *locking* se tornam importantes, sendo também referidas na literatura pelo nome de *spinlocks* (o processo gira em torno da variável enquanto aguarda o *locking*).

Semáforos

```
/* estrutura de dados de controlo */
#define R      . . . /* número de processos (pid = 0, 1, ..., R-1) */

shared unsigned int acesso = 1;

/* primitiva de entrada na região crítica */
void entrada_em_RC (unsigned int pid_proprio)
{
    if (acesso == 0) sleep(pid_proprio);
    else acesso -= 1; } } operação atómica

/* primitiva de saída da região crítica */
void saída_de_RC (unsigned int pid_proprio)
{
    if (há processos bloqueados) wake_up_one();
    else acesso += 1; } } operação atómica
```

O recurso directo a primitivas de tipo *sleep* e *wake up* não resolve por si só o problema. Continua a ser necessário garantir a atomicidade das operações!

Um semáforo é um dispositivo de sincronização, originalmente inventado por Dijkstra, que pode ser concebido como uma variável do tipo

```
typedef struct
{
    unsigned int val; /* valor de contagem */
    NODE *queue;      /* fila de espera dos processos bloqueados */
} SEMAPHORE;
```

sobre a qual é possível executar as duas operações atómicas seguintes

sem_down

- se o campo val for não nulo, o seu valor é decrementado;
- caso contrário, o processo que executou a operação é bloqueado e a sua identificação é colocada na fila de espera queue;

sem_up

- se houver processos bloqueados na fila de espera queue, um deles é acordado (de acordo com uma qualquer disciplina previamente definida);
- caso contrário, o valor do campo val é incrementado.

Um semáforo só pode ser manipulado desta maneira e é precisamente para garantir isso que toda e qualquer referência a um semáforo particular é sempre feita de uma forma indirecta.

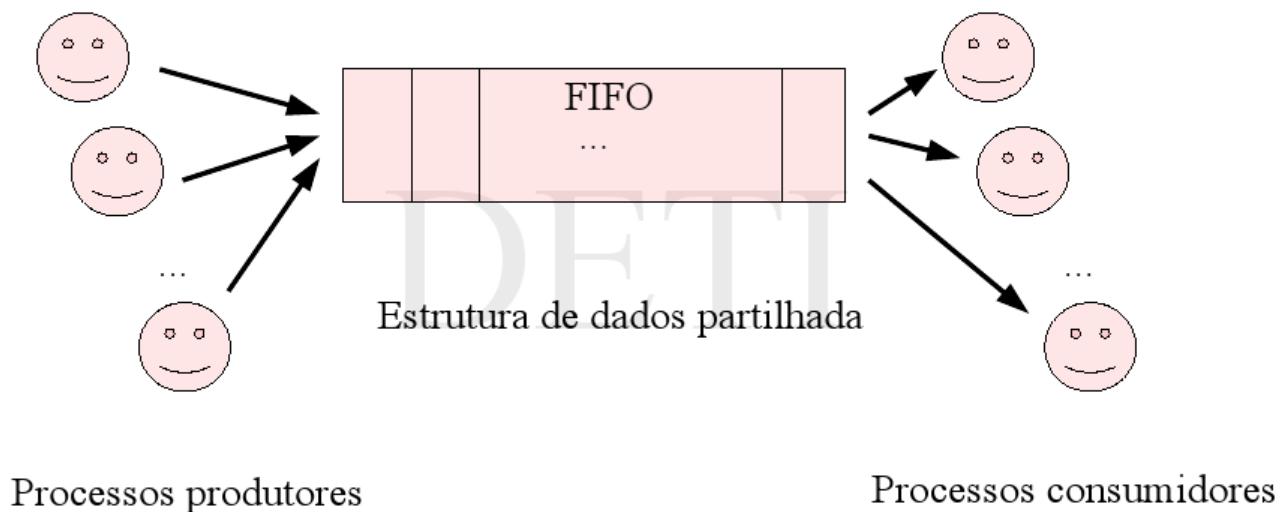
```
/* array de semáforos definidos no kernel */
#define R ... /* número de semáforos - semid = 0, 1, ..., R-1 */

static SEMAPHORE sem[R];

/* operação down */
void sem_down (unsigned int semid)
{
    inibição das interrupções;
    if (sem[semid].val == 0) sleep_on_sem (getpid(), semid);
    else sem[semid].val -= 1;
    activação das interrupções;
}

/* operação up */
void sem_up (unsigned int semid)
{
    inibição das interrupções;
    if (há processos bloqueados em sem[semid]) wake_up_one_on_sem (semid);
    else sem[semid].val += 1;
    activação das interrupções;
}
```

Problema dos produtores / consumidores (semáforos)



```

/* estrutura de dados de comunicação - memória de tipo FIFO de tamanho K */
shared FIFO *fid;

/* estrutura de dados de controlo */
shared unsigned int acesso,      /* id. do semáforo de acesso - campo val = 1 */
                  pos_vazias,   /* id. do semáforo de sincronização dos processos produtores - campo val = K */
                  pos_ocupadas; /* id. do semáforo de sincronização dos processos consumidores - campo val = 0 */

/* processos produtores - p = 0, 1, ..., N-1 */
void produtor (unsigned int p)
{
    DATA val;
    forever
    { produz_valor (&val);
        sem_down (pos_vazias);          /* aguardar a existência de posições vazias */
        sem_down (acesso);             /* entrada na região crítica */
        fifo_in (fid, val);           /* armazenamento */
        sem_up (acesso);              /* saída da região crítica */
        sem_up (pos_ocupadas);        /* sinalizar a existência de posições ocupadas */
        outra_coisa_qualquer_p ();
    }
}

/* estrutura de dados de comunicação - memória de tipo FIFO de tamanho K */
shared FIFO *fid;

/* estrutura de dados de controlo */
shared unsigned int acesso,      /* id. do semáforo de acesso - campo val = 1 */
                  pos_vazias,   /* id. do semáforo de sincronização dos processos produtores - campo val = K */
                  pos_ocupadas; /* id. do semáforo de sincronização dos processos consumidores - campo val = 0 */

/* processos consumidores - c = N, N+1, ..., N+M-1 */
void consumidor (unsigned int c)
{
    DATA val;
    forever
    { sem_down (pos_ocupadas);       /* aguardar a existência de posições ocupadas */
        sem_down (acesso);           /* entrada na região crítica */
        fifo_out (fid, &val);       /* recolha */
        sem_up (acesso);            /* saída da região crítica */
        sem_up (pos_vazias);         /* sinalizar a existência de posições vazias */
        consome_valor (val);
        outra_coisa_qualquer_c ();
    }
}

```

NOTA: notar o uso de dois semáforos aparentemente com valores redundantes.

Crítica ao uso de semáforos

A construção de soluções concorrentes baseadas em semáforos apresenta um conjunto de vantagens e desvantagens

vantagens

- **suporte ao nível do sistema de operação** – porque a sua implementação é feita pelo *kernel*, as operações sobre semáforos estão directamente disponíveis ao programador de aplicações, constituindo-se como uma biblioteca de chamadas ao sistema que podem ser usadas em qualquer linguagem de programação;

- **universalidade** – são construções de muito baixo nível e podem, portanto, devido à sua versatilidade, ser usadas no desenho de qualquer tipo de soluções;

desvantagens

- **conhecimento especializado** – a sua manipulação directa exige ao programador um domínio completo dos princípios da programação concorrente, pois é muito fácil cometer erros que originam condições de corrida e, mesmo, deadlock.

Monitores

Conceptualmente, o principal problema com o uso dos semáforos é que eles servem simultaneamente para

- garantir o acesso com exclusão mútua a uma região crítica e
- sincronizar os processos intervenientes.

BLA BLA BLA BLA:

Assim, e porque se trata de primitivas de muito baixo nível, a sua aplicação é feita segundo uma perspectiva *bottom-up* (os processos são bloqueados antes de entrarem na região crítica, se as condições à sua continuação não estiverem reunidas) e não *top-down* (os processos entram na região crítica e bloqueiam, se as condições à sua continuação não estiverem reunidas).

A primeira abordagem torna-se logicamente confusa e muito sujeita a erros, sobretudo em interacções de alguma complexidade, porque as primitivas de sincronização podem estar dispersas por todo o programa.

Uma solução é introduzir ao nível da própria linguagem de programação uma construção [concorrente] que trate separadamente o acesso com exclusão mútua a uma dada região de código e a sincronização dos processos.

Um monitor é um dispositivo de sincronização, proposto de uma forma independente por *Hoare* e *Brinch Hansen*, que pode ser concebido como um módulo especial, suportado pela linguagem de programação [concorrente] e constituído por uma estrutura de dados interna, por código de inicialização e por um conjunto de primitivas de acesso.

```

monitor exemplo
  (* estrutura de dados interna, acesso vedado ao exterior *)
  var
    val: DATA;          (* região partilhada *)
    c: condition;   (* variável de condição para sincronização *)
  (* primitivas de acesso *)
  procedure pa1 (...);
  end (* pa1 *)
  function pa2 (...): real;
  end (* pa2 *)
  (* inicialização *)
  begin
    ...
  end
end monitor;

```

Uma aplicação escrita numa linguagem concorrente que implementa o paradigma de variáveis partilhadas, é vista como um conjunto de *threads* que competem pelo acesso a estruturas de dados partilhadas.

Quando as estruturas de dados são implementadas com monitores, a linguagem de programação garante que a execução de uma primitiva do monitor é feita em regime de exclusão mútua. Assim, o compilador, ao compilar um monitor, gera o código necessário para impor esta situação.

Um *thread* entra no monitor por invocação de uma das suas primitivas, o que constitui a única forma de acesso à estrutura de dados interna. Como a execução das primitivas decorre em regime de exclusão mútua, quando um outro *thread* está no seu interior, o *thread* é bloqueado à entrada, aguardando a sua vez.

A sincronização entre *threads* é gerida pelas variáveis de condição.

Existem duas operações que podem ser executadas sobre uma variável de condição

wait

- o *thread* que invoca a operação é bloqueado na variável de condição e
- é colocado fora do monitor para possibilitar que um outro *thread* que aguarda acesso possa prosseguir;

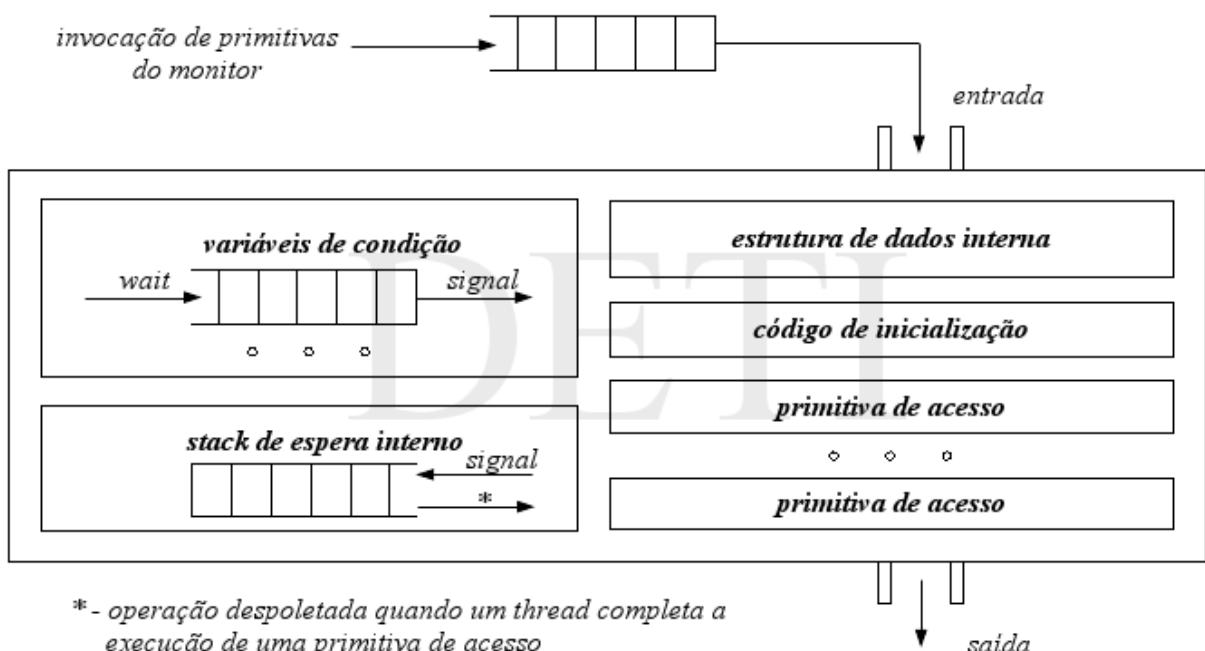
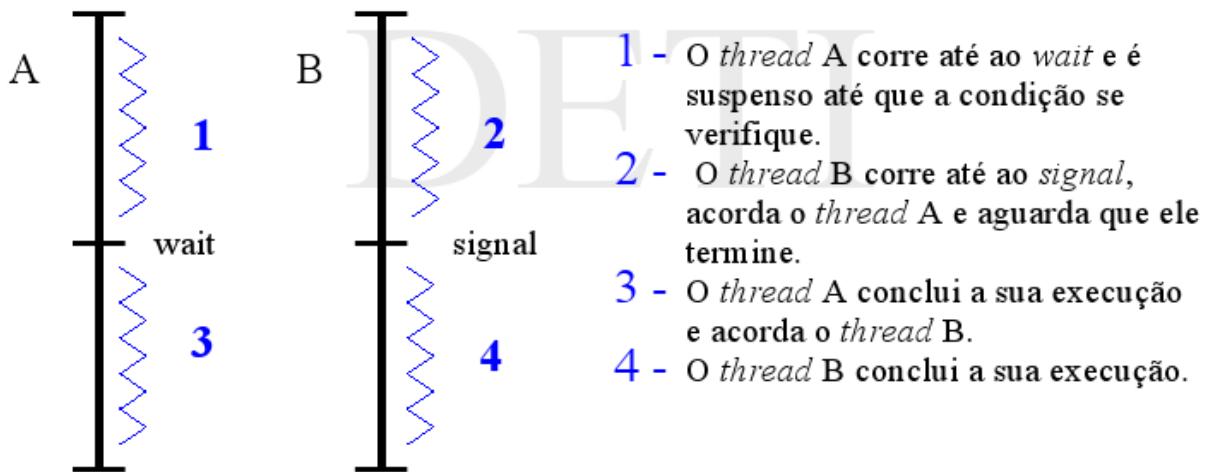
signal

- se houver *threads* bloqueados na variável de condição, um deles é acordado;
- caso contrário, nada acontece.

Para impedir a coexistência de dois *threads* dentro do monitor, é necessária uma regra que estipule como a contenção decorrente do *signal* é resolvida.

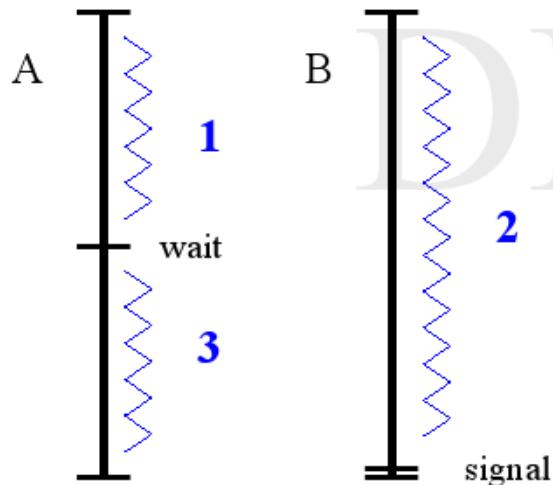
Monitor de Hoare

monitor de *Hoare* – o *thread* que invoca a operação de *signal* é colocado fora do monitor para que o *thread* acordado possa prosseguir; é muito geral, mas a sua implementação exige a existência de um *stack*, onde são colocados os *threads* postos fora do monitor por invocação de *signal*;

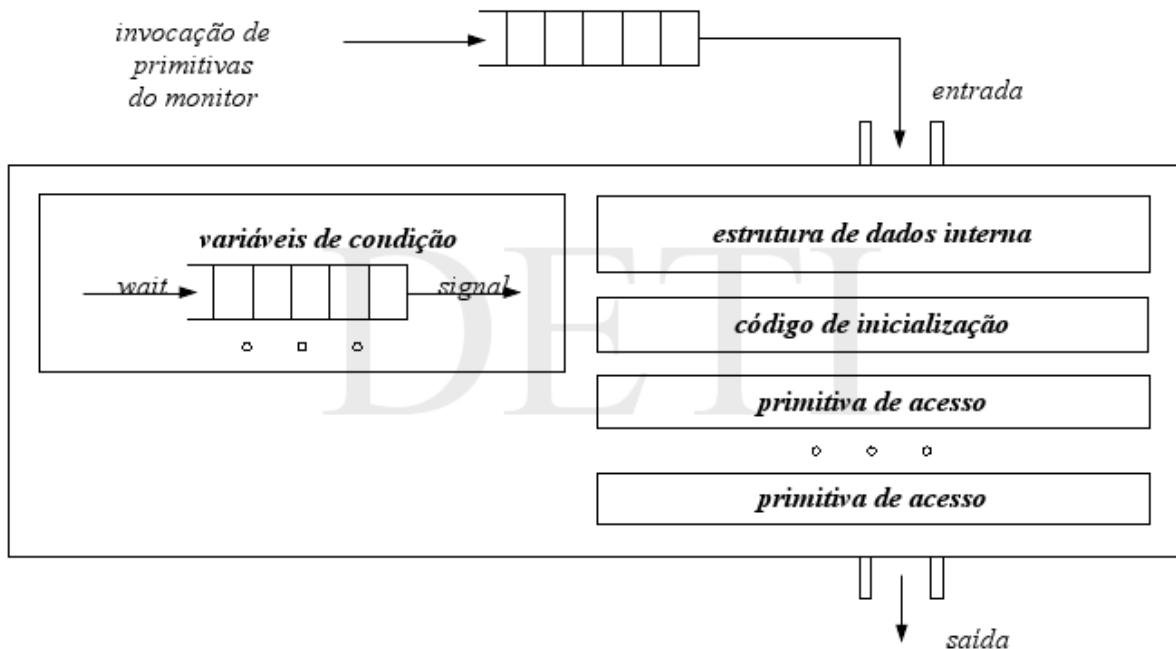


Monitor de Brinch Hansen

monitor de *Bринч Hansen* – o thread que invoca a operação de signal liberta imediatamente o monitor (*signal* é a última instrução executada); é simples de implementar, mas pode tornar-se bastante restritivo porque só há possibilidade de execução de um *signal* em cada invocação de uma primitiva de acesso;

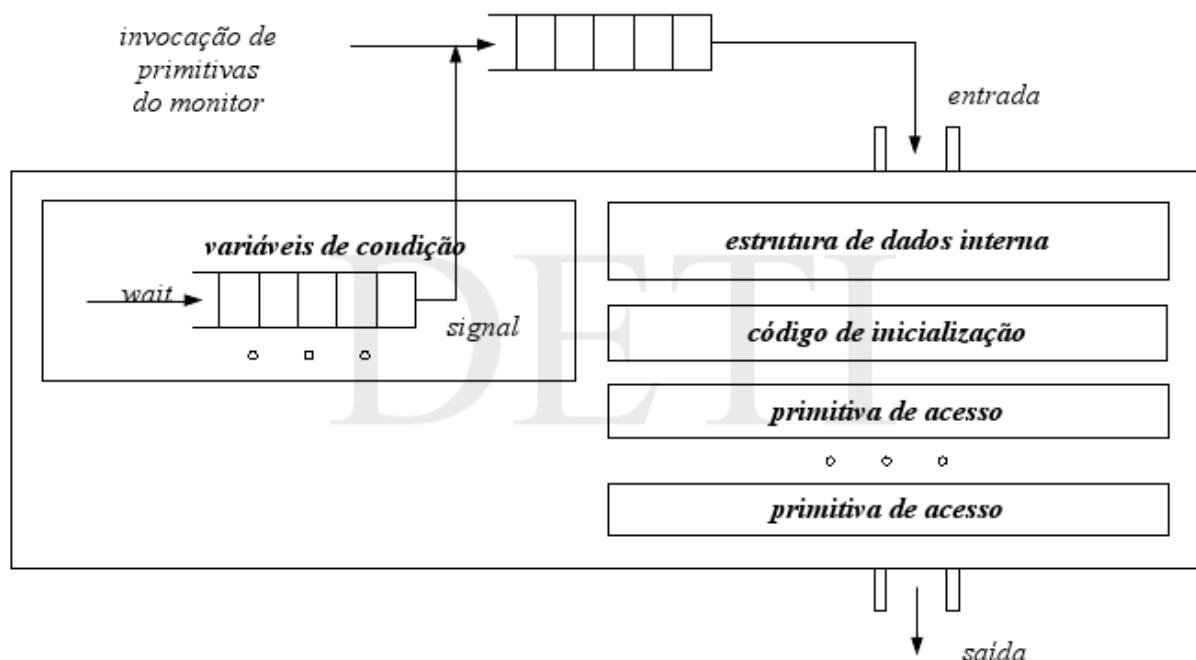
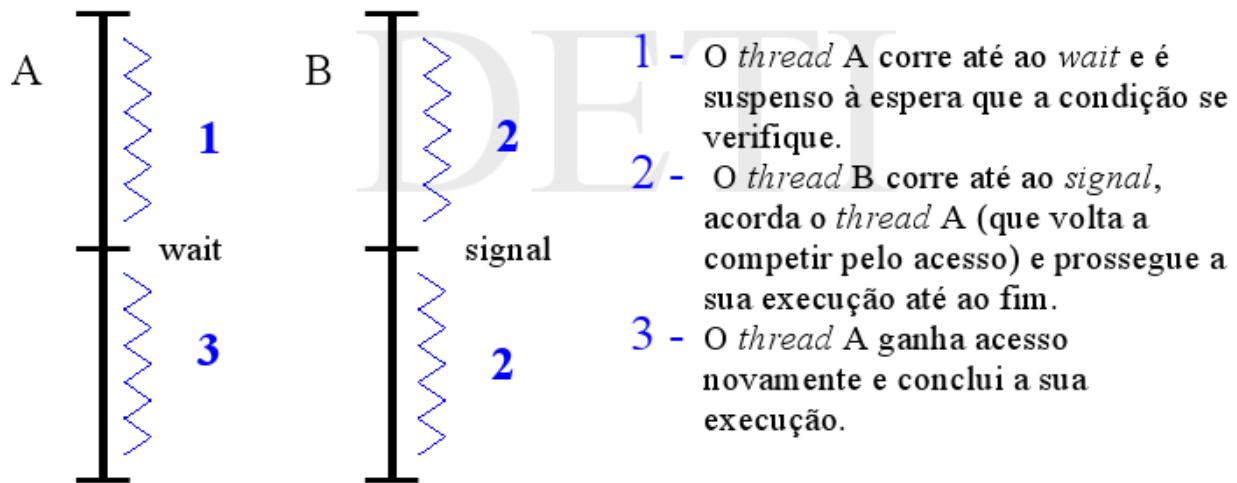


- 1 - O thread A corre até ao *wait* e é suspenso até que a condição se verifique.
- 2 - O thread B corre até ao *signal*, que deve ser a última instrução, acorda o thread A e termina.
- 3 - O thread A conclui a sua execução.



Monitor de Lampson / Redell

monitor de *Lampson / Redell* – o *thread* que invoca a operação de *signal* prossegue a sua execução, o *thread* acordado mantém-se fora do monitor e compete pelo acesso a ele; é simples de implementar, mas pode originar situações em que alguns *threads* são colocados em **adiamento indefinido**.



Problema dos produtores / consumidores (monitores)



Processos produtores

Processos consumidores

```
monitor transf;          (* monitores de Hoare e de Brinch Hansen *)
  var
    fifo: FIFO;
    n_pos_vazias: integer;
    fifo_vazio, fifo_cheio: condition;
  procedure put(val: DATA);
  begin
    if n_pos_vazias = 0 then wait(fifo_cheio);
    fifo_in(fifo, val);
    n_pos_vazias := n_pos_vazias - 1;
    signal(fifo_vazio);
  end; (* put *)
  procedure get(var val: DATA);
  begin
    if n_pos_vazias = K then wait(fifo_vazio);
    fifo_out(fifo, val);
    n_pos_vazias := n_pos_vazias + 1;
    signal(fifo_cheio);
  end; (* get *)
  begin
    n_pos_vazias := K;
  end;
end monitor; (* transf *)
```

NOTA: só temos uma variável desta vez, mas duas condições de espera.

```

monitor transf;          (* monitor de Lampson / Redell *)
  var
    fifo: FIFO;
    n_pos_vazias: integer;
    fifo_vazio, fifo_cheio: condition;
  procedure put(val: DATA);
  begin
    while n_pos_vazias = 0 then wait(fifo_cheio);
    fifo_in(fifo, val);
    n_pos_vazias := n_pos_vazias - 1;
    signal(fifo_vazio);
  end; (* put *)
  procedure get(var val: DATA);
  begin
    while n_pos_vazias = K then wait(fifo_vazio);
    fifo_out(fifo, val);
    n_pos_vazias := n_pos_vazias + 1;
    signal(fifo_cheio);
  end; (* get *)
  begin
    n_pos_vazias := K;
  end;
end monitor; (* transf *)

```

NOTA: igual ao anterior mas temos um *while* caso um *signal* seja mandado por alma e graça do Espírito Santo; não estou a ver aqui onde se vê o caso flagrante de competir pelo acesso ao monitor, a não ser que o *wait* seja mandado a mais do que uma *thread* (?).

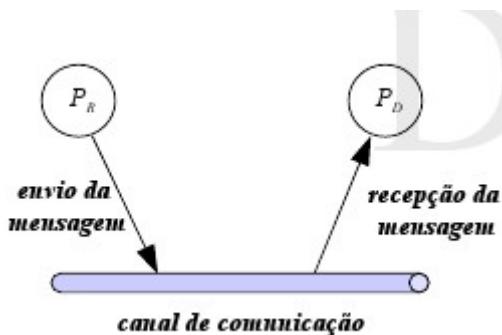
```

procedure produtor(p: integer); (* p = 0, 1, ..., N-1*)
  var
    val: DATA;
  begin
    forever
    begin
      produz_valor(val);
      transf.put(val);           (* armazenamento do valor *)
      outra_coisa_qualquer_p();
    end
  end; (* produtor *)
procedure consumidor(c: integer); (* c = N, N+1, ..., N+M-1 *)
  var
    val: DATA;
  begin
    forever
    begin
      transf.get(val);          (* recolha do valor *)
      consome_valor(val);
      outra_coisa_qualquer_c();
    end
  end; (* consumidor *)

```

Passagem de mensagens

- Uma forma alternativa de comunicação entre processos é através da troca de mensagens.
- Trata-se de um mecanismo absolutamente geral.
- Não exigindo partilha do espaço de endereçamento, a sua aplicação, de um modo mais ou menos uniforme, é igualmente válida tanto:
 - em ambientes monoprocessador, como
 - em ambientes multiprocessador,
 - ou de processamento distribuído.



O princípio em que se baseia é muito simples:

- sempre que um processo P_R , dito remetente, pretende comunicar com um processo P_D , dito destinatário, envia-lhe uma mensagem através de um canal de comunicação estabelecido entre ambos (operação de envio);
- o processo P_D , para receber a mensagem, acede ao canal de comunicação e aguardar a sua chegada (operação de recepção).

A troca de mensagens só conduzirá a uma comunicação fiável entre os processos remetente e destinatário, se for garantida alguma forma de sincronização entre eles.

A sincronização existente pode ser classificado em dois níveis

sincronização não bloqueante

quando a sincronização é da responsabilidade dos processos intervenientes:

- a operação de envio envia a mensagem e regressa sem qualquer informação sobre se a mensagem foi efectivamente recebida;
- a operação de recepção regressa independentemente de ter sido ou não recebida uma mensagem;

sincronização bloqueante

quando as operações de envio e de recepção contêm em si mesmas elementos de sincronização;

- a operação de envio envia a mensagem e bloqueia até que esta seja efectivamente recebida;
- a operação de recepção só regressa quando uma mensagem tiver sido recebida;

a sincronização bloqueante divide-se ainda em dois tipos

rendez-vous

quando, só após uma sincronização prévia entre os processos interlocutores, a transferência da mensagem tem efectivamente lugar; não exige um armazenamento intercalar da mensagem e é típica de canais de comunicação **dedicados** (ligações ponto a ponto);

remota

quando a operação de envio envia a mensagem e bloqueia, aguardando confirmação de que a mensagem foi efectivamente recebida pelo destinatário; pode existir ou não armazenamento intercalar da mensagem e é típica de canais de comunicação **partilhados**.

/ operação de envio não bloqueante */*

```
void msg_send_nb(unsigned int did, MESSAGE msg);
```

/ operação de recepção não bloqueante */*

```
void msg_receive_nb(unsigned int sid, MESSAGE *msg, bool *msg_arr);
```

/ operação de envio bloqueante */*

```
void msg_send(unsigned int did, MESSAGE msg);
```

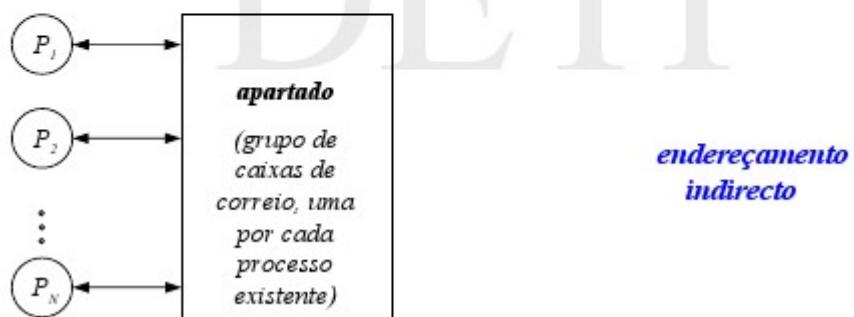
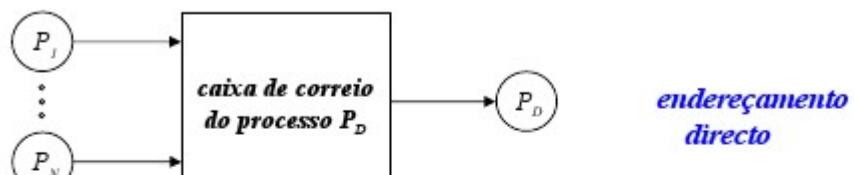
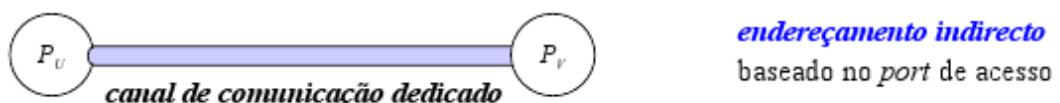
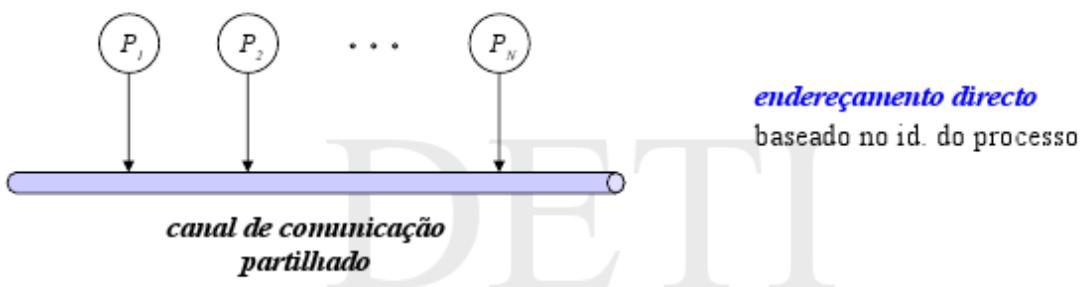
/ operação de recepção bloqueante */*

```
void msg_receive(unsigned int sid, MESSAGE *msg);
```

Para que a mensagem possa ser encaminhada entre o processo remetente e o processo destinatário, é fundamental que as operações de envio e de recepção tenham um parâmetro que faça de algum modo referência à identidade do processo interlocutor

- Se essa referência é explícita, o endereçamento diz-se **directo**.
- Caso contrário, o endereçamento diz-se **indirecto** e o que é referenciado explicitamente é o canal de comunicação.

Além disso, o canal de comunicação pode permitir o armazenamento intercalar da mensagem, tipicamente em estruturas de dados partilhadas que implementam filas de espera onde a ordem cronológica de chegada é em princípio respeitada – as chamadas **caixas de correio**.



Formato da Mensagem

cabeçalho

- identificação do remetente
- identificação do destinatário
- tipo
- informação de controlo
n.º da mensagem
comprimento do conteúdo informativo

conteúdo informativo

- comprimento fixo ou variável
(geralmente estruturado segundo um tipo de dados definido pelos processos envolvidos na comunicação)

Problema dos produtores / consumidores (pass. de mens.)

```
/* supõe-se existir uma caixa de correio, com capacidade para K mensagens, previamente criada */
static unsigned int com; /* identificador da caixa de correio */

/* estrutura de dados da mensagem trocada */
typedef struct
{
    DATA info;           /* só tem conteúdo informativo */
} MESSAGE;

void produtor(unsigned int p) /*p = 0, 1, ..., N-1*/
{
    MESSAGE msg;

    forever
    { produz_valor(&msg.info);
        msg_send_nb(com, msg); /* envia a mensagem e regressa imediatamente,
                               se a caixa de correio estiver cheia bloqueia */
        outra_coisa_qualquer_p();
    }
}

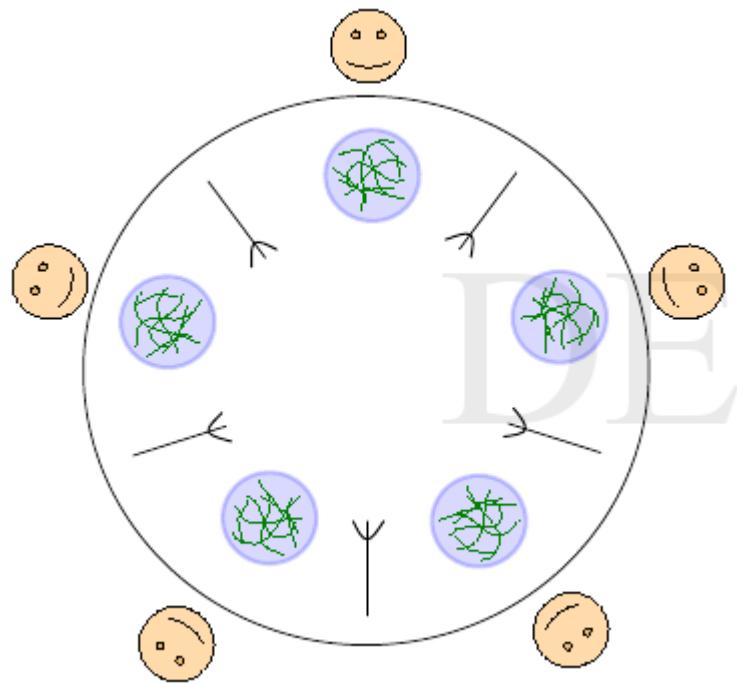
/* supõe-se existir uma caixa de correio, com capacidade para K mensagens, previamente criada */
static unsigned int com; /* identificador da caixa de correio */

/* estrutura de dados da mensagem trocada */
typedef struct
{
    DATA info;           /* só tem conteúdo informativo */
} MESSAGE;

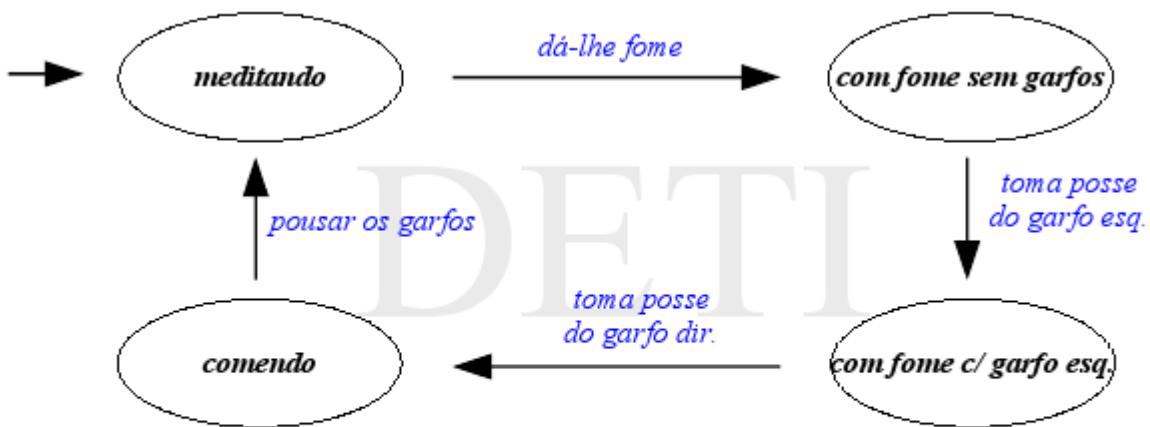
void consumidor(unsigned int c) /*c = N, N+1, ..., N+M-1*/
{
    MESSAGE msg;

    forever
    { msg_receive(com, &msg); /* aguarda a chegada de uma mensagem */
        consome_valor(msg.info);
        outra_coisa_qualquer_c();
    }
}
```

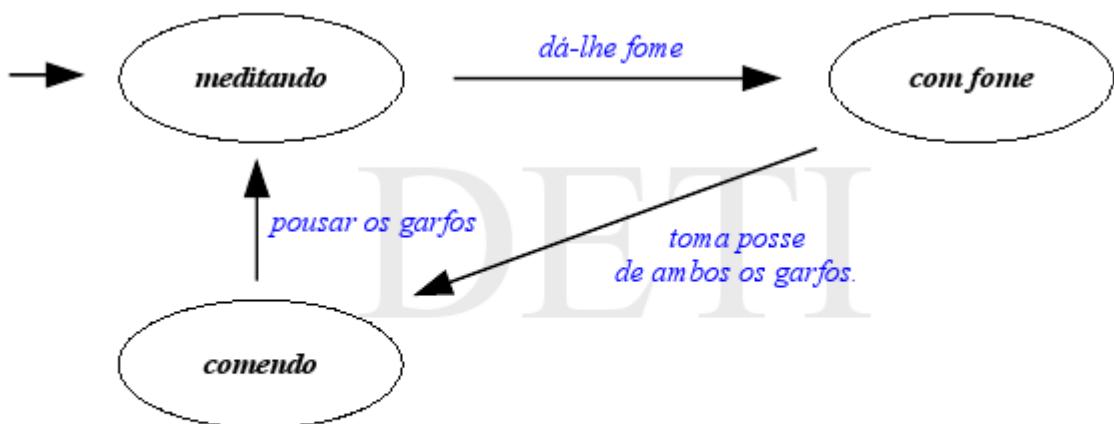
Jantar dos filósofos



- 5 filósofos estão sentados à volta de uma mesa.
- Cada filósofo tem um prato de esparguete à frente e um garfo de cada lado, como é ilustrado na figura.
- Os filósofos alternam períodos em que meditam, com períodos em que comem.
- Para comer, um filósofo tem primeiro que pegar nos garfos colocados à sua esquerda e à sua direita. Um só não é suficiente porque o esparguete é muito escorregadio

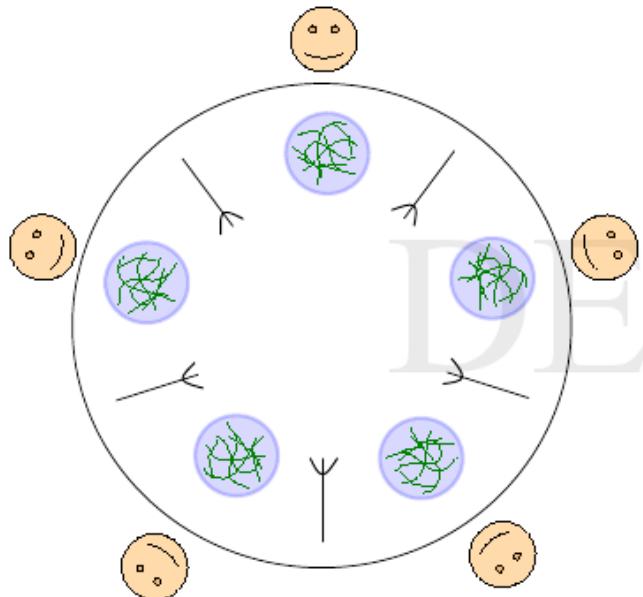


Pode gerar
deadlock



Pode gerar adiamento indefinido

Jantar dos filósofos – semáforos



Estrutura de dados de suporte

- /* estado de cada filósofo */
 shared unsigned int estado[N];
- /* ponto de sincronização de cada filósofo */
 ***/**
shared unsigned int sync[N];
- /* exclusão mútua */
 shared unsigned int acesso;

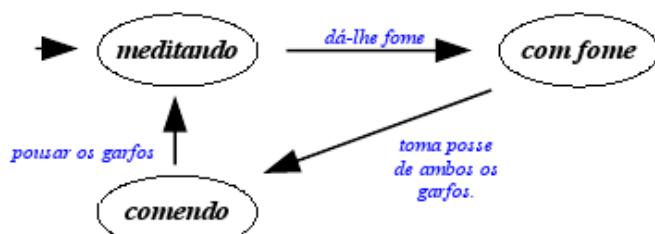
NOTA: é **shared unsigned int vez[N]** e não **sync[N]**

```

/* caracterização do estado interno */
#define N           5           /* n.º de filósofos */
enum { MEDITANDO=0, COM_FOME, COMENDO };
shared unsigned int estado[N] = {MEDITANDO, MEDITANDO, ... , MEDITANDO};

/* caracterização do mecanismo de sincronização */
shared unsigned int acesso; /* id. do semáforo de acesso */
shared unsigned int vez[N]; /* id. dos semáforos de sincronização */

void filosofo(unsigned int f) /*f=0,1,...,N-1*/
{
    forever
    {
        medita();
        da_lhe_fome(f);
        pega_nos_garfos(f);
        come();
        pousa_os_garfos(f);
    }
}
  
```



```

void da_lhe_fome (unsigned int f)
{
    sem_down(acesso)           /* entrada na região crítica */

    estado[f] = COM_FOME;      /* o filósofo assume que tem fome */

    sem_up(acesso);           /* saída da região crítica */
}
  
```

Será necessário entrar em região crítica ?

```
void pousa_os_garfos (unsigned int f)
{
    sem_down (acesso);                                /* entrada na região critica */
    estado[f] = MEDITANDO;                            /* pousa os garfos */

/* põe o filósofo à esquerda a comer se ... */
    if (estado[ESQ(f) == COM_FOME && estado[ESQ(ESQ(f))] != COMENDO)
    {
        estado[ESQ(f)] = COMENDO;
        sem_up(vez[ESQ(f)]);
    }

/* põe o filósofo à direita a comer se ... */
    if (estado[DIR(f) == COM_FOME && estado[DIR(DIR(f))] != COMENDO)
    {
        estado[DIR(f)] = COMENDO;
        sem_up(vez[DIR(f)]);
    }

    sem_up (acesso)                                     /* sai da região critica */
}

#define DIR(f) ((f+1)%N)      /* id do filósofo à direita do filósofo f */

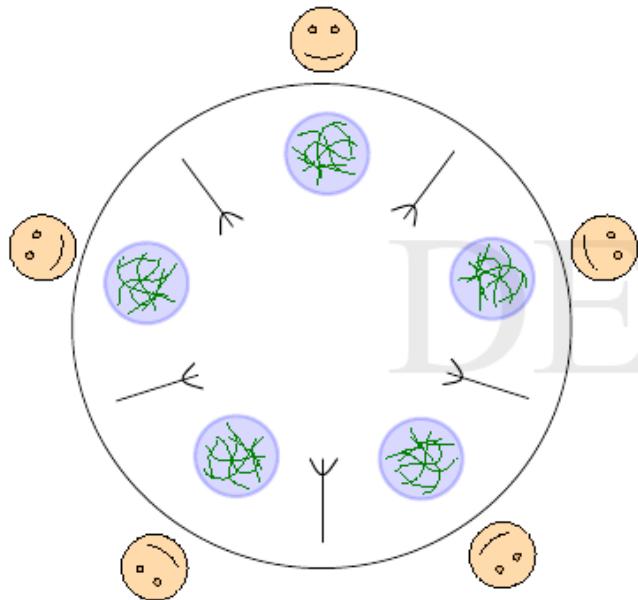
void pega_nos_garfos (unsigned int f)
{
    sem_down(acesso);                                /* entrada na região critica */

    if (estado[ESQ(f)] != COMENDO && estado[DIR(f)] != COMENDO)
    {
        estado[f] = COMENDO;                          /* toma posse dos garfos */
        sem_up(acesso);                            /* saída da região critica */
    }
    else
    {
        sem_up(acesso);                            /* saída da região critica */
        sem_down(vez[f]);                         /* aguarda pela posse dos garfos */
    }
}
```

NOTA: temos um semáforo por filósofo.

NOTA: mexer no vez é sempre dentro do semáforo acesso.

Jantar dos filósofos – monitor



Estrutura de dados de suporte

- /* estado de cada filósofo */
`unsigned int estado[N];`
- /* ponto de sincronização de cada filósofo */
`condition sync[N];`

NOTA: é **condition vez[N]** e não sync[N]

(* threads filósofos - $f = 0, 1, \dots, N-1$ *)

```
procedure filosofo (f: integer);
begin
  forever
  begin
    meditar();
    mesa.da_lhe_fome(f)
    mesa.pega_nos_garfos(f);
    comer();
    mesa.pousar_os_garfos(f)
  end
end; (* filosofo *)
```

NOTA: *procedure filosofo()* igual ao *main()* dos semáforos excepto que acrescenta o “mesa.” .

```

monitor mesa;                                (* monitor de Hoare ou de Lampson / Redell *)

const
  N = 5;
  MEDITANDO = 0;
  COM_FOME = 1;
  COMENDO = 2;

var
  estado: array [0:N-1] of integer;          (* estado dos filósofos *)
  vez: array [0:N-1] of condition;        (* sincronização *)

procedure da_lhe_fome(id: integer);
begin
  estado[id] := COM_FOME;
end

procedure pega_nos_garfos (id: integer);
var
  esq, dir: integer;
begin
  esq := (id-1+N) mod N;
  dir := (id+1) mod N;
  if estado[esq] <> COMENDO and estado[dir] <> COMENDO then
    begin
      estado[id] = COMENDO;
    end
  else
    begin
      wait(vez[id]);
    end
  end;

```



```

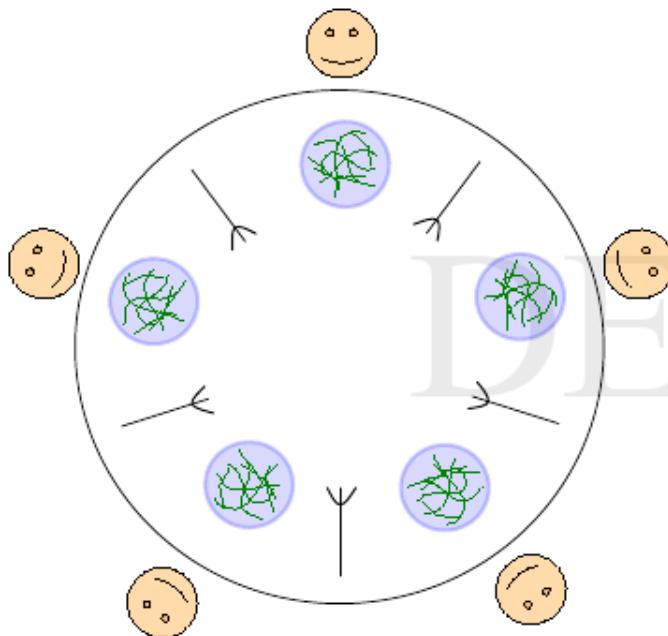
procedure pousa_os_garfos(id: integer);
var
  esq, dir, esq2, dir2: integer;
begin
  esq := (id+N-1) mod N;
  esq2 := (esq+N-1) mod N;
  if estado[esq] = COM_FOME and estado[esq2] <> COMENDO then
  begin
    estado[esq] := COMENDO;
    signal(vez[esq]);
  end

  dir := (id+N-1) mod N;
  dir2 := (dir+N-1) mod N;
  if estado[dir] = COM_FOME and estado[dir2] <> COMENDO then
  begin
    estado[dir] := COMENDO;
    signal(vez[dir]);
  end
end

```

NOTA: algoritmo “igual” ao dos semáforos, tirando que é Pascal.

Jantar dos filósofos – mensagens



Estrutura de dados de suporte

```

●/* estado de cada filósofo */
typedef struct
{
    unsigned int estado[N];
} MESSAGE_EAF;

●/* id da mensagem de acesso, que inclui o
   estado dos filósofos */
unsigned int acesso;

●/* ponto de sincronização de cada filósofo */
unsigned int sync[N];

```

NOTA: é **unsigned int** sinc[N] e não sync[N]

```

/* caracterização do estado interno */

#define N 5
#define MEDITANDO 0
#define COM_FOME 1
#define COMENDO 2

```

/* n.º de filósofos */
 /* o filósofo está a meditar */
 /* o filósofo está com fome */
 /* o filósofo está a comer */

```

/* estrutura de dados das mensagens trocadas */
typedef struct
{
    unsigned int estado[N] /* estado dos filósofos */
} MESSAGE_EAF;

```

/* não tem conteúdo informativo */

```

typedef struct
{
    char dummy; /* não tem conteúdo informativo */
} MESSAGE_SINC;

```

```

/* A inicialização deve colocar na caixa de correio acesso uma mensagem que descreve o estado inicial do
 * problema: todos os filósofos estão a meditar
 */

```

```
static unsigned int acesso; /* id da caixa de correio de exclusão mútua */
```

```

static MESSAGE_EAF meaf =
{
    { MEDITANDO, MEDITANDO, ... , MEDITANDO}
};

```

```

/* ids. das caixas de correio (filas de mensagens) */
static unsigned int acesso; /* exclusão mútua e estado */
static unsigned int sinc[N]; /* sincronização */

/* suporte para as mensagens */
static MESSAGE_EAF meaf;
static MESSAGE_SINC sinc;

/* processos filósofos - f= 0, 1, ..., N-1 */
void main (unsigned int f)
{
    forever
    {
        medita();
        da_lhe_fome(f);
        pega_nos_garfos(f);
        come();
        pousa_os_garfos(f);
    }
}

```

NOTA: *main()* igual ao dos semáforos.

```

void da_lhe_fome(unsigned int f)
{
    msg_receive (acesso, &meaf); /* recebe vector de estado */
    meaf.estado[f] = COM_FOME;
    msg_send_nb(acesso, meaf); /* devolve vector */
}

#define ESQ(f) ((f-1+N)%N) /* id do filósofo à esquerda do filósofo f*/
#define DIR(f) ((f+1)%N) /* id do filósofo à direita do filósofo f*/

void pega_nos_garfos(unsigned int f)
{
    msg_receive(acesso, &meaf); /* recebe vector de estado */
    if (meaf.estado[ESQ(f)] != COMENDO && meaf.estado[DIR(f)] != COMENDO)
    {
        meaf.estado[f] = COMENDO;
        msg_send_nb(acesso, meaf); /* devolve vector */
    }
    else
    {
        msg_send_nb(acesso, meaf); /* devolve vector */
        msg_receive(sinc[f], &sinc); /* o filósofo aguarda */
    }
}

```

NOTA: **unsigned int** sinc[N] desempenha a mesma função do *wait()*.

```

void pousa_os_garfos (unsigned int f)
{
    msg_receive (acesso, &meaf);      /* recebe vector de estado */
    meaf.estado[f] = MEDITANDO;

    /* põe o filósofo à esquerda a comer se ... */
    if (meaf.estado[ESQ(f)] == COM_FOME &&
        meaf.estado[ESQ(ESQ(f))] != COMENDO)
    {
        meaf.estado[ESQ(f)] = COMENDO;
        msg_send_nb (sinc[ESQ(f)], msinc);
    }

    /* põe o filósofo à direita a comer se ... */
    if (meaf.estado[DIR(f)] == COM_FOME &&
        meaf.estado[DIR(DIR(f))] != COMENDO)
    {
        meaf.estado[DIR(f)] = COMENDO;
        msg_send_nb (sinc[DIR(f)], msinc);
    }

    msg_send_nb (acesso, meaf);      /* devolve vector */
}

```

NOTA: as três implementações usam ou emulam a existência de

- array de estado
- *wait()* por filósofo e um processo de outro filósofo de quebrar esse *wait()* (*sem_up()* nos semáforos, *signal()* nos monitores, envio de mensagem de sincronização para a caixa de mensagens específica desse filósofo nas mensagens)

Recursos

Um recurso é algo que um processo precisa para a sua execução. Pode ser:

- **um componente físico do sistema computacional**
 - processador
 - região de
 - memória principal
 - ou de memória de massa,
 - dispositivo concretos de entrada / saída,
 - etc;
- **uma estrutura de dados comuns** definida ao nível do sistema de operação
 - tabela de controlo de processos,
 - canais de comunicação,
 - etcdefinida entre processos de uma mesma aplicação.

Em termos do tipo de apropriação que os processos fazem do recurso, há:

recursos preemptable

Se podem ser retirados aos processos que os detêm sem que daí resulte qualquer consequência irreparável à boa execução dos processos;

São, por exemplo, em ambientes multiprogramados:

- o processador,
- ou as regiões de memória principal onde o espaço de endereçamento de um processo está alojado;

recursos non-preemptable

Se não podem ser retirados aos processos que os detêm;

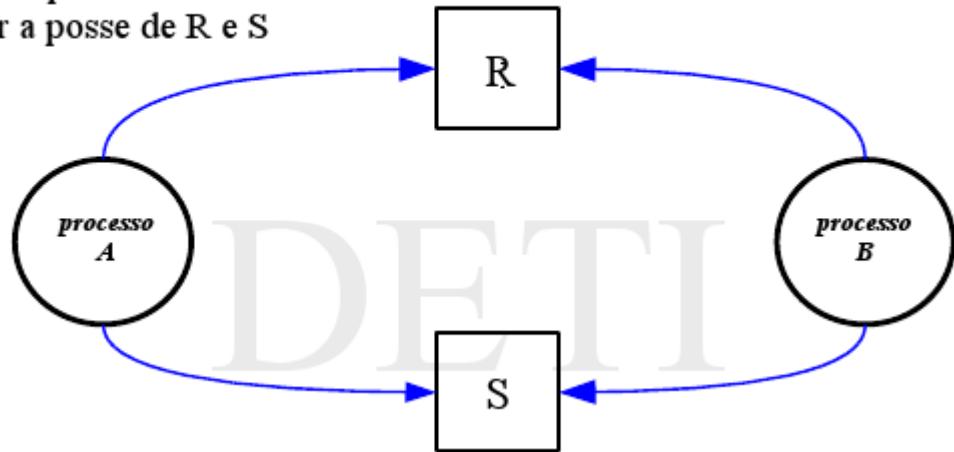
São, por exemplo,

- a impressora,
- ou uma estrutura de dados partilhada que exige exclusão mútua para a sua manipulação.

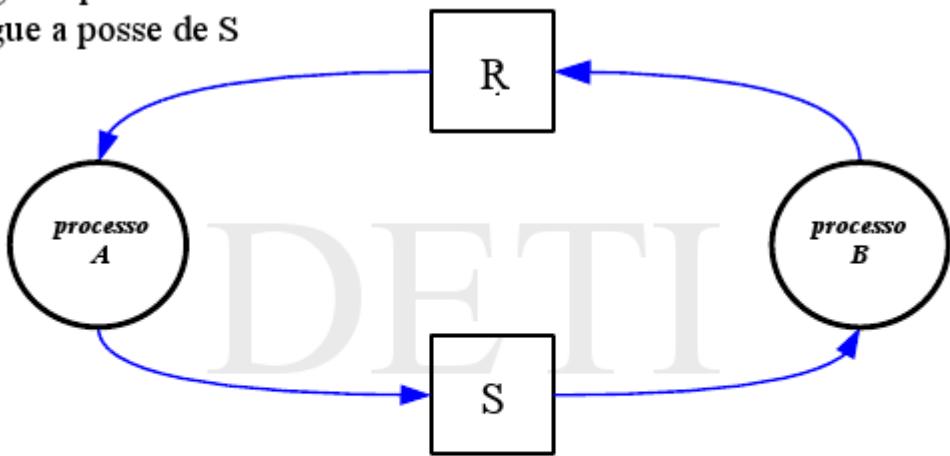
Numa situação de *deadlock*, só os recursos *non-preemptable* são relevantes.

Caracterização esquemática de deadlock

A requer a posse de R e S
B requer a posse de R e S



A consegue a posse de R
B consegue a posse de S



situação típica de deadlock

(a mais simples possível)

Condições necessárias à ocorrência de *deadlock*

Pode demonstrar-se que, sempre que ocorre *deadlock*, há quatro condições que ocorrem necessariamente. São elas

condição de exclusão mútua

cada recurso existente, ou está livre, ou foi atribuído a um e um só processo (a sua posse não pode ser partilhada);

condição de espera com retenção

cada processo, ao requerer um novo recurso, mantém na sua posse todos os recursos

anteriormente solicitados;

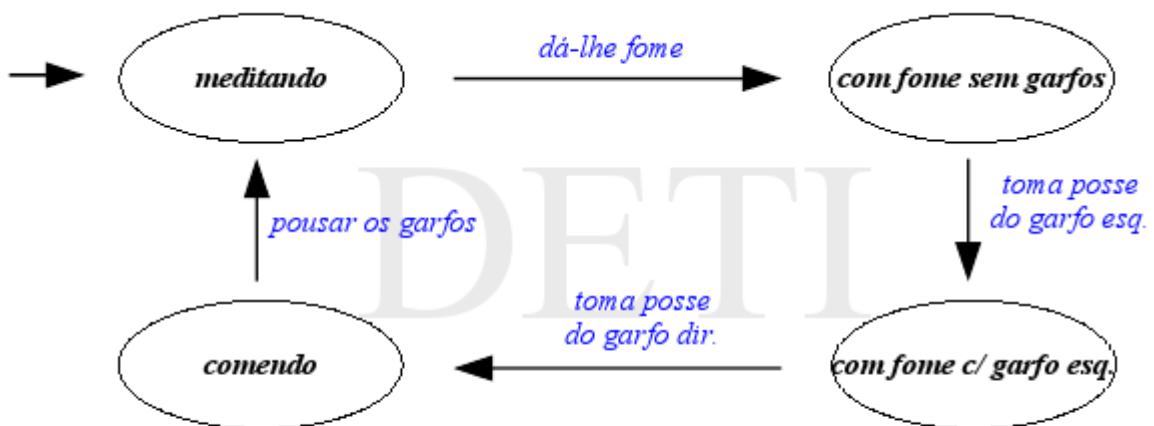
condição de não libertação

ninguém, a não ser o próprio processo, pode decidir da libertação de um recurso que lhe tenha sido previamente atribuído;

condição de espera circular (ou ciclo vicioso)

formou-se uma cadeia circular de processos e recursos, em que cada processo requer um recurso que está na posse do processo seguinte na cadeia.

Exemplo: Jantar dos filósofos



Pode gerar deadlock

Análise Crítica

Se a todos os filósofos lhes dá fome ao mesmo tempo (improvável mas possível) pode acontecer que todos peguem no garfo da esquerda e fiquem bloqueados à espera do da direita.

Está-se, assim, perante uma situação característica de *deadlock* e, como é fácil constatar, todas as condições referidas anteriormente são verificadas

exclusão mútua

cada garfo está na posse de um só filósofo de cada vez;

espera com retenção

cada filósofo, ao tentar pegar no garfo da direita, mantém na sua posse o garfo da esquerda;

não libertação

desde que tome posse de um garfo, cada filósofo conserva-o até ter acabado de comer;
espera circular (ou ciclo vicioso)

formou-se uma cadeia circular de filósofos e

garfos em que cada filósofo conserva o seu garfo da esquerda enquanto espera tomar
posse do da direita.

Prevenção de deadlock no sentido estrito

As condições necessárias à ocorrência de deadlock conduzem à proposição

há deadlock \Rightarrow há exclusão mútua no acesso a um recurso **AND**
 há espera com retenção **AND**
 há não libertação de recursos **AND**
 há espera circular

que é equivalente a

 não há exclusão mútua no acesso a um recurso **OR**
 não há espera com retenção **OR**
 há libertação de recursos **OR**
 não há espera circular \Rightarrow não há *deadlock* .

Assim, desde que uma das condições necessárias à ocorrência de *deadlock* seja negada pelo algoritmo de acesso aos recursos, o *deadlock* torna-se impossível.

Políticas com esta característica designam-se de políticas de prevenção de *deadlock* no sentido estrito (*deadlock prevention*, em inglês).

A primeira delas, exclusão mútua no acesso a um recurso, é bastante restritiva porque só pode ser negada tratando-se de um recurso passível de partilha em simultâneo. Caso contrário, são introduzidas condições de corrida que conduzem, ou podem conduzir, a inconsistência de informação.

O acesso para leitura por parte de múltiplos processos a um dado ficheiro é um exemplo típico da negação desta condição. Note-se que, neste caso, é comum permitir também um acesso para escrita por parte de um processo de cada vez. Quando isto acontece, porém, não se pode impedir completamente a existência de condições de corrida, com a consequente inconsistência de informação. Porquê?

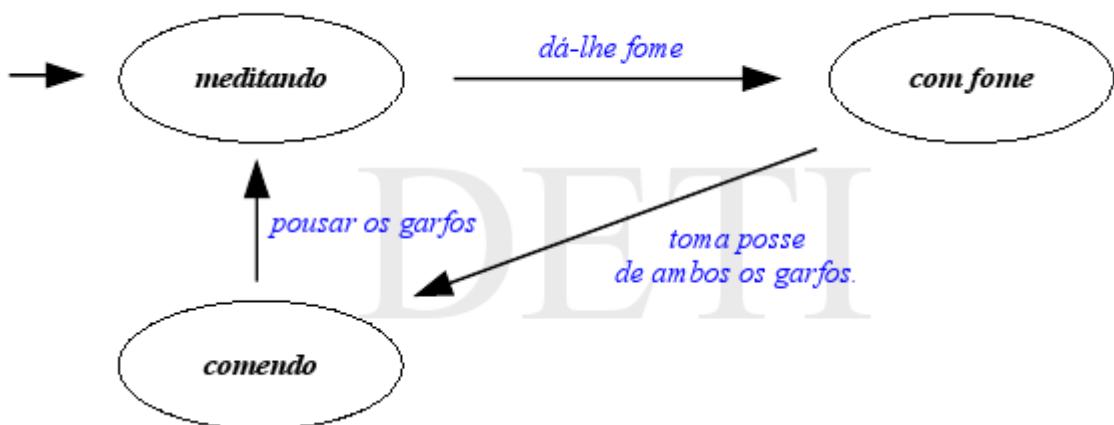
É, por isso, que só as três últimas condições são em geral objecto de negação.

Negando a condição de espera com retenção

Significa que um processo tem que solicitar de um só vez todos os recursos que vai precisar para a sua continuação.

- Se os obtém, o completamento da acção associada está garantido.
- Se não os obtém, tem que aguardar.

Note-se que a ocorrência de adiamento indefinido não está impedida. A solução deve garantir para tal que os recursos necessários serão mais tarde ou mais cedo atribuídos ao processo. A introdução de mecanismos de *aging* é uma solução muito usada nestas situações.

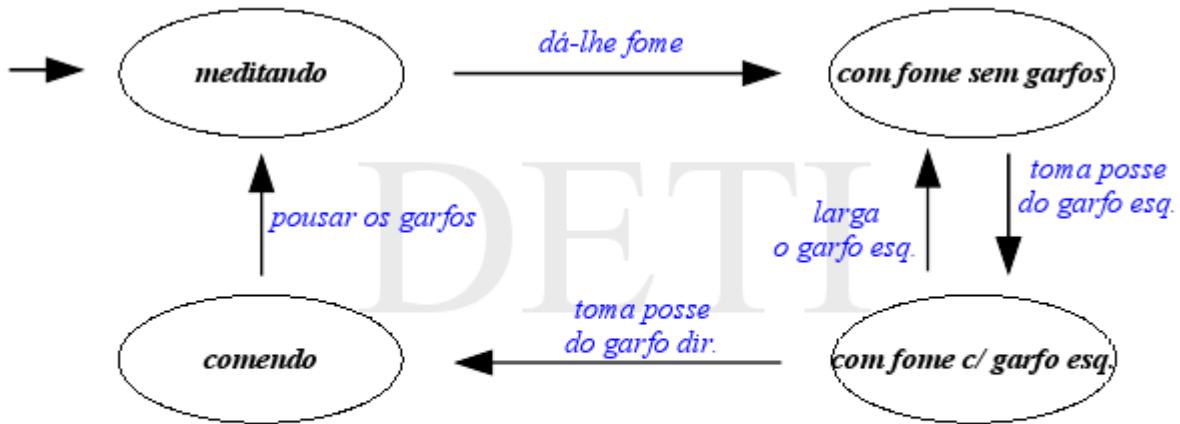


Impondo a condição de libertação de recursos

Significa que um processo quando não obtém os recursos que necessita para a sua continuação, tem que libertar todos os recursos na sua posse, tentando mais tarde a sua obtenção a partir do princípio. Alternativamente, isto significa também impor que um processo só pode deter um recurso de cada vez (o que é completamente inviável em muitas situações).

Um cuidado a ter numa solução deste tipo é que o processo não entre em *busy waiting*. O processo deve bloquear e ser mais tarde acordado quando houver libertação de recursos.

Note-se que a ocorrência de adiamento indefinido não está impedida. A solução deve garantir para tal que os recursos necessários serão mais tarde ou mais cedo atribuídos ao processo. A introdução de mecanismos de *aging* é uma solução muito usada nestas situações.



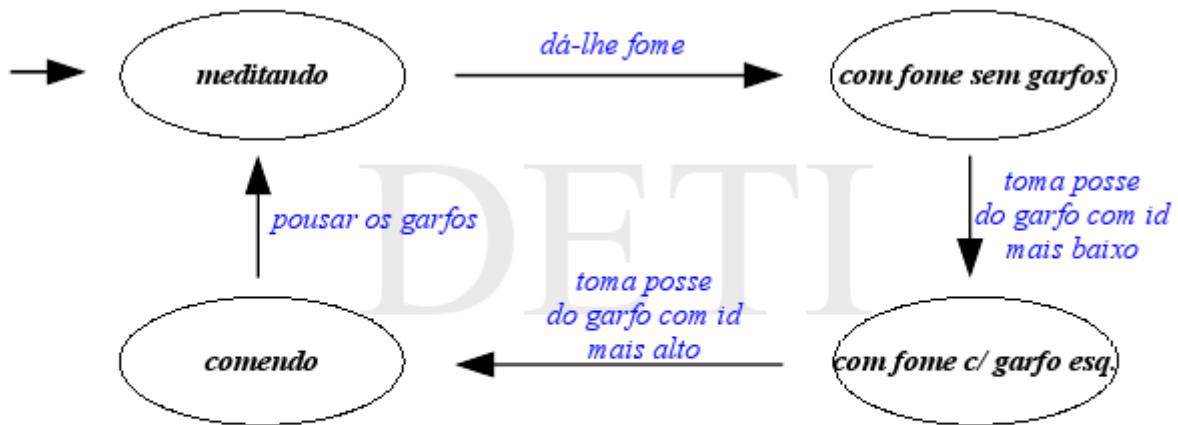
Pode gerar *busy waiting* e *adiamento indefinido*

Negando a condição de espera circular

Significa estabelecer uma ordenação linear dos recursos e impor que um processo quando procura obter os recursos que necessita para a sua continuação, o faça sempre por ordem crescente do número associado a cada um.

Desta maneira, a possibilidade de formação de uma cadeia circular de processos e recursos está posta de parte.

Note-se que a ocorrência de adiamento indefinido não está impedida. A solução deve garantir para tal que os recursos necessários serão mais tarde ou mais cedo atribuídos ao processo. A introdução de mecanismos de *aging* é uma solução muito usada nestas situações.



Pode gerar *adiamento indefinido*

Face ao modo utilizado para identificação dos filósofos e dos garfos, é fácil constatar que

- os filósofos, 0 a N-2, ao pegarem primeiro no garfo da esquerda e, só depois, no garfo da direita, solicitam os garfos (recursos) por ordem crescente do seu número de ordem;
- com o filósofo N-1 passa-se exactamente o contrário, o garfo da esquerda tem o número de ordem N-1 e o da direita o número de ordem 0.

Assim, a imposição de uma ordenação crescente na atribuição de recursos exige apenas que o filósofo N-1 pegue nos garfos pela ordem inversa à dos outros. Primeiro o da direita e, só depois, o da esquerda!

Note-se que a solução não é geral (o **adiamento indefinido não está completamente resolvido**).

Prevenção de deadlock no sentido estrito

Análise Crítica

As políticas de prevenção de deadlock no sentido estrito, embora seguras, são muito restritivas, pouco eficientes e difíceis de aplicar em situações muito gerais.
Resumindo, tem-se que

negação da condição de exclusão mútua

só pode ser aplicada a recursos passíveis de partilha em simultâneo;

negação da condição de espera com retenção

exige o conhecimento prévio de todos os recursos que vão ser necessários, considera

sempre o pior caso possível (uso de todos os recursos em simultâneo);

imposição da condição de não libertação

ao supor a libertação de todos os recursos anteriores quando o próximo não puder ser atribuído, atrasa a execução do processo de modo muitas vezes substancial;

negação da condição de espera circular

desaproveita recursos eventualmente disponíveis que poderiam ser usados na continuação do processo.

Prevenção de *deadlock* no sentido lato

Uma alternativa menos restritiva do que a prevenção de *deadlock* no sentido estrito é

- não negar *a priori* qualquer das condições necessárias à ocorrência de deadlock,
- mas monitorar continuamente o estado interno do sistema de modo a garantir que a sua evolução se faz apenas entre estados ditos seguros.

Define-se-se neste contexto **estado seguro** como uma qualquer distribuição dos recursos do sistema, livres ou atribuídos aos processos que coexistem, que possibilita a terminação de todos eles.

Por oposição, um **estado é inseguro** se não for possível fazer-se uma tal afirmação sobre ele.

Políticas com esta característica designam-se de políticas de prevenção de deadlock no sentido lato (*deadlock avoidance*, em inglês).

Convém notar o seguinte

- é necessário
 - o conhecimento completo de todos os recursos do sistema e
 - cada processo tem que indicar à cabeça a lista de todos os recursos que vai precisarsó assim se pode caracterizar um estado seguro;
- **um estado inseguro não é sinónimo de *deadlock*** – vai, contudo, considerar-se sempre o pior caso possível para garantir a sua não ocorrência.

Definindo

NTR_i – n.º total de recursos do tipo i (com $i = 0, 1, \dots, N-1$)

$R_{i,j}$ – n.º de recursos do tipo i usados pelo processo j (com $i = 0, 1, \dots, N-1$, $j = 0, 1, \dots, M-1$)

$A_{i,j}$ – n.º de recursos do tipo i já atribuídos ao processo j (com $i = 0, 1, \dots, N-1$, $j = 0, 1, \dots, M-1$) .

O problema pode ser abordado de duas maneiras diferentes

- impedimento de lançamento de um novo processo – quando não podem ser garantidas as condições da sua terminação;

– o processo P_M só é lançado se e só se

$$NTR_i \geq R_{i,M} + \sum_{j=0}^{M-1} R_{i,j} \quad (\text{com } i = 0, 1, \dots, N-1) .$$

- impedimento de atribuição de um novo recurso a um dado processo – quando daí resultar uma situação insegura

Algoritmo dos banqueiros de Dijkstra

- um novo recurso de tipo i só é atribuído ao processo P_m se e só se for possível ordenar os processos numa sucessão $j' = f(i,j)$ em que se tem sempre que

$$R_{i,j'} - A_{i,j'} < NTR_i - \sum_{k=j'}^{M-1} A_{i,k} \quad (\text{com } i = 0, 1, \dots, N-1, j' = 0, 1, \dots, M-1) ;$$

ou seja, quando for possível encontrar pelo menos uma sucessão de atribuição de recursos que conduza à terminação de todos os processos que coexistem.

Exemplo: Jantar dos filósofos



- cada filósofo, quando pretende os garfos, continua a pegar primeiro no garfo da

esquerda e, só depois, no da direita, mas agora mesmo que o garfo da esquerda esteja sobre a mesa, o filósofo nem sempre pode pegar nele;

- só o poderá fazer se pelo menos um dos outros filósofos estiver a meditar; caso contrário, a atribuição origina uma situação insegura; porquê?
- não se trata, contudo, de uma solução geral (**o adiamento indefinido não está completamente resolvido**).

Algoritmo dos banqueiros de Dijkstra

```
/* caracterização do estado interno */
#define N          5           /* n.º de filósofos */
#define MEDITANDO  0           /* o filósofo está a meditar */
#define GARFO_ESQ  1           /* o filósofo aguarda a posse do garfo da esquerda */
#define GARFO_DIR  2           /* o filósofo aguarda a posse do garfo da direita */
#define COMENDO    3           /* o filósofo está a comer */

shared unsigned int estado[N] = {MEDITANDO, MEDITANDO, ..., MEDITANDO};

/* caracterização do mecanismo de sincronização */
#define ESQ(f)    ((f-1)%N)   /* ident. do filósofo à esquerda do filósofo f */
#define DIR(f)    ((f+1)%N)   /* ident. do filósofo à direita do filósofo f */
#define GESQ(f)    f           /* ident. do garfo da esquerda do filósofo f */
#define GDIR(f)   ((f+1)%N)   /* ident. do garfo da direita do filósofo f */
shared unsigned int acesso;
shared unsigned int vez[N]; /* id. dos semáforos de sincronização pela posse de garfo - campo val = 0 */

/* processos filósofos - f = 0, 1, ..., N-1 */
void main (unsigned int f)
{
    forever
    {
        meditar ();
        pegar_nos_garfos_esquerda_direita (f);
        comer ();
        pousar_os_garfos_esquerda_direita (f);
    }
}
```

```

/*procedimento de pegar nos garfos um de cada vez primeiro o da esquerda, depois o da direita */
void pegar_nos_garfos_esquerda_direita (unsigned int f)
{
    sem_down (acesso);                                /* entrada na região crítica */
    estado[f] = GARFO_ESQ;                          /* o filósofo assumiu que quer o garfo da esquerda */
    avaliar_situacao (f, GESQ(f));                  /* tenta garantir a sua posse */
    sem_up (acesso);                                /* saída da região crítica */
    sem_down (vez[GESQ(f)]);                         /* aguarda, se não consegue obter o garfo da esquerda */
    sem_down (acesso);                                /* entrada na região crítica */
    estado[f] = GARFO_DIR;                          /* o filósofo assumiu que quer o garfo da direita */
    avaliar_situacao (f, GDIR(f));                  /* tenta garantir a sua posse */
    sem_up (acesso);                                /* saída da região crítica */
    sem_down (vez[GDIR(f)]);                         /* aguarda, se não consegue obter o garfo da direita */
}

/*procedimento de pousar nos garfos um de cada vez primeiro o da esquerda, depois o da direita */
void pousar_os_garfos_esquerda_direita (unsigned int f)
{
    sem_down (acesso);                                /* entrada na região crítica */
    estado[f] = MEDITANDO;                           /* o filósofo prepara-se para meditar */
    avaliar_situacao (ESQ(f), GESQ(f));             /* avalia o caso do filósofo da esquerda que pode querer o garfo */
    avaliar_situacao (DIR(f), GDIR(f));             /* avalia o caso do filósofo da direita que pode querer o garfo */
    sem_up (acesso);                                /* saída da região crítica */
}

/*procedimento de avaliar a situação */
void avaliar_situacao (unsigned int f, unsigned int g)
{
    if ((g == GESQ(f)) && (estado[f] == GARFO_ESQ))
    {
        BOOLEAN avanco = FALSE;
        unsigned int i;
        for (i = 0; i < N; i++)                      /* verificar se há algum filósofo a meditar */
            avanco = avanco || (estado[i] == MEDITANDO);
        if (avanco && (estado[ESQ(f)] != COMENDO))
        {
            estado[f] = GARFO_DIR;                  /* o garfo está efectivamente na sua posse */
            sem_up (vez[g]);                      /* o processo de continuação é garantido */
        }
    }
    else if ((g == GDIR(f)) && (estado[f] == GARFO_DIR) && (estado[DIR(f)] != COMENDO))
    {
        estado[f] = COMENDO;                      /* o garfo está efectivamente na sua posse */
        sem_up (vez[g]);                        /* o processo de continuação é garantido */
    }
}

```

Análise Crítica

As políticas de prevenção de *deadlock* no sentido lato, embora igualmente seguras, não são menos restritivas e ineficientes do que as políticas de prevenção de *deadlock* no sentido estrito. São, porém, mais versáteis e, por isso, mais fáceis de aplicar em situações gerais.

Resumindo, tem-se que

- **impedimento de lançamento de um novo processo** – a avaliação de estado seguro é feita *a priori*, o que tem como consequência uma política extremamente restritiva que leva a condição de pior caso possível ao limite;
- **impedimento de atribuição de um novo recurso a um dado processo** – a avaliação de estado seguro é adiada para o momento de atribuição de um novo recurso, o que possibilita uma gestão mais eficiente dos recursos disponíveis e, portanto, um aumento de *throughput* do sistema.

Detecção e recuperação de deadlock

Uma última alternativa é pura e simplesmente atribuir os recursos sempre que eles estão disponíveis. A possibilidade de *deadlock* está então sempre presente e terá que ser tida em conta.

Existem duas estratégias que podem ser seguidas

estratégia da avestruz

a mais comum, ignorar pura e simplesmente o problema e, quando o *deadlock* acontecer, matar os processos envolvidos ou, no caso limite, reiniciar o sistema;

detecção de deadlock

monitorar de tempos a tempos o estado dos diferentes processos, procurando determinar se existem cadeias circulares de processos e recursos (a teoria de grafos é comumente utilizada no desenvolvimento de algoritmos deste tipo).

Após a detecção de deadlock, a cadeia circular de processos e recursos pode ser quebrada usando diversos métodos

libertação forçada de um recurso

- em algumas situações, um recurso pode ser retirado a um processo, que é entretanto suspenso, permitindo o prosseguimento da execução dos restantes;
- mais tarde, o recurso volta a ser atribuído ao processo e este é recomeçado;
- trata-se do método mais eficaz,
- mas exige que o estado do recurso possa ser salvaguardado;

rollback

- estabelecer um funcionamento do sistema que impõe o armazenamento periódico do estado dos diferentes processos:
 - deste modo, quando ocorre deadlock, um recurso é libertado e
 - o processo que o detinha vê a sua execução recuada até a um ponto anterior à atribuição desse recurso;

morte de processos

- é o método mais radical
- e mais fácil de implementar.

Sinais em Unix

Um sinal constitui uma interrupção produzida no contexto de um processo onde lhe é comunicada a ocorrência de um acontecimento especial.

- Pode ser despoletado pelo *kernel*, em resultado de situações de erro ao nível *hardware* ou de condições específicas ao nível *software*,
- pelo próprio processo,
- por outro processo,
- pelo utilizador através do dispositivo standard de entrada.

Quando recebe um sinal um processo pode:

ignorá-lo

não fazer nada face à sua ocorrência;

bloqueá-lo

impedir que interrompa o processo durante intervalos de processamento bem definidos;

executar uma acção associada

- pode ser a acção estabelecida por defeito quando o processo é criado (conduz habitualmente à sua terminação ou suspensão de execução),
- ou uma acção específica que é introduzida (registada) pelo próprio processo em *runtime*.

Existem em Unix um conjunto variado de chamadas ao sistema que possibilitam o processamento de sinais. Destacam-se entre elas:

kill

um sinal (especificado em argumento) é enviado a um processo ou grupo de processos;

raise

um sinal (especificado em argumento) é enviado ao próprio processo;

sigaction

registo de um função para serviço a um sinal específico;

sigprocmask

manipulação da máscara que inibe ou activa o processamento de sinais específicos;

sigpending

determinação dos sinais pendentes;

sigsuspend

suspensão da execução do processo até à chegada de um sinal.

gcc -D__USE_POSIX ...

Tabela dos sinais mais comuns (standard Posix.1)

<i>Sinal</i>	<i>Valor</i>	<i>Acção</i>	<i>Causa</i>
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort (3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm (2)
SIGTERM	15	Term	Termination signal
SIGUSR1	10	Term	User-defined signal 1
SIGUSR2	12	Term	User-defined signal 2
SIGCHLD	17	Ign	Child stopped or terminated
SIGCONT	18		Continue if stopped
SIGSTOP	19	Stop	Stop process
SIGTSTP	20	Stop	Stop typed at tty
SIGTTIN	21	Stop	tty input for background process
SIGTTOU	22	Stop	tty output for background process

NOTA – The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.

Exemplo de processamento de um sinal

```
/* Variáveis localmente globais */
static unsigned int a = 0;                      /* variável contadora */
static BOOLEAN alteracao = FALSE;                /* alteração da variável contadora */

static void control_C (int);
                                         /* protótipo da rotina de atendimento */

void main (int argc, char *argv[])
{
    struct sigaction act, oact;                  /* acção a introduzir e acção existente */
    BOOLEAN fim = FALSE;                         /* sinalização de fim de processamento */

    /* registo da nova função de serviço do sinal */
    act.sa_handler = control_C;                  /* indicação da função de serviço */
    sigemptyset (&act.sa_mask);                 /* nenhum outro sinal será bloqueado */
    act.sa_flags = 0;                            /* comportamento por defeito */

    if (sigaction (SIGINT, &act, &oact) != 0)    /* registo da rotina */
    {
        perror ("erro no registo da nova função");
        exit (EXIT_FAILURE);
    }

    /* processamento */
    printf ("Prima sucessivamente a tecla ctrl-c\n");
    do
    { switch (a)
        { case 0: printf ("aguardando!\n"); break;
          case 1: printf ("primeira vez!\n"); break;
          case 2: printf ("segunda vez!\n"); break;
          case 3: printf ("terceira vez!\n"); fim = TRUE;
        }
        if ((a == 1) || (a == 2)) alteracao = FALSE;
        while (!alteracao);
    } while (!fim);
    printf ("acabou!\n");

    exit (EXIT_SUCCESS);
}

static void control_C (int signum)
{
    a += 1;
    alteracao = TRUE;
}
```

Semáforos em Unix

A implementação de semáforos em Unix ultrapassa em muitos aspectos a definição original de Dijkstra. As principais diferenças são

tratamento em bloco de grupos de semáforos

é possível criar, destruir e manipular atomicamente mais do que um semáforo de cada vez;

operações básicas diferentes

as chamadas ao sistema são *semget*, *semop* e *semctl*; além da sincronização convencional, em que os processos bloqueiam se o campo val for zero, é também possível bloquear processos se o campo val não for zero; além disso, as operações de *down* e *up* podem ser também realizadas com decrementos e incrementos do campo val diferentes de um.

```
gcc -D_SVID_SOURCE ...
```

Memória partilhada em Unix

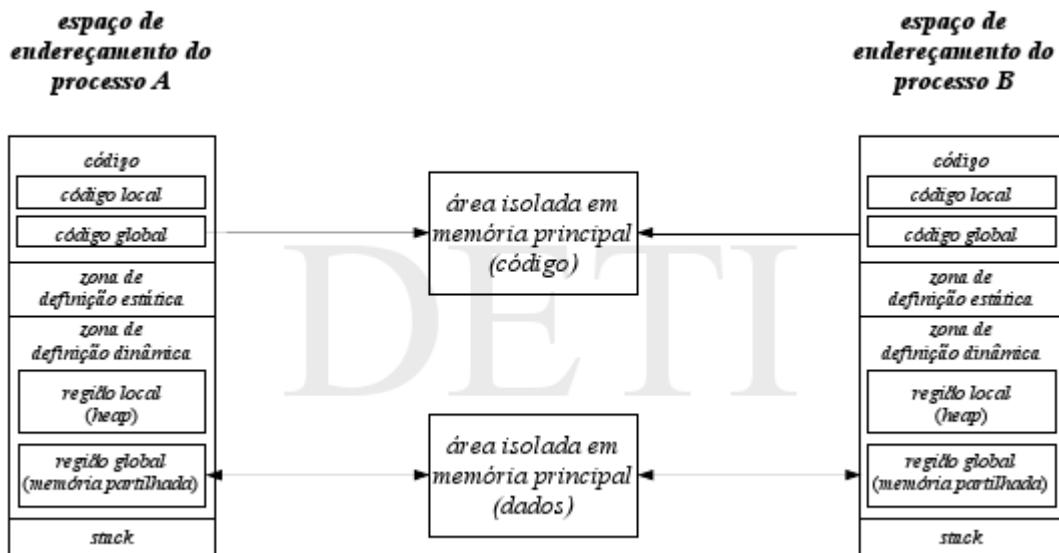
Uma região de memória partilhada é estabelecida no espaço de endereçamento do processo apenas em *runtime*. Está localizada na

zona de definição dinâmica

quando representa partilha de dados, o acesso à informação aí armazenada é feito tipicamente pela desreferenciação de um ponteiro para uma estrutura de dados cujos campos constituem os diferentes componentes de interesse:

zona de código

quando representa partilha de bibliotecas de sistema, o acesso à informação aí armazenada é feito tipicamente pela desreferenciação de um ponteiro para uma estrutura de dados cujos campos constituem ponteiros para funções específicas.



O estabelecimento de uma região de memória partilhada no espaço de endereçamento de um processo exige as operações seguintes

criação ou ligação a uma região previamente criada

é aí definido o tipo de acesso *read/write* ou *readonly*;

anexação da região ao espaço de endereçamento do processo

obtém-se aqui um ponteiro para a sua localização;

desanexação da região do espaço de endereçamento do processo

operação complementar da anterior;

destruição de uma região existente

operação complementar da criação

Estas operações são construídas a partir das chamadas ao sistema *shmget*, *shmat*, *shmdt* e i.

gcc -D_SVID_SOURCE ...

Threads e monitores em Unix

O standard POSIX, IEEE 1003.1c, define um interface de programação de aplicações (API) para criação e sincronização de *threads*. Os sistemas de operação da família Unix fornecem habitualmente uma biblioteca que o implementa: *pthread*. Permite a construção de monitores em Linguagem C.

Como o C não é uma linguagem concorrente, o conceito de monitor não é suportado directamente pela linguagem, mas pode ser implementado a partir de *mutexes* e variáveis de condição fornecidos pela biblioteca.

São monitores de tipo *Lamson / Redell*.

```
gcc -lpthread ...
```

Entre as funções fornecidas, destacam-se

pthread_create

- lançamento de um novo thread (correspondente ao fork no caso de processos)

pthread_exit

- terminação de um *thread* (correspondente ao *exit*)

pthread_join

- espera pela terminação de *thread* (correspondente ao *waitpid*)

pthread_self

- identificação de um *thread* (correspondente ao *getpid*)

pthread_once, pthread_mutex_*, pthread_cond_*

- necessárias para a construção de monitores;

Mensagens em Unix

A implementação de mensagens em Unix é bastante restritiva. O modelo assume a existência de filas de mensagens onde são armazenadas mensagens de diferentes tipos, descritos por números inteiros positivos.

O formato de cada mensagem é

```
struct msgbuf
{ unsigned long mtype; /* tipo da mensagem, > 0 */
  DATA info;           /* conteúdo informativo */
}
```

Um aspecto a ter em conta no estabelecimento dos argumentos das primitivas de envio e recepção de mensagens é que se passa um ponteiro para uma variável de tipo *struct msgbuf*, que representa a mensagem, mas se indica como seu comprimento apenas o comprimento do conteúdo informativo. Isto significa que é possível enviar e receber mensagens de comprimento nulo.

- O envio de mensagens é, em princípio, **não bloqueante**. A primitiva só bloqueia se não houver espaço suficiente na fila para armazenamento da mensagem.
- A recepção pode ser **blockante** ou **não blockante** e permite alternativamente recolher a primeira mensagem de um dado tipo, de qualquer tipo, de um tipo diferente do tipo especificado ou de um tipo menor ou igual ao tipo especificado, existente na fila.
- Estas operações são construídas a partir das chamadas ao sistema *msgget*, *msgsnd*, *msgrcv* e *msgctl*.

```
gcc -D_SVID_SOURCE ...
```

Pipes em Unix

Um *pipe* é um canal de comunicação elementar que é estabelecido entre dois ou mais processos. O mecanismo de *piping* constitui a forma tradicional de comunicação entre processos em Unix, tendo sido inclusivamente introduzido por ele.

Apresenta, contudo, algumas limitações

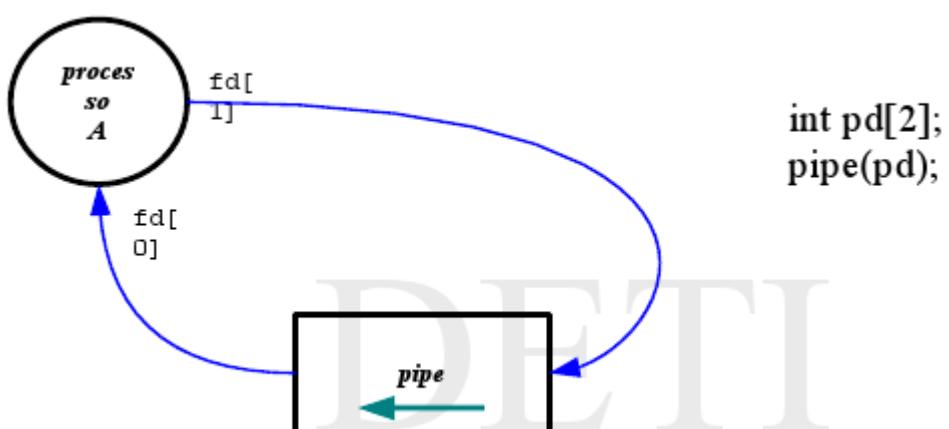
o canal de comunicação é unidireccional

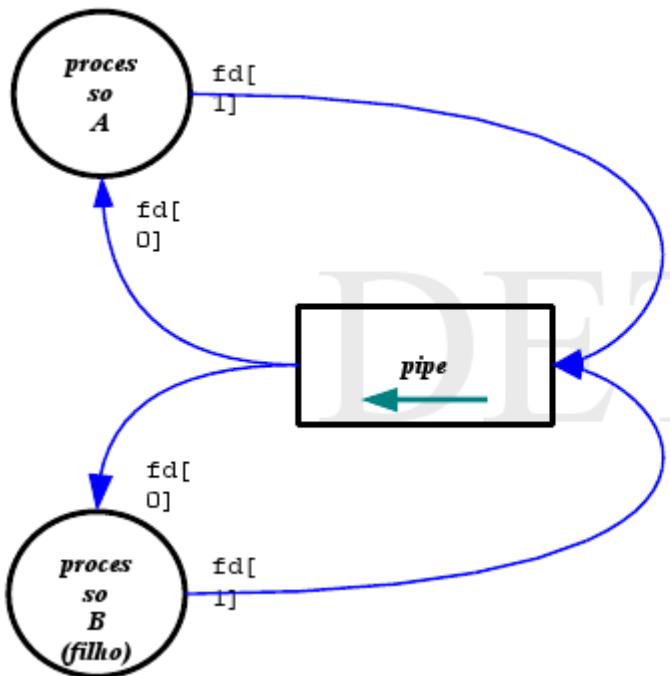
só permite comunicação num sentido;

só pode ser usado entre processos que estão relacionados entre si

um *pipe* é criado por um processo, que a seguir se duplica uma ou mais vezes e a comunicação é então estabelecida entre o pai e o filho, ou entre dois irmãos.

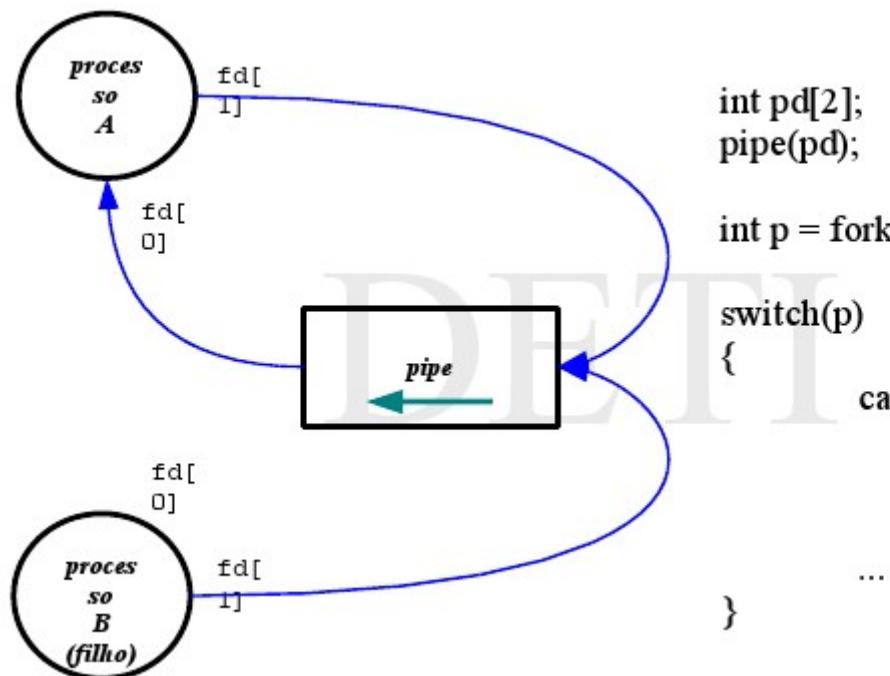
A sua principal utilização é na composição de comandos complexos a partir de uma organização em cadeia de comandos mais simples em que o standard *output* de um dado comando é redireccionado para a entrada de um *pipe* e o standard *input* do comando seguinte é redireccionado para a saída do mesmo *pipe*.

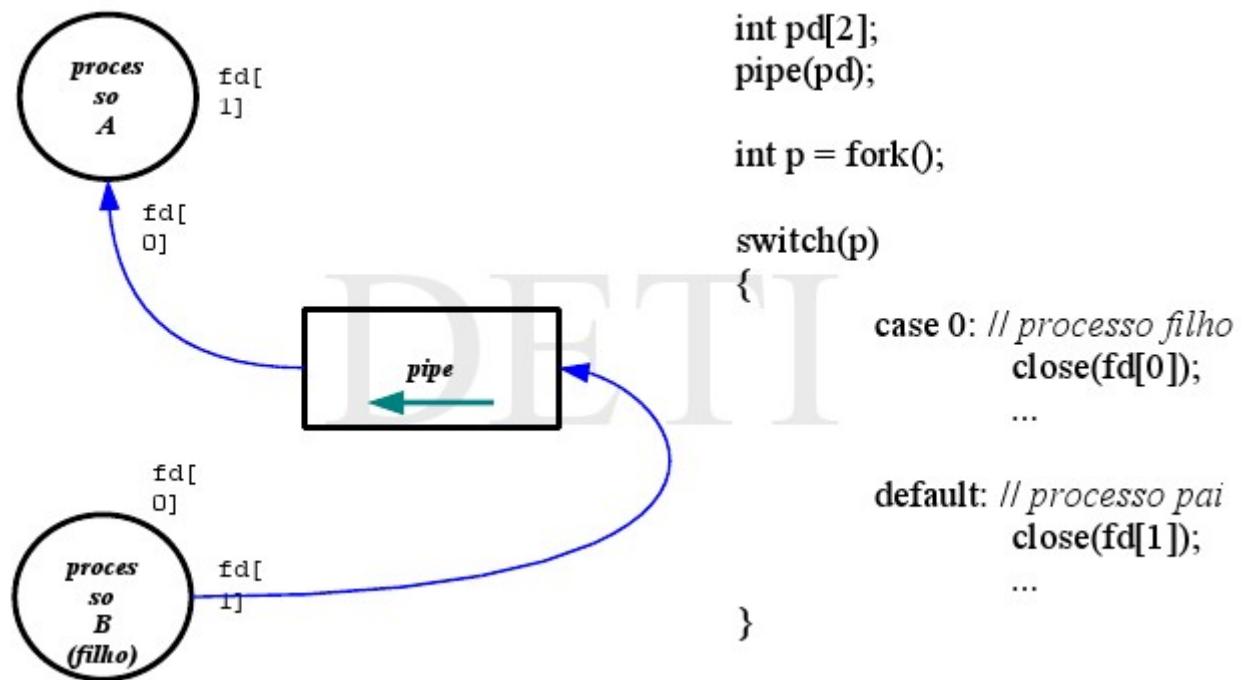




```
int pd[2];
pipe(pd);

int p = fork();
```





Exemplo de comunicação pai – filho

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

void main (void)
{
    pid_t pid;                                /* identificador do processo */
    int status;                               /* status de execução do processo filho */
    int fd[2];                                /* descritores de ficheiro para o pipe */
                                                /* linha de texto */
    char line[14];                            /* variável de controlo de leitura / escrita */

    /* criação do pipe */
    if (pipe (fd) != 0)
        { perror ("erro na criação do pipe");
          exit (EXIT_FAILURE);
        }

    /* duplicação do processo */
    if ((pid = fork ()) < 0)
        { perror ("erro na duplicação do processo");
          exit (EXIT_FAILURE);
        }

```

```

/* a comunicação é estabelecida do pai para o filho */
if (pid != 0)
    /* processo pai - envia uma mensagem ao processo filho */

    { if (close (fd[0]) != 0)
        { perror ("pai - erro no fecho do canal de entrada");
            exit (EXIT_FAILURE);
        }
        if ((n = write (fd[1], "hello world!\n", 13)) != 13)
        { if (n != -1)
            fprintf (stderr, "pai - foram escritos %d bytes\n", n);
            else perror ("pai - erro de escrita no canal de saída");
            exit (EXIT_FAILURE);
        }
        if (close (fd[1]) != 0)
        { perror ("pai - erro no fecho do canal de saída");
            exit (EXIT_FAILURE);
        }
        if (wait (&status) != pid)
        { perror ("pai - erro na espera pelo processo filho");
            exit (EXIT_FAILURE);
        }
        if (WIFEXITED (status))
            printf ("pai - o processo que lancei terminou\nno seu status foi %d\n",
                   WEXITSTATUS (status));
    }

else /* processo filho - recebe a mensagem do processo pai e
       escreve-a no ecrã do monitor vídeo */
    { if (close (fd[1]) != 0)
        { perror ("filho - erro no fecho do canal de saída");
            exit (EXIT_FAILURE);
        }
        if ((n = read (fd[0], line, 13)) != 13)
        { if (n != -1)
            fprintf (stderr, "filho - foram escritos %d bytes\n", n);
            else perror ("filho - erro de leitura no canal de
                        entrada");
            exit (EXIT_FAILURE);
        }
        printf ("filho - a mensagem do meu pai foi: %s", line);
        if (close (fd[0]) != 0)
        { perror ("filho - erro no fecho do canal de entrada");
            exit (EXIT_FAILURE);
        }
    }
exit (EXIT_SUCCESS);
}

```

Gestão de memória

Introdução

O espaço de endereçamento de um processo tem que estar, pelo menos parcialmente, residente em memória principal para que ele possa ser executado;

Num sistema multiprogramado a memória principal é partilhada entre o núcleo (*kernel*) do sistema de operação e os vários processos de utilizador.

A memória principal tem vindo a crescer, mas

- continua a ser finita,
- “os programas tendem a expandir-se preenchendo toda a memória disponível” (Lei de *Parkinson*)

Logo:

- ou se limita o número de processos que podem coexistir
- ou se arranja uma forma de a expandir – área de swapping.

A memória principal tem vindo a ficar mais rápida, mas o acesso continua a ser

- lento do ponto de vista do processador

Logo:

- deve-se arranjar uma forma de acelerar o acesso – memória cache.

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 processadores actuais,a memória cache está localizada

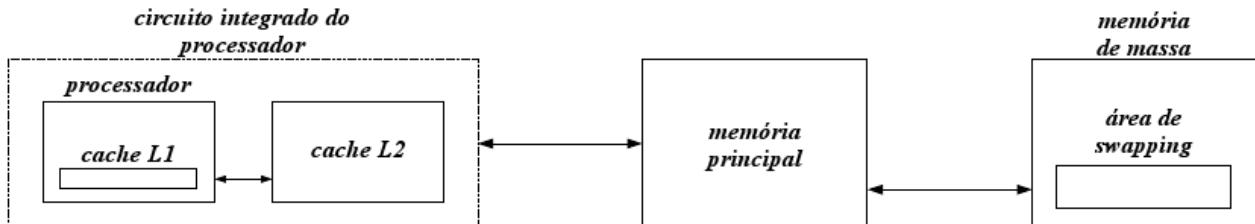
- no próprio circuito integrado do processador (nível 1) e
- num circuito integrado autónomo colado no mesmo substrato (nível 2),
e o controlo da transferência de dados de e para a memória principal é feito de um modo quase completamente transparente ao programador de sistemas;

a memória de massa, 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 factor limitativo ao número de processos que podem correntemente existir.

assim, a memória de um sistema computacional está organizada tipicamente em níveis diferentes, formando uma hierarquia

- **memória cache** - pequena (centenas de KB), 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 de massa** - grande (dezenas, centenas ou milhares de GB), lenta, não volátil e barata;



este tipo de organização baseia-se no pressuposto de 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;

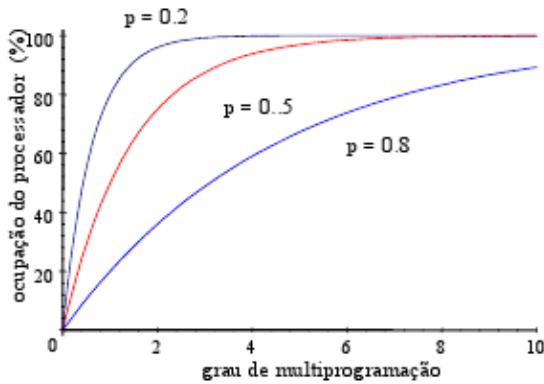
a justificação deste pressuposto é o **princípio da localidade de referência**; trata-se de uma constatação heurística sobre o comportamento de um programa em execução que estabelece que as referências à memória durante a execução de um programa tendem a concentrar-se em fracções bem definidas do seu espaço de endereçamento durante intervalos mais ou menos longos.

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

$$\text{fracção de ocupação do processador} = 1 - p^n \quad (\text{modelo simplificado})$$

p - fracção 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

$$p = 0,8$$

D T

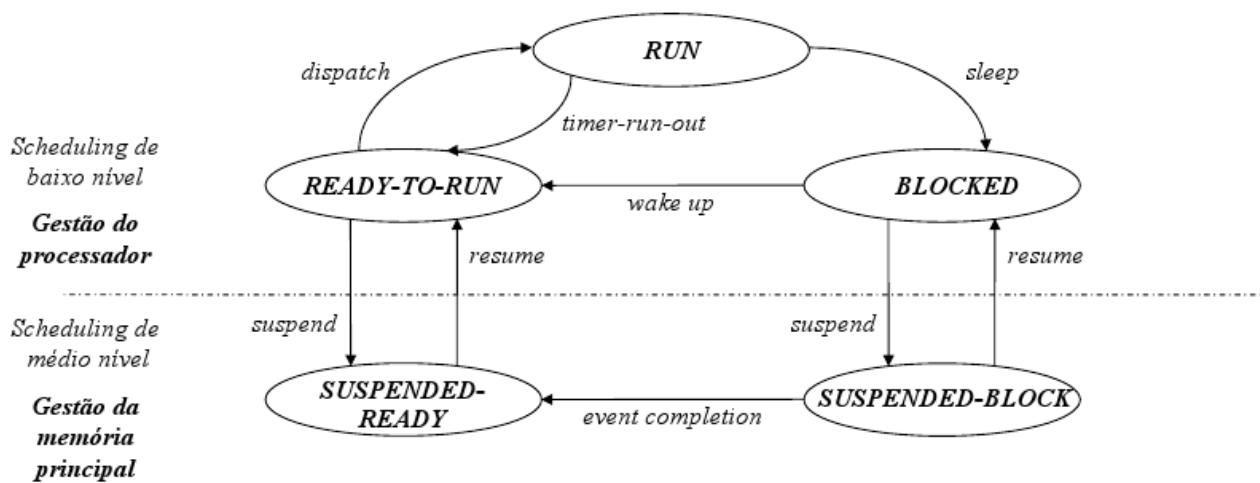
Papel da gestão de memória em multiprogramação

O papel da gestão de memória num ambiente de multiprogramação centra-se, pois, no controlo da transferência de dados entre a memória principal e a memória de massa, visando

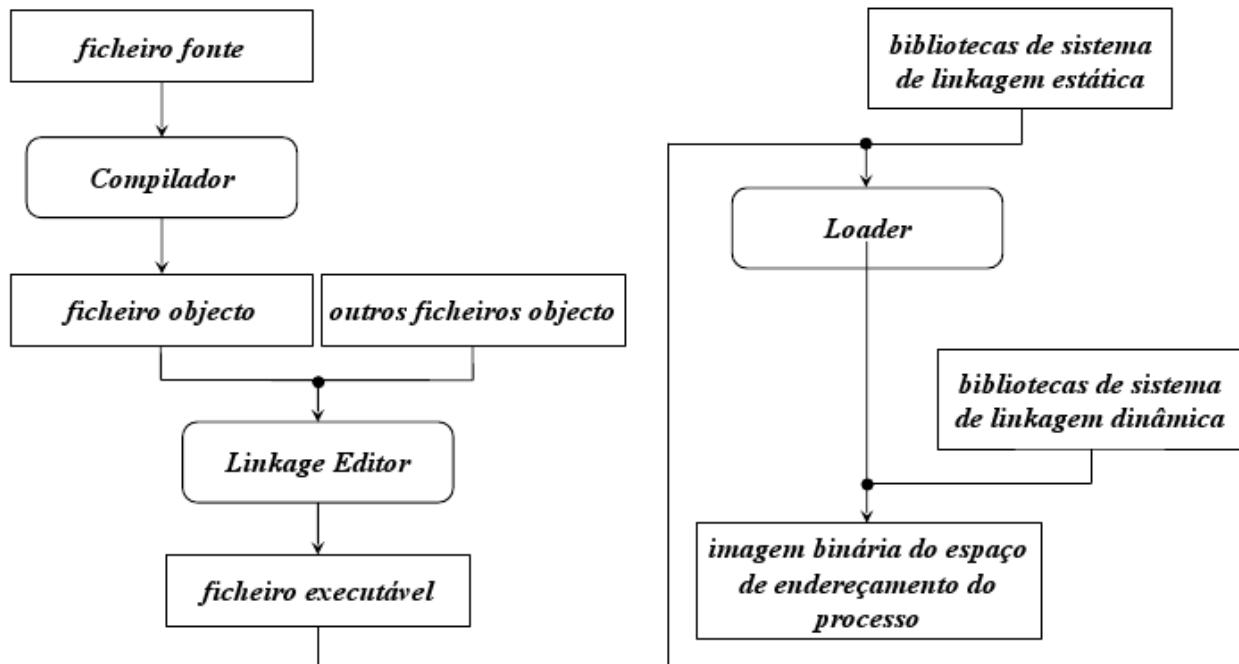
- **registo** das partes da memória principal que estão **ocupadas** e aquelas que estão **livres**;
- **reserva** de porções da memória principal para os processos que dela vão precisar e a sua **libertaçāo** [quando já não são necessárias];
- a **transferência para a área de swapping** de todo o, ou parte do, espaço de endereçamento de um processo quando a memória principal é demasiado pequena para conter todos os processos que coexistem.

Além disso a gestão de memória deve ter em consideração

- a relocalização do espaço de endereçamento de um processo
- a protecção de acessos
- a partilha
- a organização lógica da memória principal.



Etapas da produção de um programa



- os ficheiros objecto, resultantes do processo de compilação, são ficheiros relocalizáveis (**relocatáveis**) – os endereços das diversas instruções e das constantes e variáveis são calculados a partir do início do módulo que tem por convenção o endereço 0;
- o papel do processo de *linkagem* é juntar 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;

por oposição à situação anterior, designada de **linkagem estática**, alguns sistemas de operação suportam a chamada **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);
- assim, 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 directamente;

- neste tipo de esquema, todos os processos que utilizam uma mesma biblioteca de sistema, executam a mesma cópia do código, minimizando portanto a ocupação da memória principal;

ficheiro fonte

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

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

produção do ficheiro objecto

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

produção do ficheiro executável

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

```
$ 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 1    df *ABS*  00000000 hello.c
00000000 1    d .text      00000000
00000000 1    d .data      00000000
00000000 1    d .bss       00000000
00000000 1    d .rodata    00000000
00000000 1    d .note.GNU-stack 00000000
00000000 1    d .comment   00000000
00000000 g    F .text      0000002f main
00000000          *UND*  00000000 puts

RELOCATION RECORDS FOR [.text]:
OFFSET      TYPE            VALUE
0000001f R_386_32        .rodata
00000024 R_386_PC32      puts

Contents of section .rodata:
0000 48656c6c 6f2c2077 6f726c64 210a0000  hello world!.....

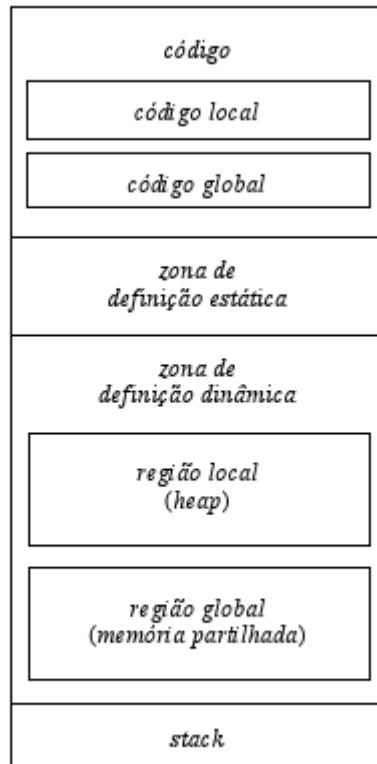
Contents of section .comment:
0000 00474343 3a202847 4e552920 342e302e  .GCC: (GNU) 4.0.
0010 33202855 62756e74 7520342e 302e332d  3 (Ubuntu 4.0.3-
0020 31756275 6e747535 2900                 lubuntu5).
$
```

```
$ 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 0x080482c0

DYNAMIC SYMBOL TABLE:
00000000 DF *UND*    0000017a GLIBC_2.0      puts
00000000 DF *UND*    000000e6 GLIBC_2.0      __libc_start_main
0804847c g  DO .rodata 00000004 Base          _IO_stdin_used
00000000 w   D *UND*    00000000                 _Jv_RegisterClasses
00000000 w   D *UND*    00000000                 __gmon_start__DYNAMIC

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE           VALUE
08049570 R_386_GLOB_DAT  __gmon_start__
08049580 R_386_JUMP_SLOT puts
08049584 R_386_JUMP_SLOT __libc_start_main
$
```



Zona de definição dinâmica inclui:

- região local (**heap**)
- região global (**memória partilhada**)
- 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 o *stack* têm potencialmente um crescimento mais ou menos acentuado durante a execução do processo;
- assim, 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*;

qualquer que seja o caso, porém, a imagem binária do espaço de endereçamento do processo representa um espaço de endereçamento relocável, 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; dois problemas têm que ser resolvidos

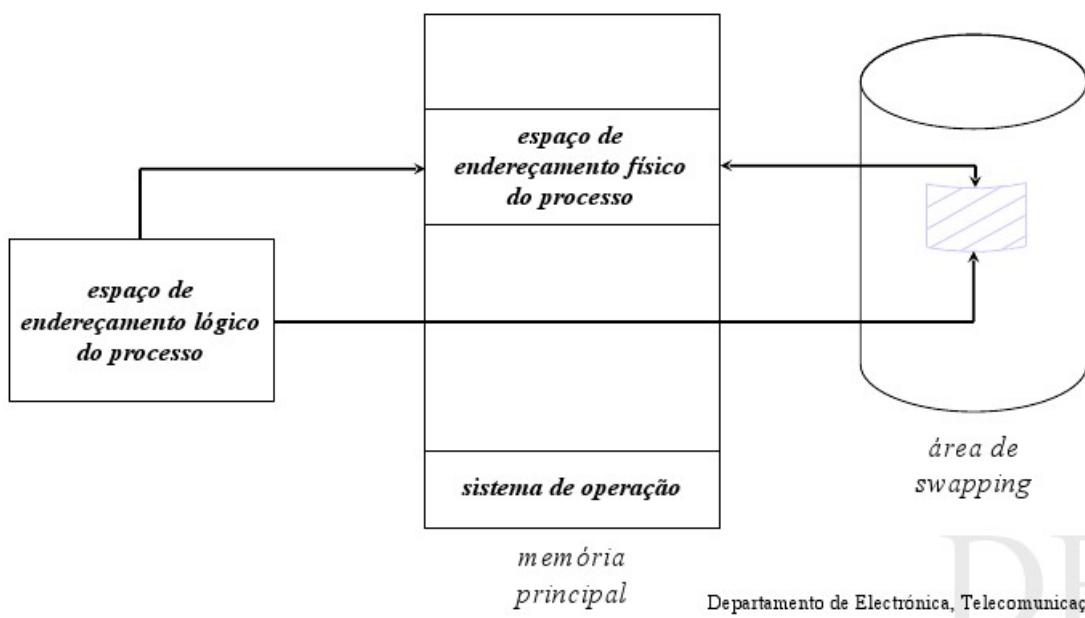
- **relocação dinâmica** – a capacidade de conversão em *run time* de um endereço

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

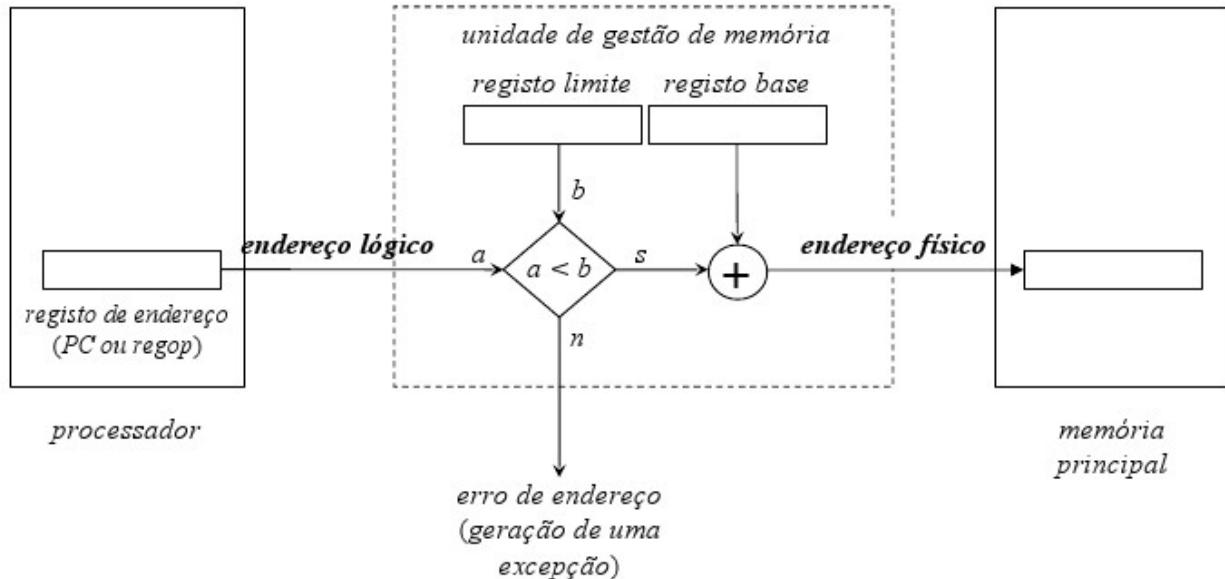
A política de reserva de espaço em memória principal para armazenamento do espaço de endereçamento dos processos que correntemente coexistem, diz-se originar uma organização de memória real quando existe uma correspondência biunívoca (de um para um) entre o espaço de endereçamento lógico de um processo e o seu espaço de endereçamento físico.



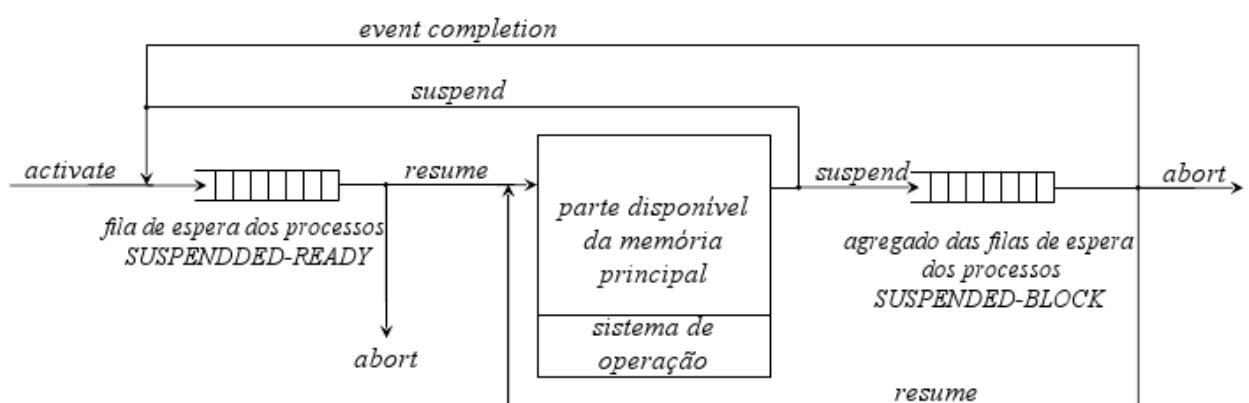
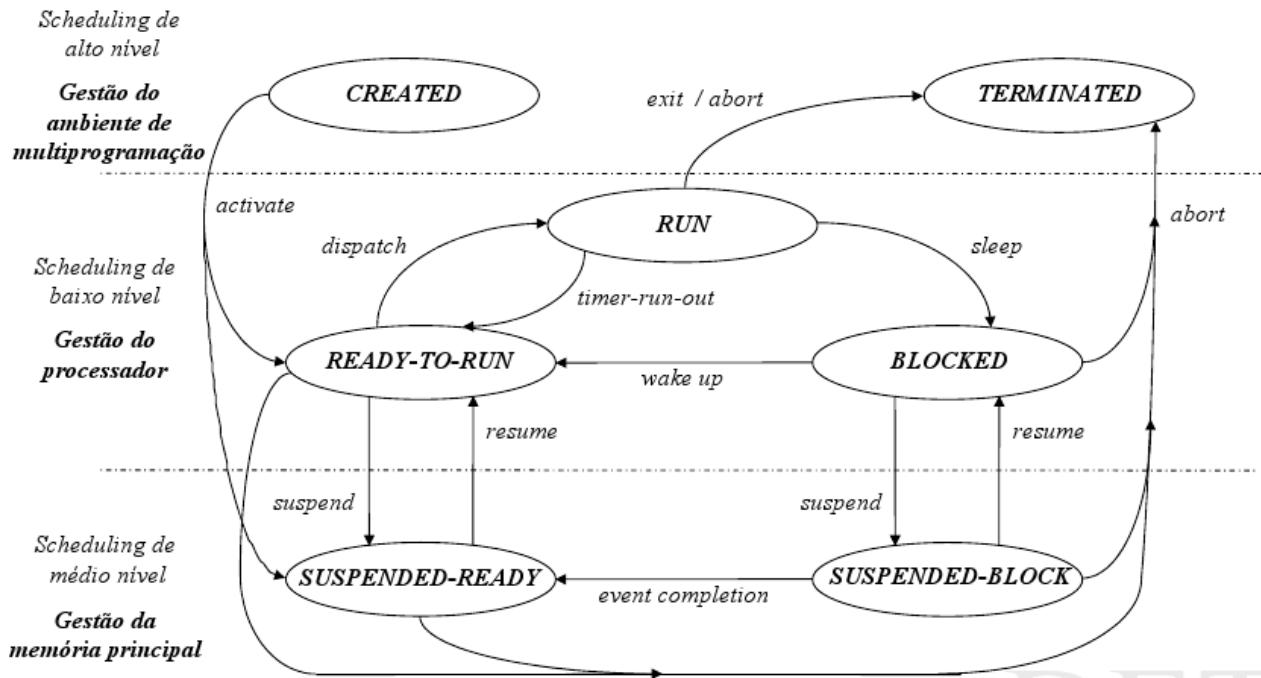
As consequências desta política são de três tipos:

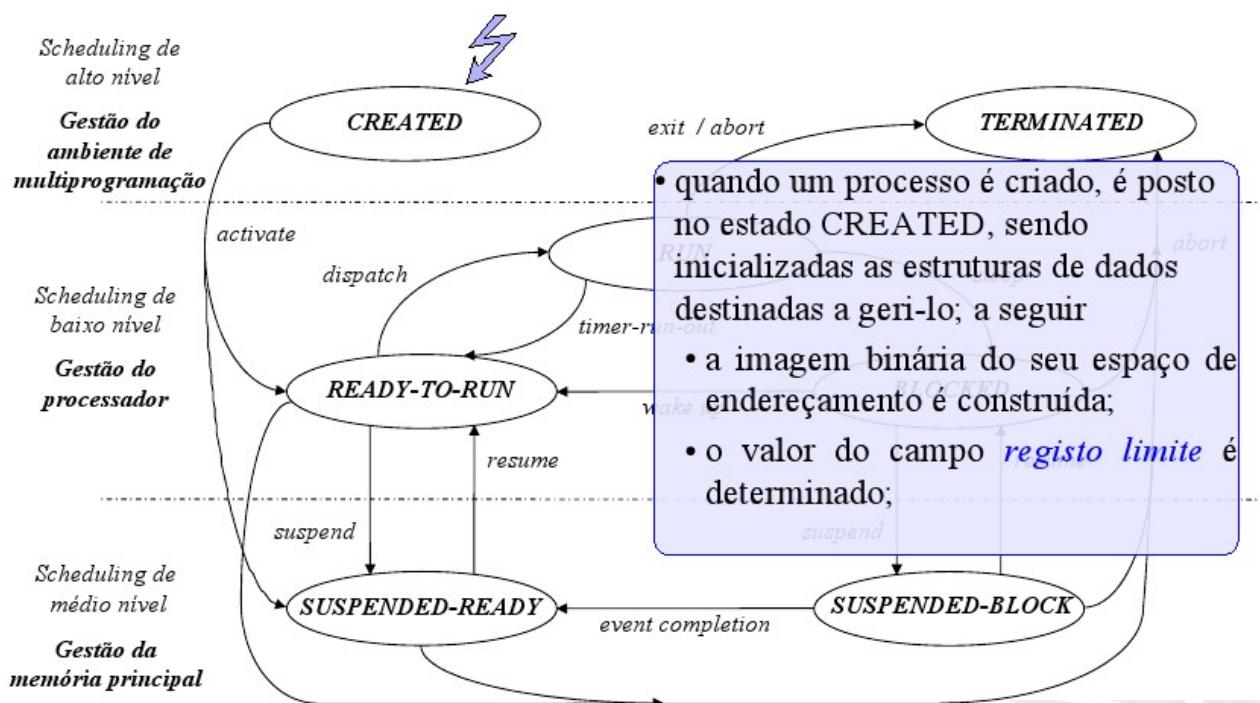
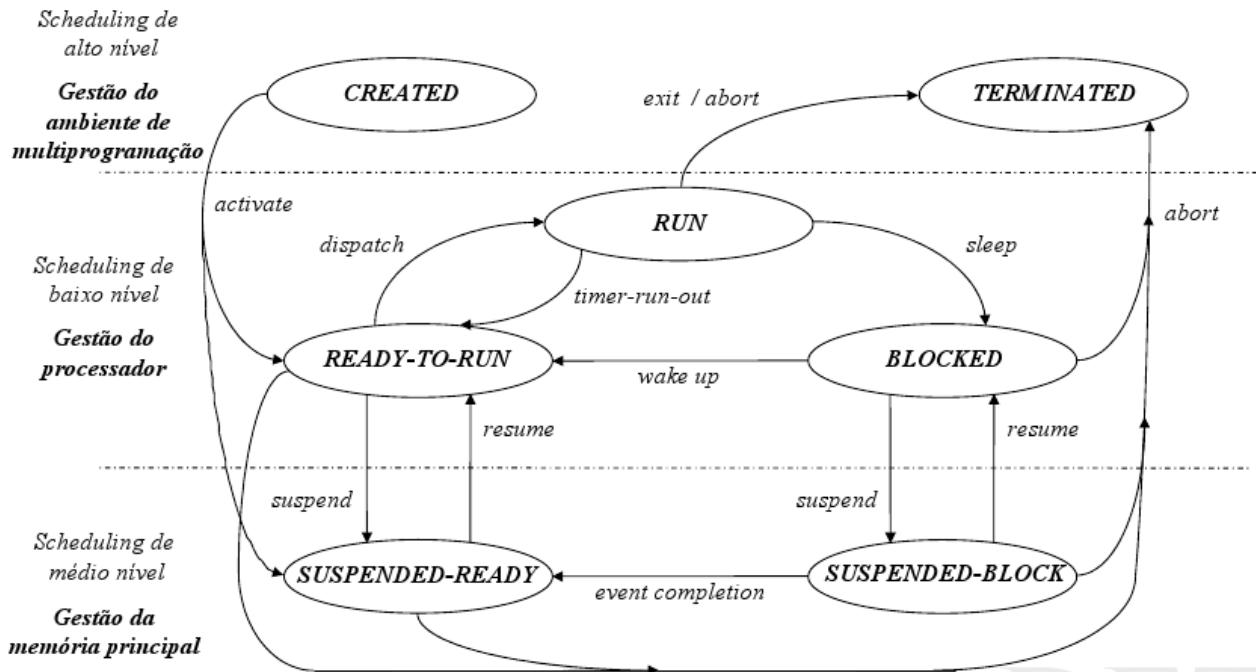
- **limitação do espaço de endereçamento de um processo** - em nenhum caso a gestão de memória pode suportar mecanismos automáticos que permitam que o espaço de endereçamento de um processo seja superior ao tamanho da memória principal disponível;
- **contiguidade do espaço de endereçamento físico** - 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 estar directamente carregados em memória principal por falta de espaço.

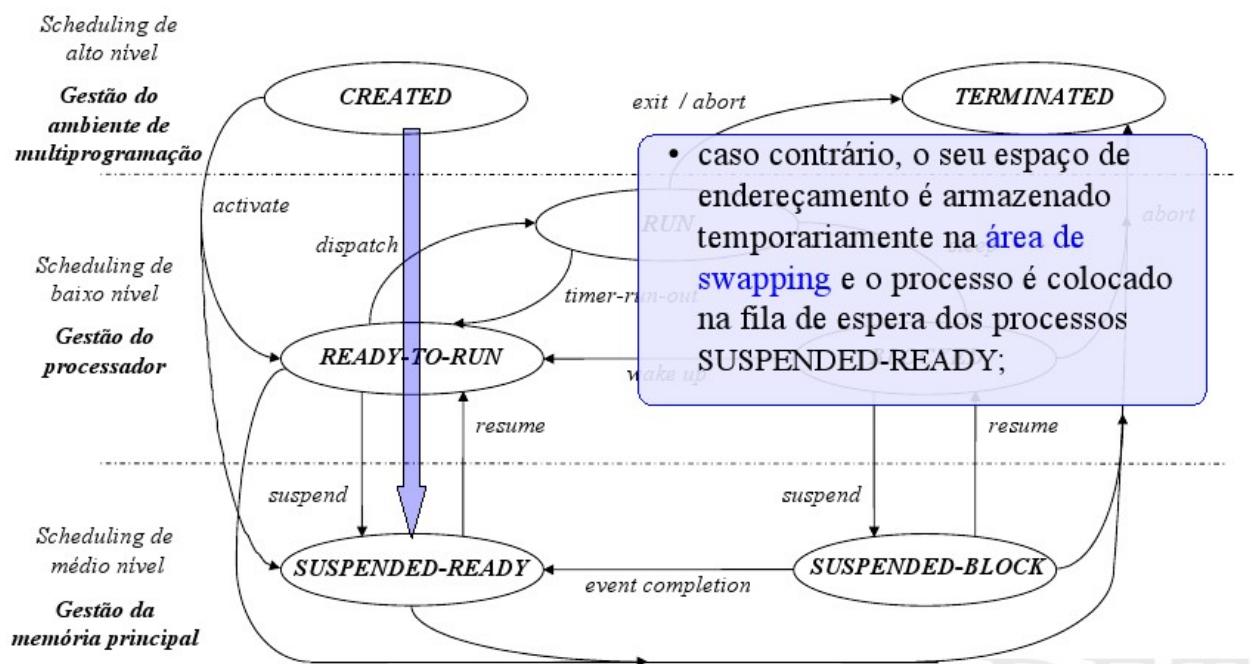
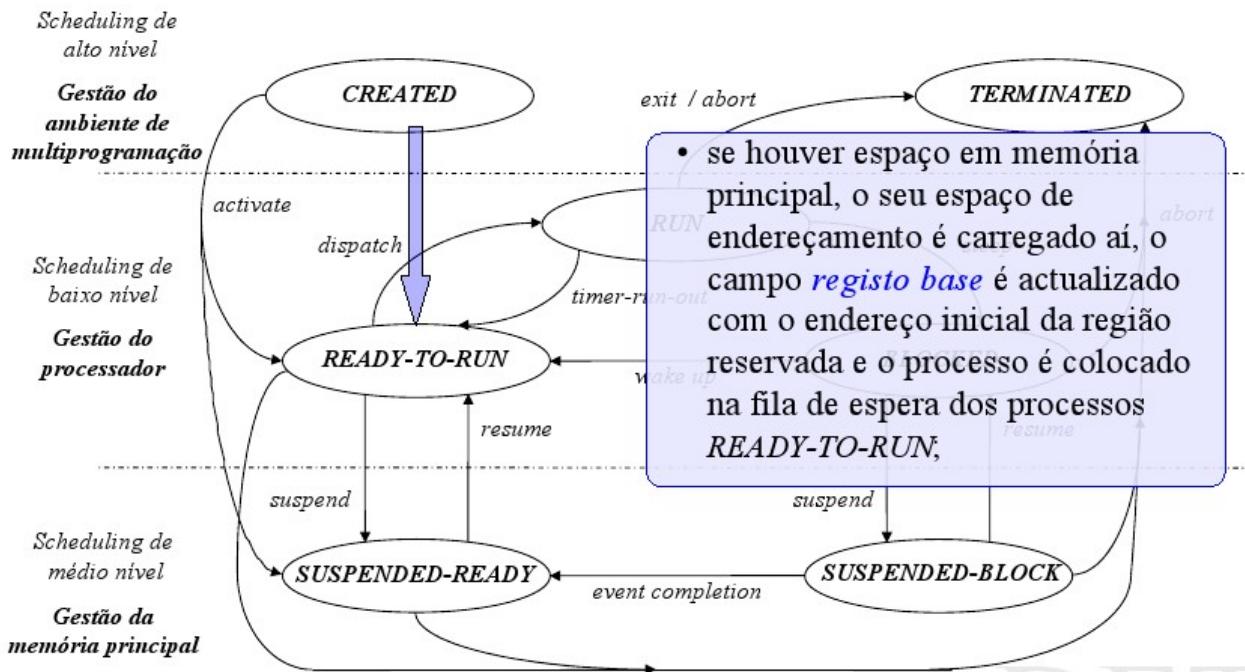
Tradução de um endereço lógico num endereço físico



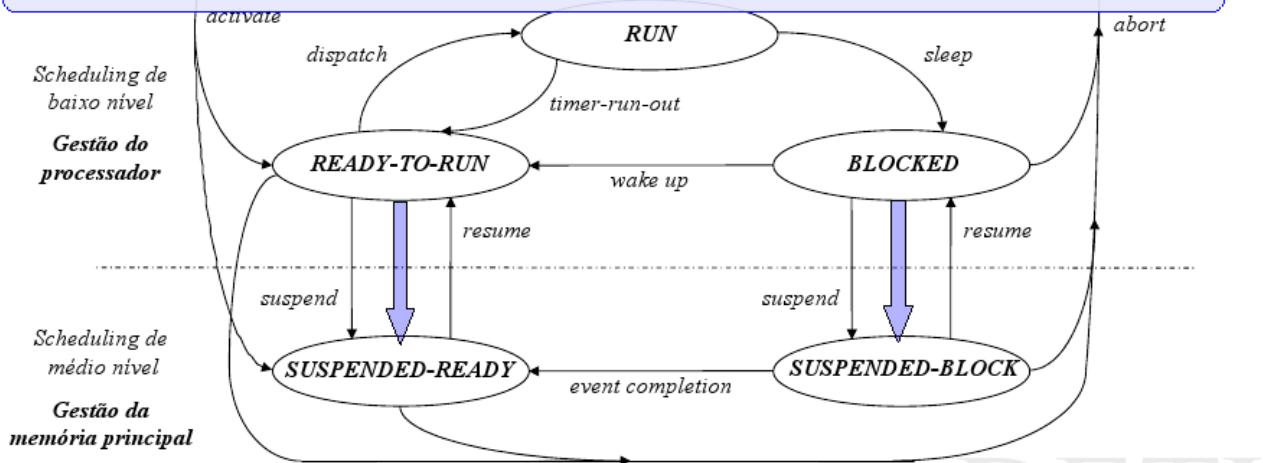
- quando ocorre uma comutação de processos, a operação *dispatch* carrega o registo base e o registo limite com os valores presentes nos campos correspondentes da entrada da tabela de controlo de processos associada com o processo que vai ser calendarizado para execução;
- o valor do registo base representa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



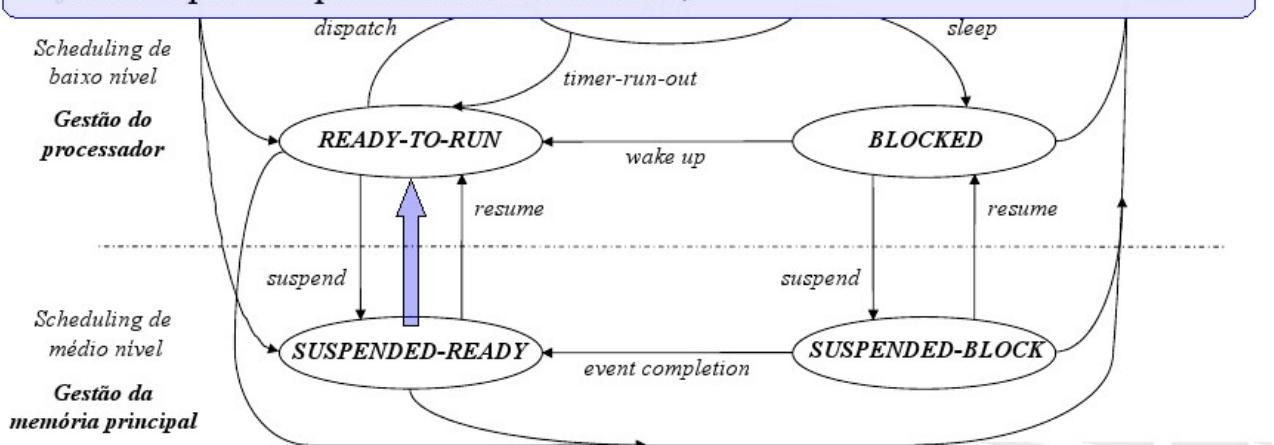




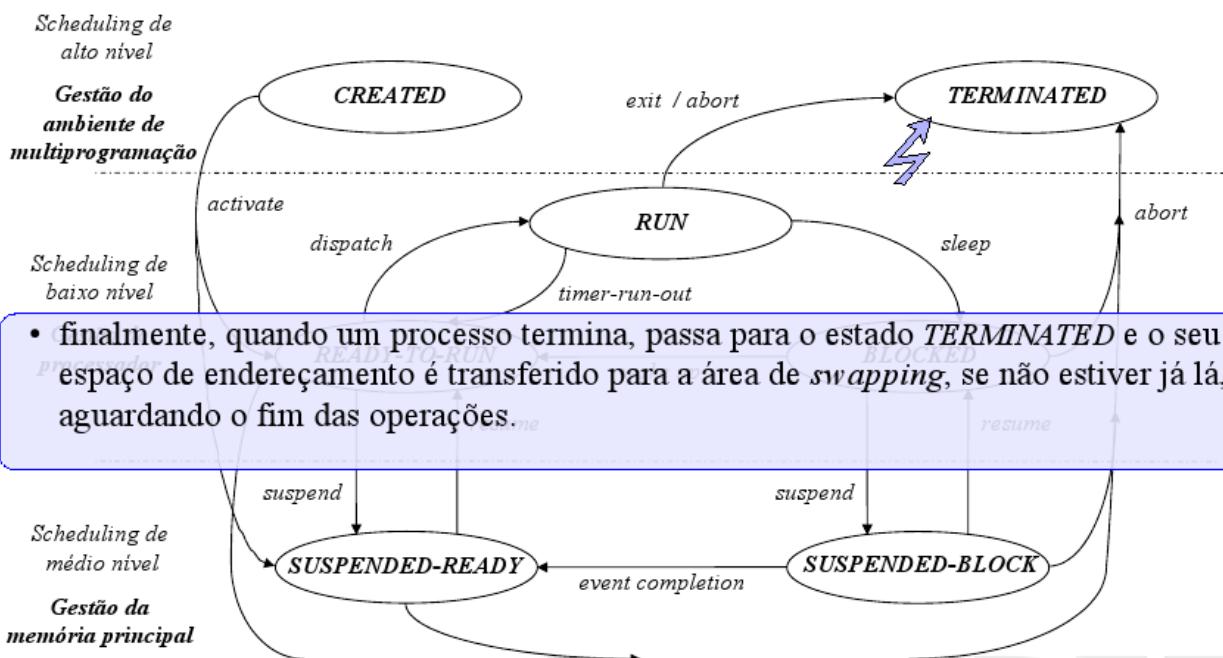
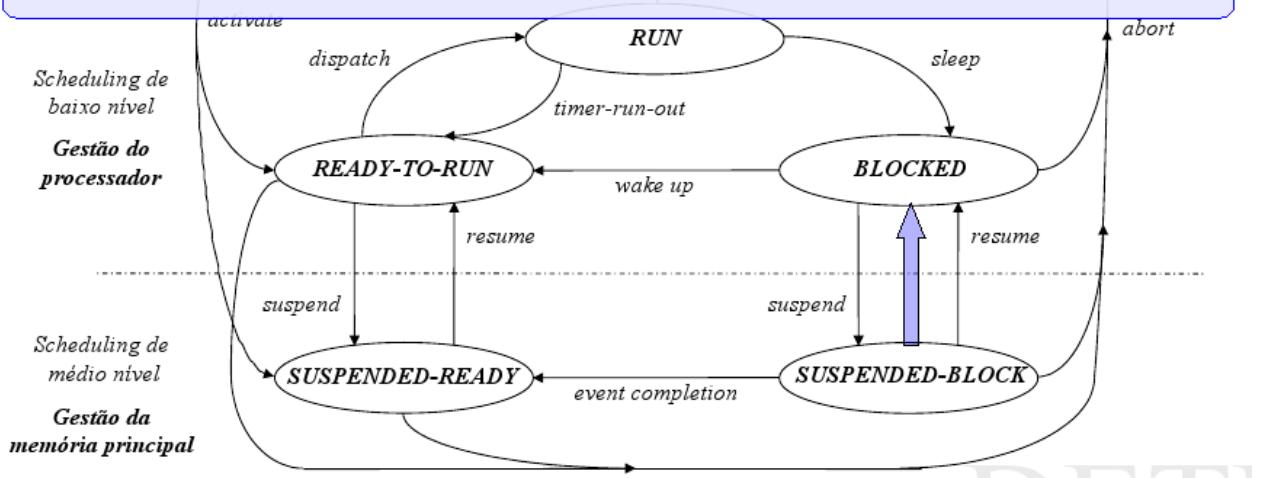
- ao longo da sua execução, o espaço de endereçamento de um processo pode ser deslocado temporariamente para a *área de swapping*, passando o seu estado de **READY-TO-RUN** para **SUSPENDED-READY**, ou de **BLOCKED** para **SUSPENDED-BLOCK**;



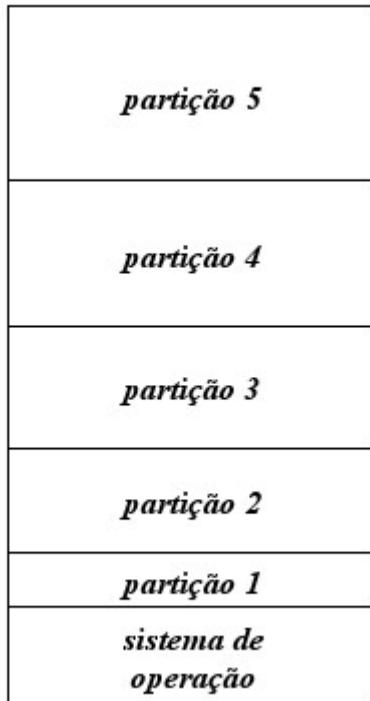
- Sempre que há espaço na memória, um dos processos presentes na fila de espera dos processos **SUSPENDED-READY** é seleccionado, o seu espaço de endereçamento é carregado, o campo **registro base** da entrada da tabela de controlo de processos é actualizado com o endereço inicial da região reservada 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;



Arquitectura 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 físico de um só processo;

diferentes disciplinas de *scheduling* podem ser usadas na selecção do processo cujo espaço de endereçamento vai ser carregado a seguir em memória principal; entre elas destacam-se

- **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 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 selecção por um processo maior; ao atingir-se um valor pré-definido, o processo já não pode ser descartado e a primeira regra é aplicada.

Vantagens

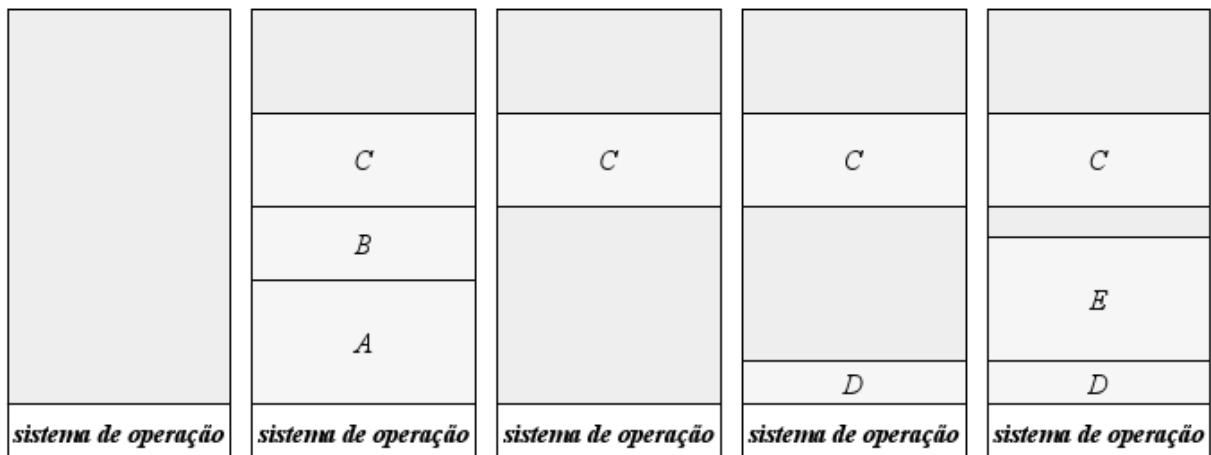
- **simples de implementar** - não exige hardware ou estruturas de dados especiais;
- **eficiente** - a selecção pode ser feita muito rapidamente.

Desvantagens

- **conduz a uma grande fragmentação interna da memória principal** – a parte de cada partição que não contém o 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.

Arquitectura 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.



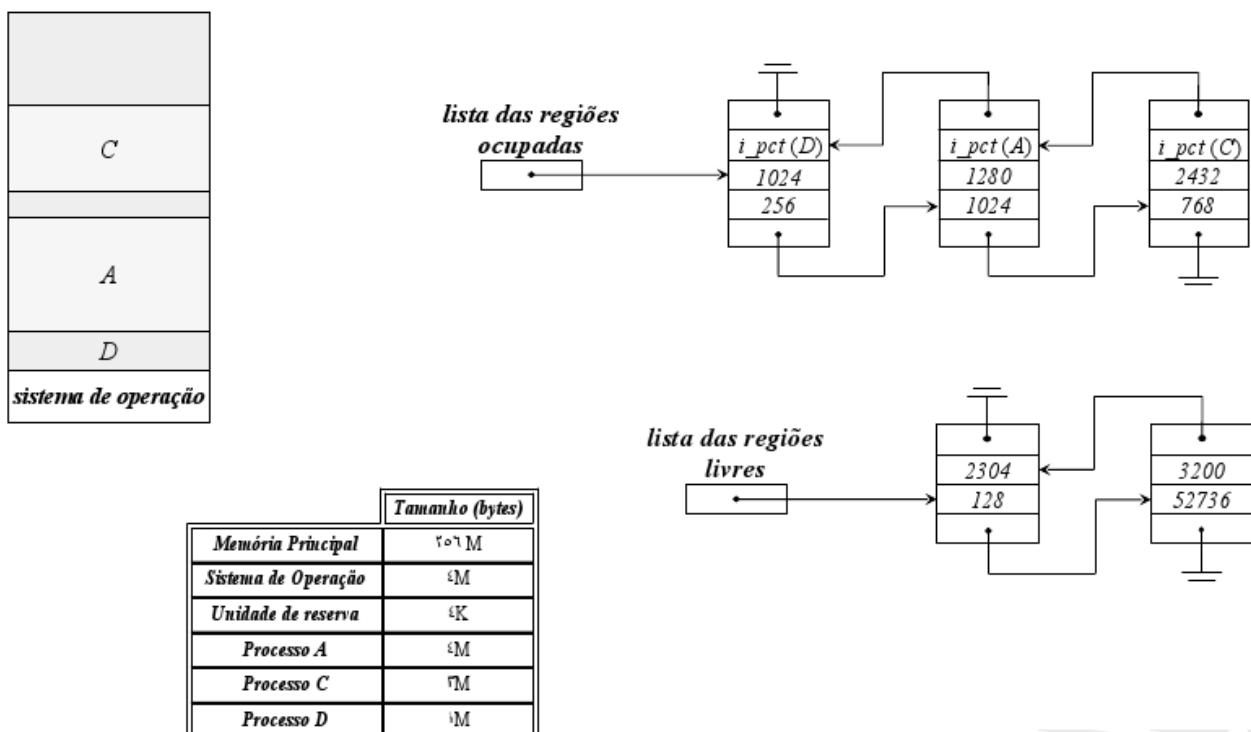
como a memória é reservada dinamicamente, o sistema de operação tem que manter um registo actualizado das regiões ocupadas e das regiões livres;

uma maneira de fazer isso é construindo duas listas biligadas

- **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;



- a valorização do critério de justiça é a disciplina de *scheduling* geralmente adoptada, 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 arquitectura 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 efectuar 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;

- assim, 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 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!

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 biligadas.

Desvantagens

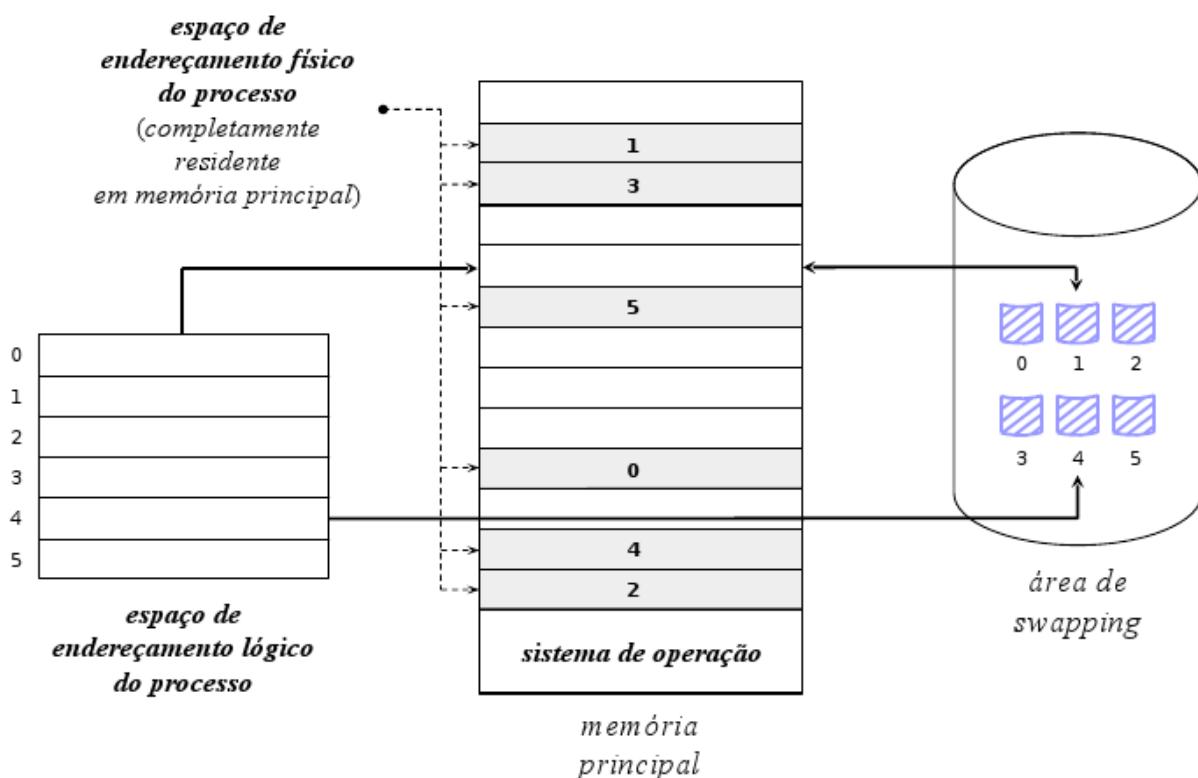
- conduz a uma grande **fragmentação externa da memória principal** – a fracção da memória principal que acaba por ser desperdiçada, face ao tamanho reduzido das regiões em que está dividida, pode atingir em alguns casos cerca de um terço do total (regra dos 50%);
- **pouco eficiente** – não é possível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

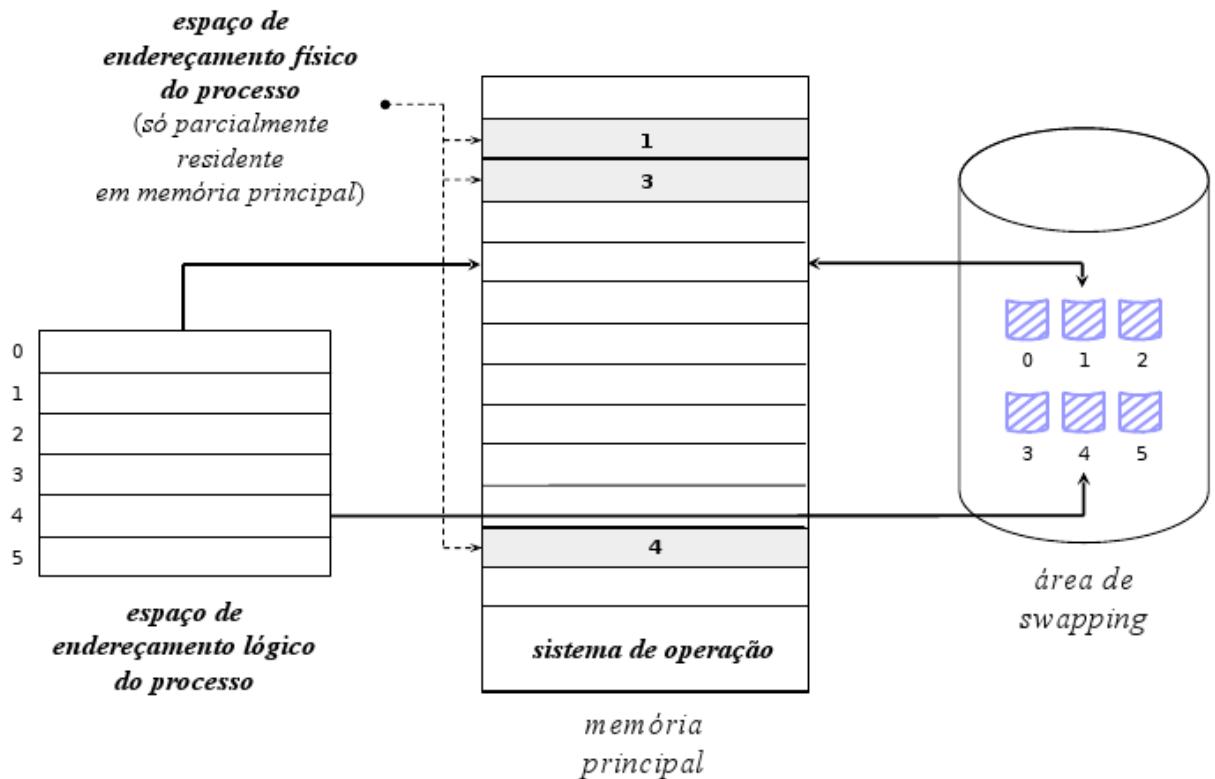
Organização de memória virtual

A política de reserva de espaço em memória principal para armazenamento do espaço de endereçamento dos processos que correntemente coexistem, diz-se originar uma organização de memória virtual quando o espaço de endereçamento lógico de um processo e o seu espaço de endereçamento físico estão totalmente dissociados.

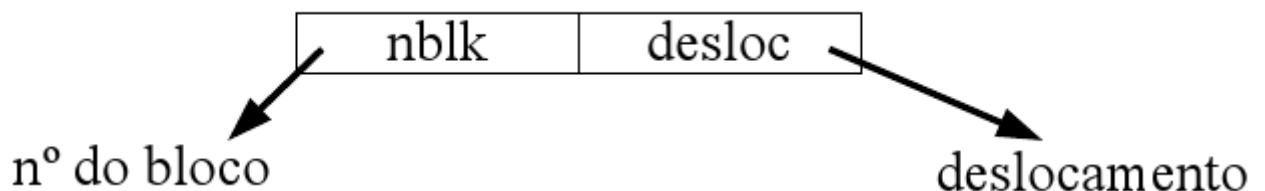
As consequências desta política são de três tipos

- **não limitação do espaço de endereçamento de um processo** – espaço de endereçamento de um processo pode ser 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, podem estar 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 actualizada dos espaços de endereçamento dos processos que correntemente coexistem, nomeadamente da sua parte variável (zonas de definição estática e dinâmica e *stack*).

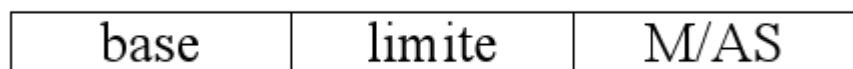




- O espaço de endereçamento lógico de um processo está dividido num conjunto de blocos.
- Um endereço lógico compõe-se em duas componentes: a referência a um bloco e um deslocamento dentro do bloco:



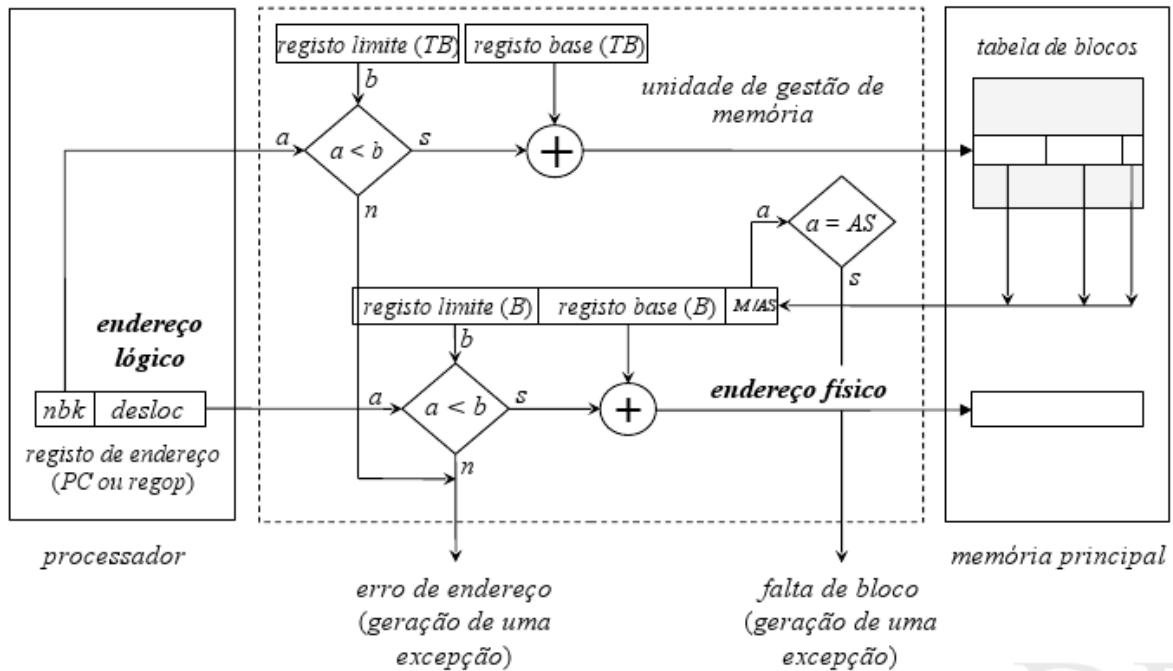
- A tabela de blocos representa o mapeamento dos blocos em regiões da memória real.
 - Há uma entrada na tabela por cada bloco;
 - A tabela é caracterizada por um endereço base (TB-base) e pelo número de entradas (TB-limite).
- Cada entrada na tabela de blocos caracteriza o bloco em: endereço base, dimensão e indicação do seu estado de swap in/out.



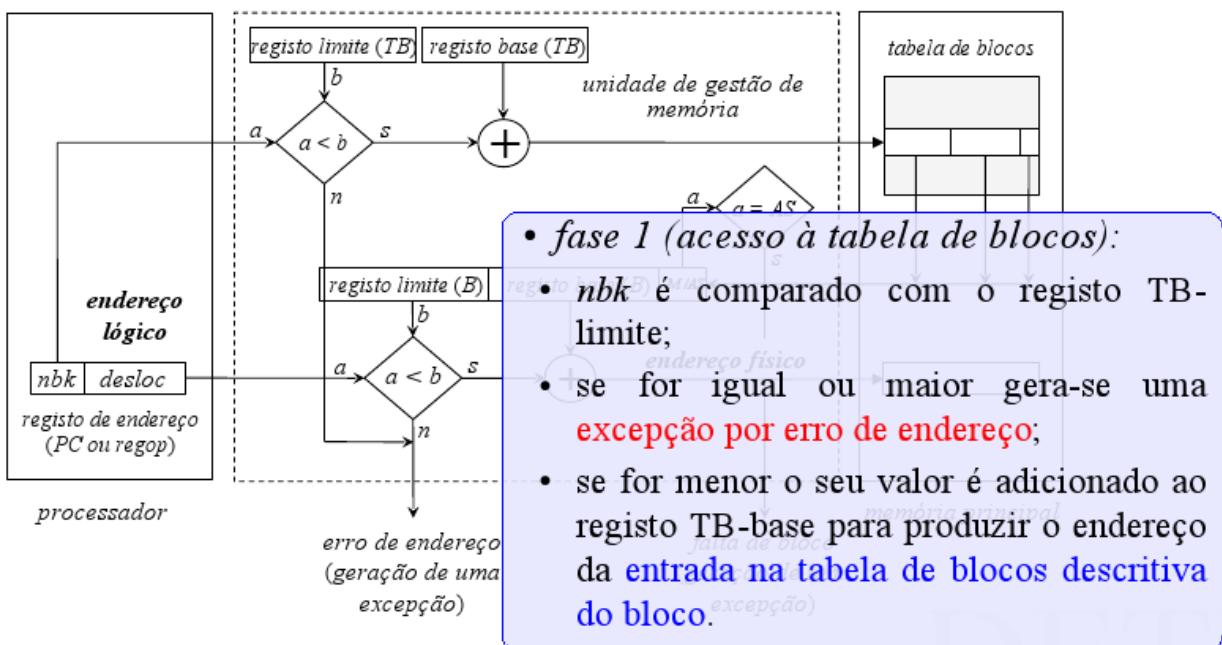
- endereço base: representa o endereço base do bloco em memória principal
- dimensão do bloco: representa o endereço limite dentro do bloco
- M/AS: indica se o bloco se encontra em memória ou em área de swap.

Organização de memória virtual: acesso à memória

Tradução de um endereço lógico num endereço físico

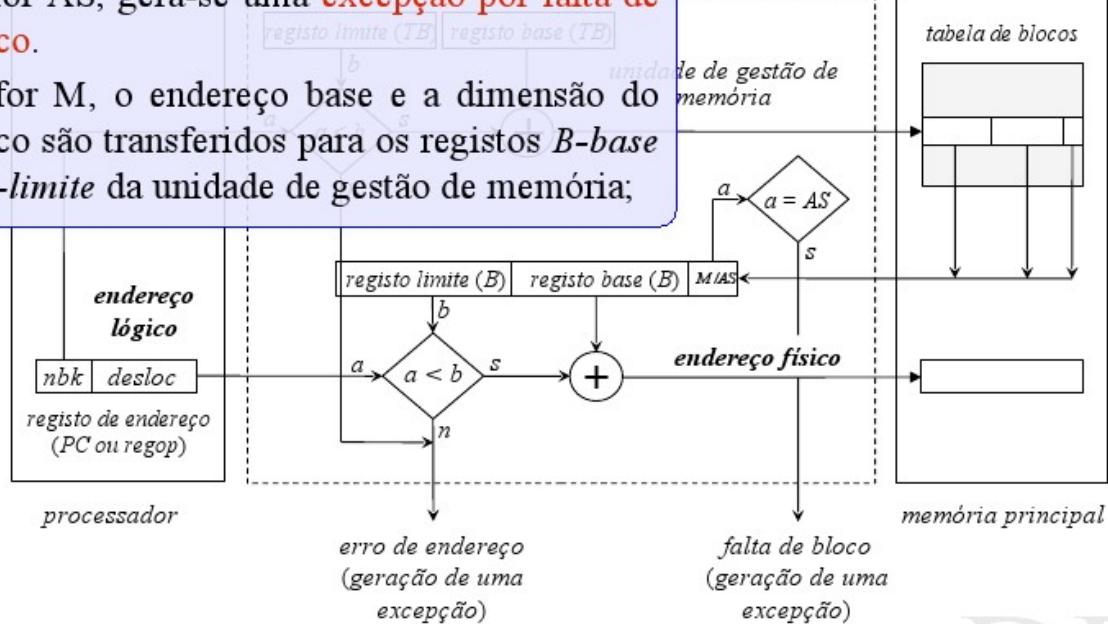


Tradução de um endereço lógico num endereço físico



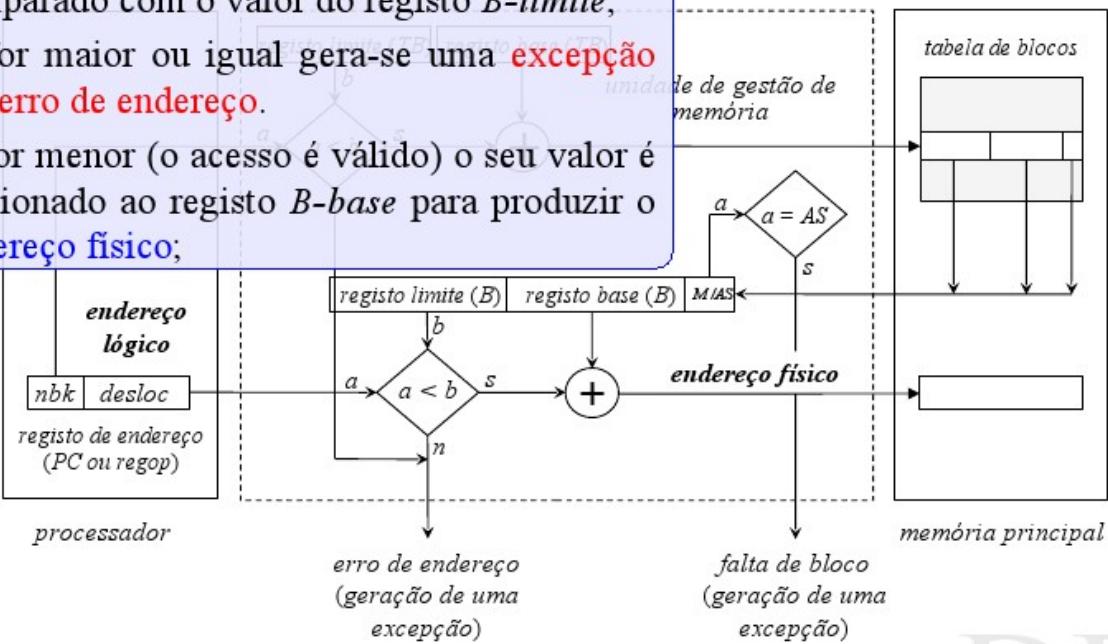
• fase 2 (swap in):

- o campo M/AS é avaliado;
- se for AS, gera-se uma **excepção por falta de bloco**.
- se for M, o endereço base e a dimensão do bloco são transferidos para os registos *B-base* e *B-limite* da unidade de gestão de memória;

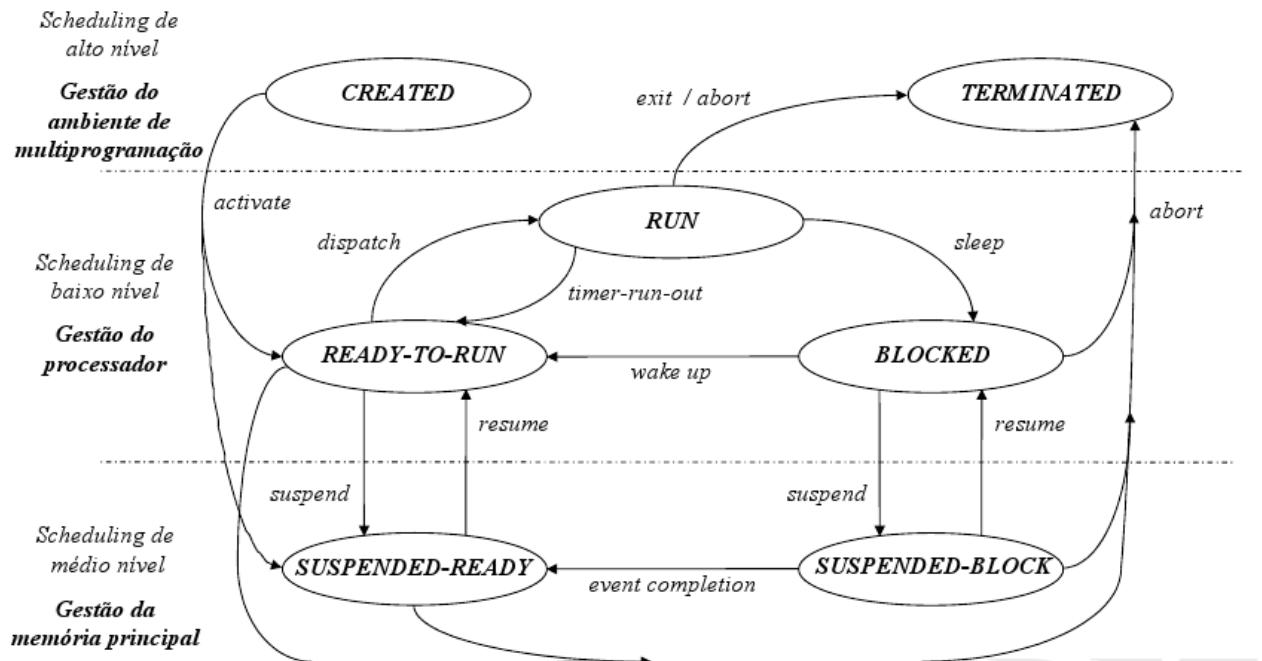


• fase 3 (geração do endereço físico):

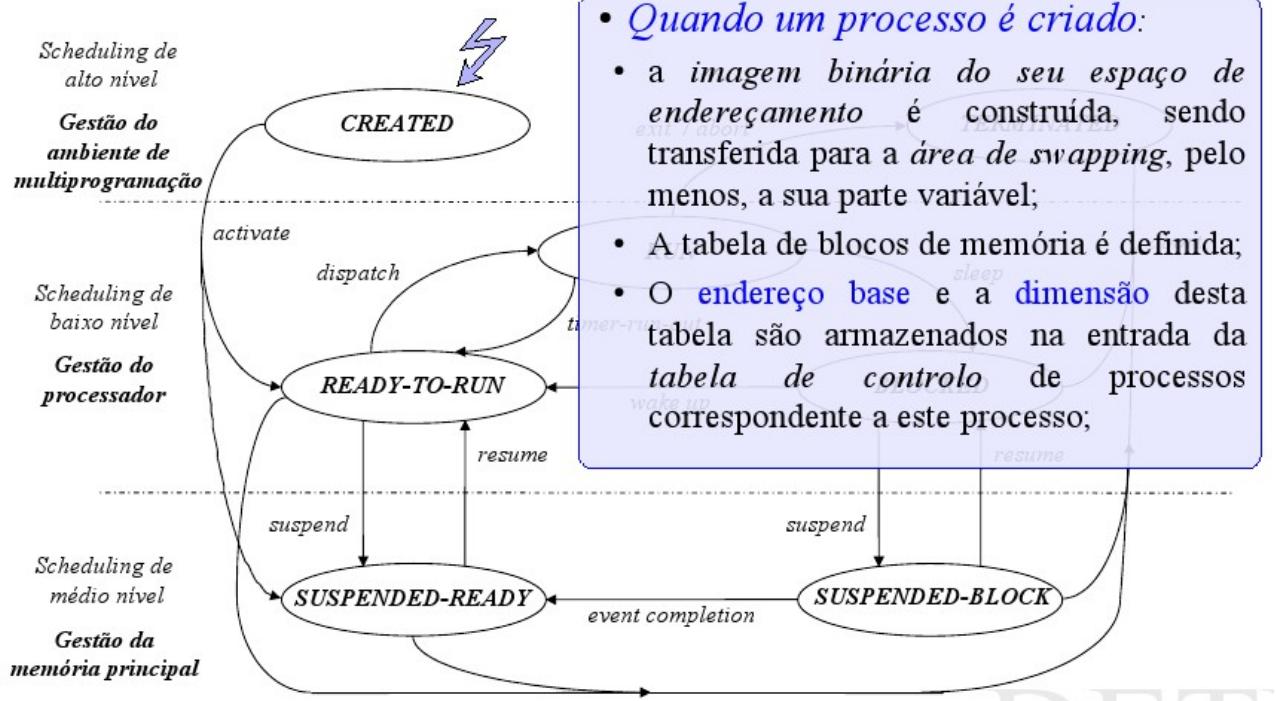
- o campo *desloc* do *endereço lógico* é comparado com o valor do registo *B-limite*;
- se for maior ou igual gera-se uma **excepção por erro de endereço**.
- se for menor (o acesso é válido) o seu valor é adicionado ao registo *B-base* para produzir o *endereço físico*;

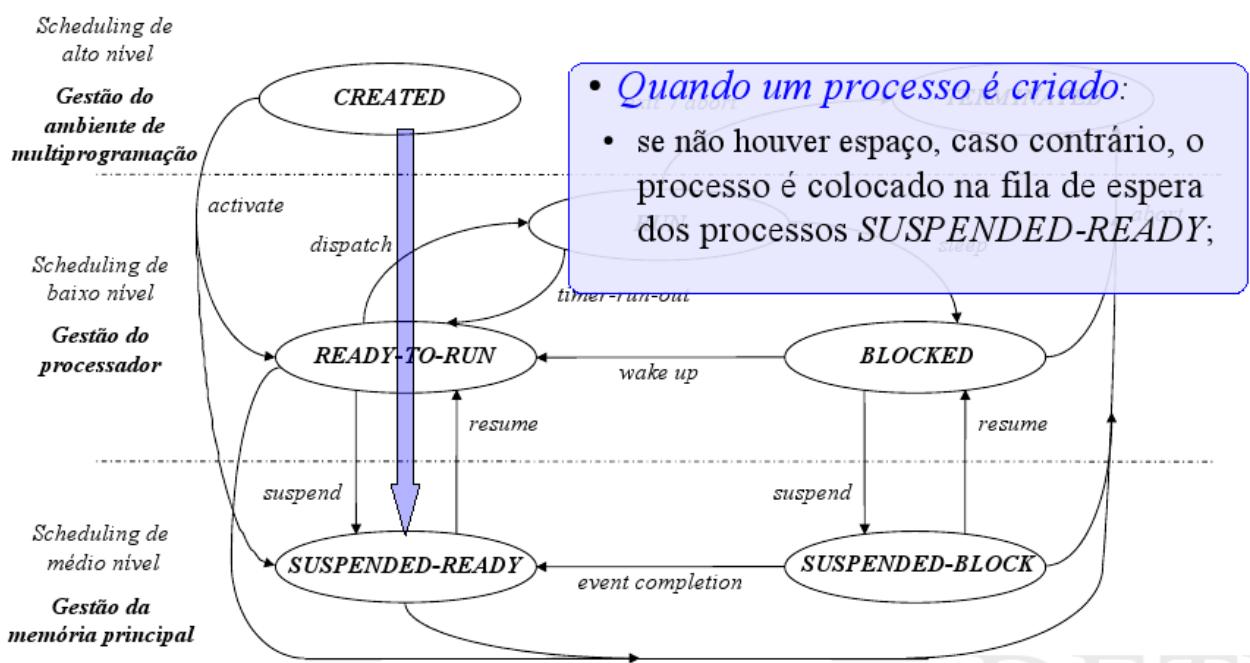
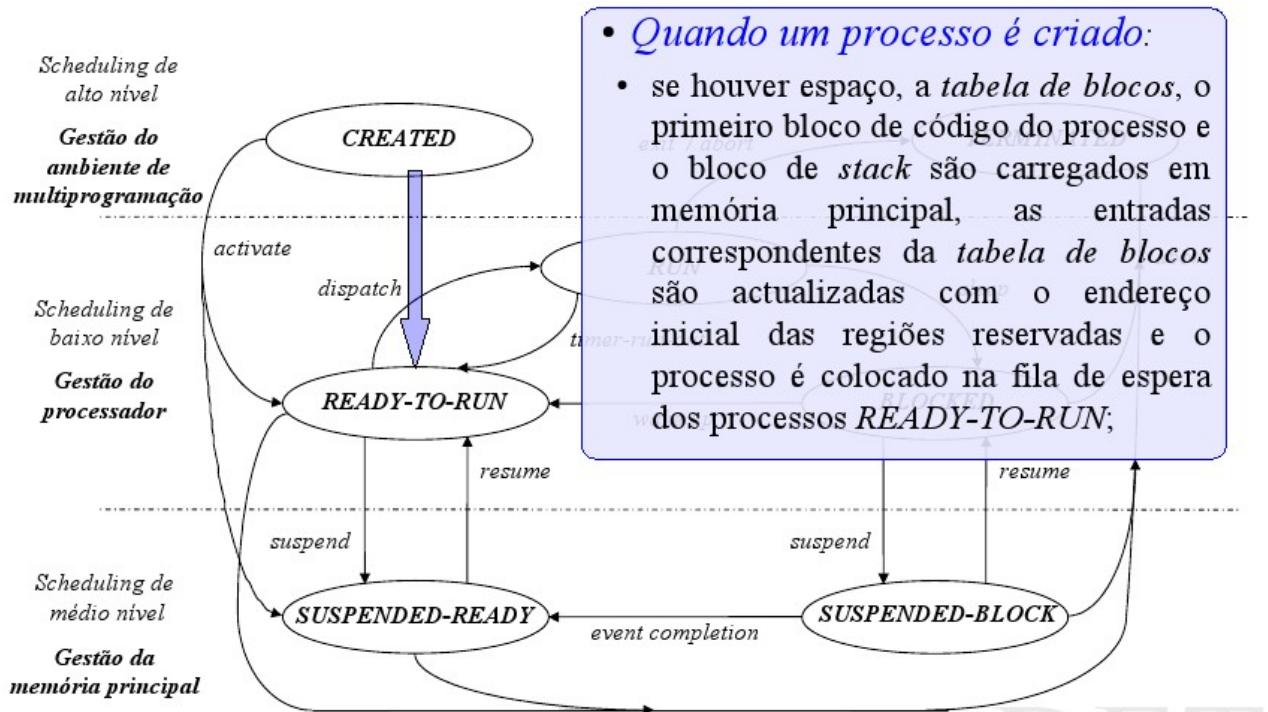


Organização de memória virtual: estado CREATED



Organização de memória real - continuação





Organização de memória virtual: estado READY-TO-RUN

Se o processo é posto em execução (**dispatch**):

- Os registos TB-base e TB-limite da unidade de gestão de memória são carregados com os valores do endereço base e a dimensão da tabela de blocos do processo;

Se o processo é suspenso:

- Todos os blocos residentes em memória principal são transferidos para a área de *swaping*, sendo as entradas da tabela de blocos actualizadas em conformidade.

Organização de memória virtual: excepção por falta de bloco

- salvaguardar o contexto do processo na entrada correspondente da tabela de controlo de processos, colocando o seu estado em BLOCKED
- determinar se existe espaço em memória principal para carregar o bloco em falta
 - caso exista, seleccionar uma região livre;
 - caso não exista, seleccionar uma região cujo bloco vai ser substituído:
 - se o bloco tiver sido modificado, proceder à sua transferência para a área de *swapping*;
 - actualizar 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 *scheduller* para calendarizar para execução um dos processos da fila de espera dos processos READY-TO-RUN;
- quando a transferência estiver concluída, actualizar 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.

Organização de memória virtual: análise crítica

- Aumento de versatilidade, mas com um custo que se traduz na transformação de cada acesso à memória em dois acessos
 - no primeiro é acedida a tabela de blocos do processo;
 - no segundo é acedida a posição de memória específica.
- A organização de memória virtual resulta num fraccionamento do espaço de endereçamento lógico do processo em blocos que são tratados dinamicamente como sub-espacos 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;
- Há a possibilidade de ocorrer um acesso a um bloco actualmente não residente em memória principal.

Organização de memória virtual: optimização

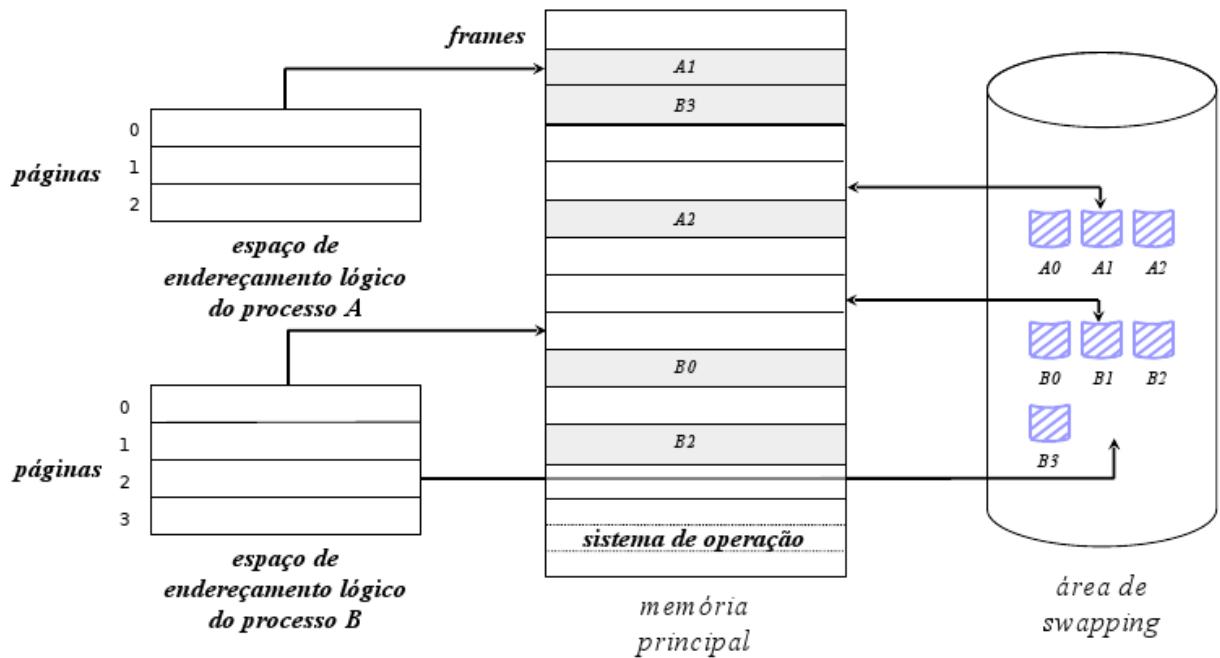
- A necessidade deste duplo acesso à memória pode ser minimizada tirando partido do princípio da localidade de referência.
- Como os acessos tendem a estar concentrados num conjunto bem definido de blocos durante intervalos de tempo alargados de execução do processo, a unidade de gestão de memória mantém habitualmente armazenado numa memória associativa interna, designada de *translation lookaside buffer* (TLB), o conteúdo

- das entradas da tabela de blocos que foram ultimamente referenciadas
- O primeiro acesso pode ser transformado num acesso interno ao processador:
 - há um *hit* se a entrada está em cache;
 - há um *miss* caso contrário.
- 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).

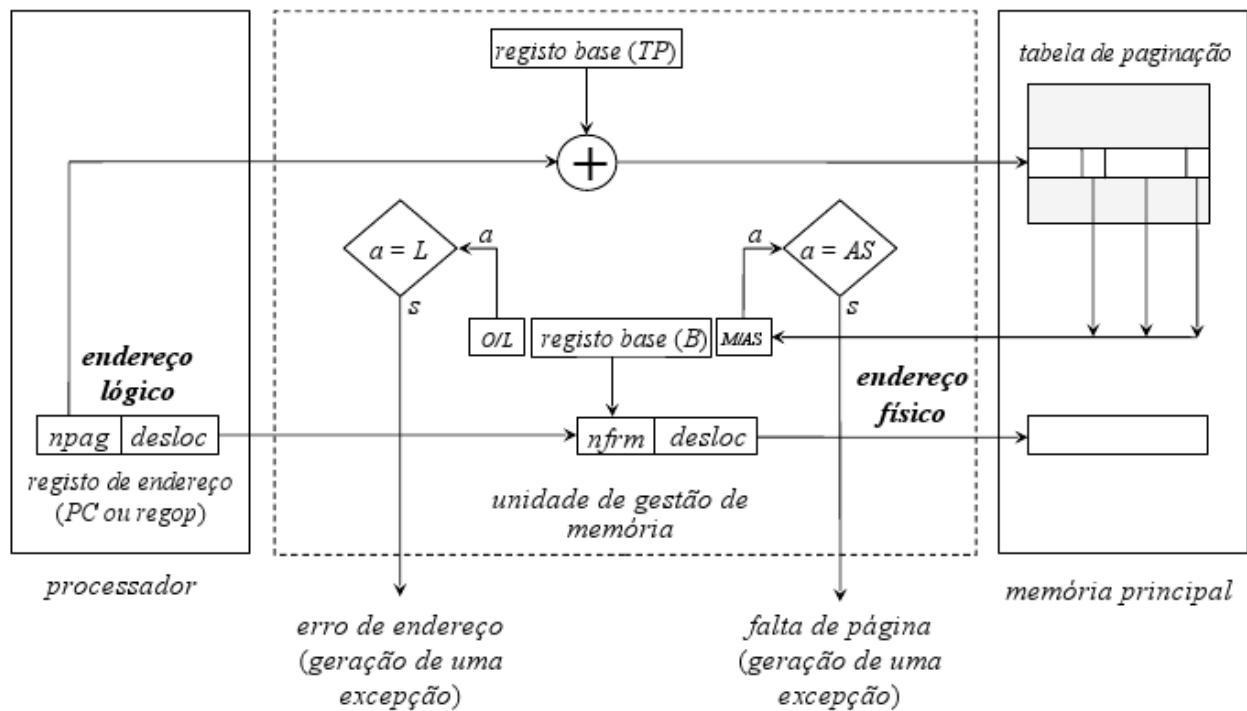
Arquitectura paginada

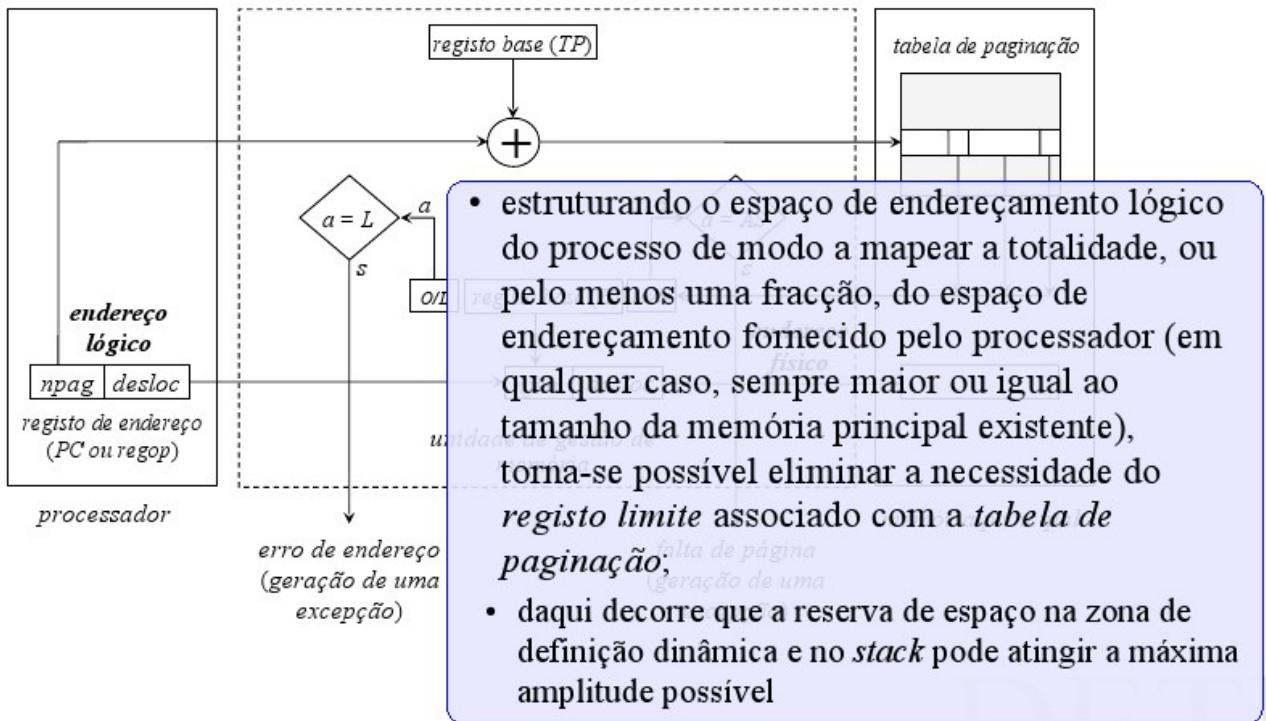


- os blocos, designados aqui de páginas, são todos iguais e têm um tamanho correspondente a uma potência de 2, tipicamente 4 ou 8 KB;
- face a isto, o mecanismo de divisão do espaço de endereçamento lógico do processo é meramente operacional: os bits mais significativos do endereço definem o número da página e os menos significativos o deslocamento;
- a memória principal, por seu lado, é vista também como estando dividida em blocos, os *frames* de memória, 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 do *stack*, o fim).

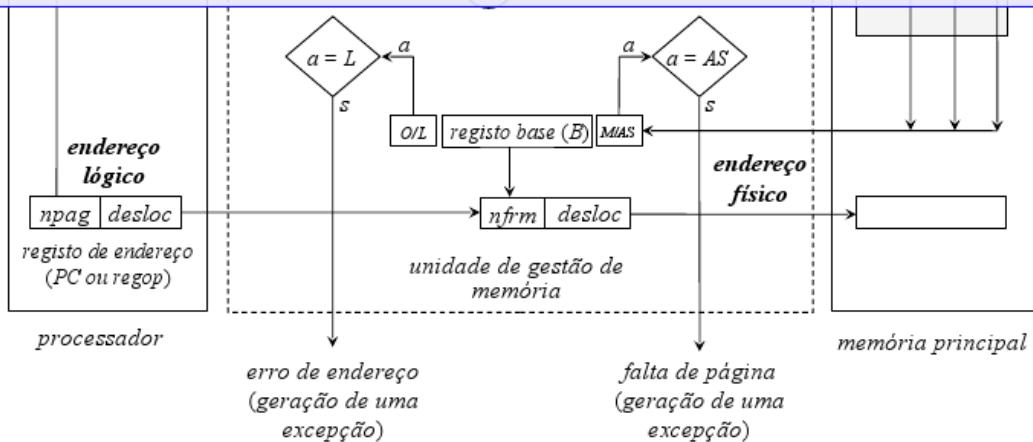


Arquitectura paginada: acesso à memória

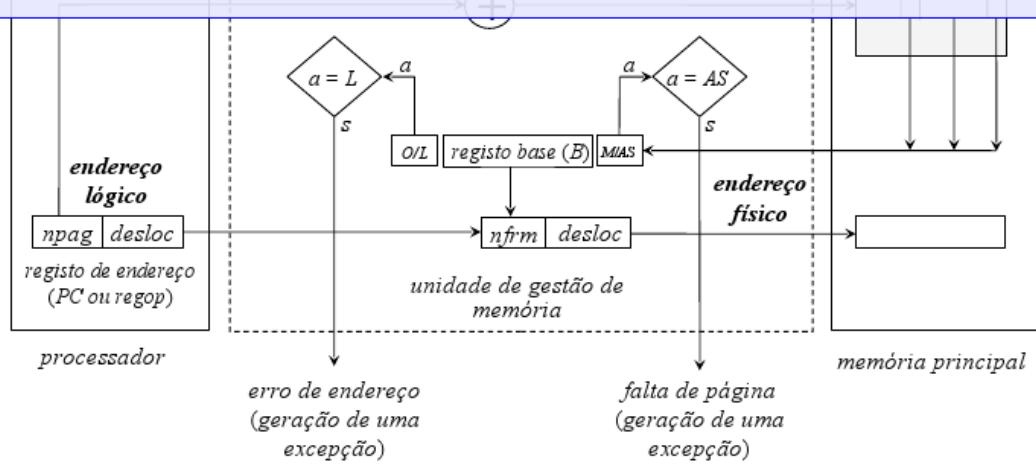




- o **registro limite** associado com uma página particular não existe nunca, pois o endereço físico é formado pela concatenação entre o campo **nfrm**, que identifica o **frame** de memória principal onde a página está localizada, e o campo **desloc**, o deslocamento dentro da página (ou do **frame**,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



- tipicamente, 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



Arquitectura paginada: entrada da tabela de paginação

<i>O/L</i>	<i>M/AS</i>	<i>Ref</i>	<i>Mod</i>	<i>Prot</i>	<i>N. do frame em memória</i>	<i>N. do bloco na área de swapping</i>
------------	-------------	------------	------------	-------------	-------------------------------	--

- 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
- Prot** – indicação do tipo de acesso permitido (*r-only* 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))
- N. do frame em memória** – localização da página se residente em memória principal
- N. do bloco na área de swapping** – localização da página na área de *swapping* se já lhe foi atribuído espaço

Arquitectura paginada: vantagens

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 actuais de uso geral está já preparada para a sua implementação.

Desvantagens

- **acesso à memória mais longo** – cada acesso à memória transforma-se num duplo acesso por consulta prévia da tabela de paginação; contudo, este aspecto pode ser minimizado se a unidade de gestão de memória contiver uma memória associativa, o chamado *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 eficiência.

Arquitectura segmentada

A arquitectura paginada divide o espaço de endereçamento lógico do processo de uma forma cega, praticamente sem usar qualquer informação sobre a estrutura subjacente. A excepção é, como foi referido atrás, a prática actual de colocar sempre no início de uma nova página (fim, no caso do *stack*) cada uma das regiões funcionalmente distintas.

Daqui decorrem 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.

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 arquitectura paginada, numa multiplicidade espaços de endereçamento lineares autónomos definidos na fase de *linkagem*.

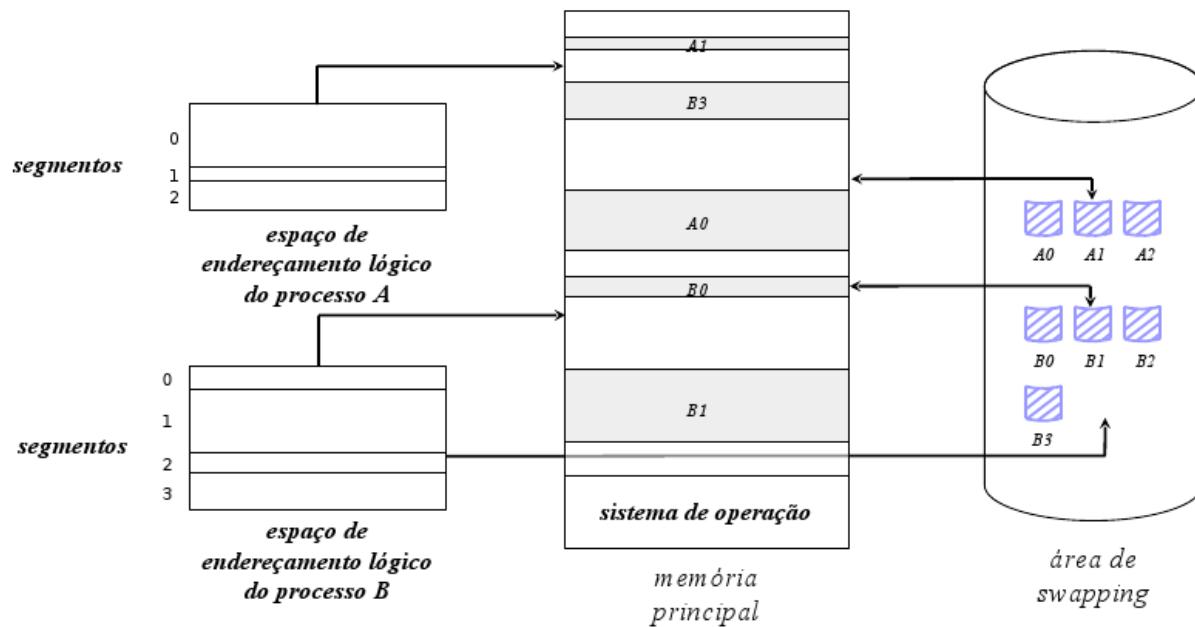
Assim, cada módulo da aplicação (ficheiro em código fonte de compilação separada) vai originar dois espaços de endereçamento autónomos: um para o código e outro, na zona de definição estática, para as variáveis globais à aplicação (definidas localmente) e para as variáveis localmente globais (internas 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 **arquitectura segmentada**.

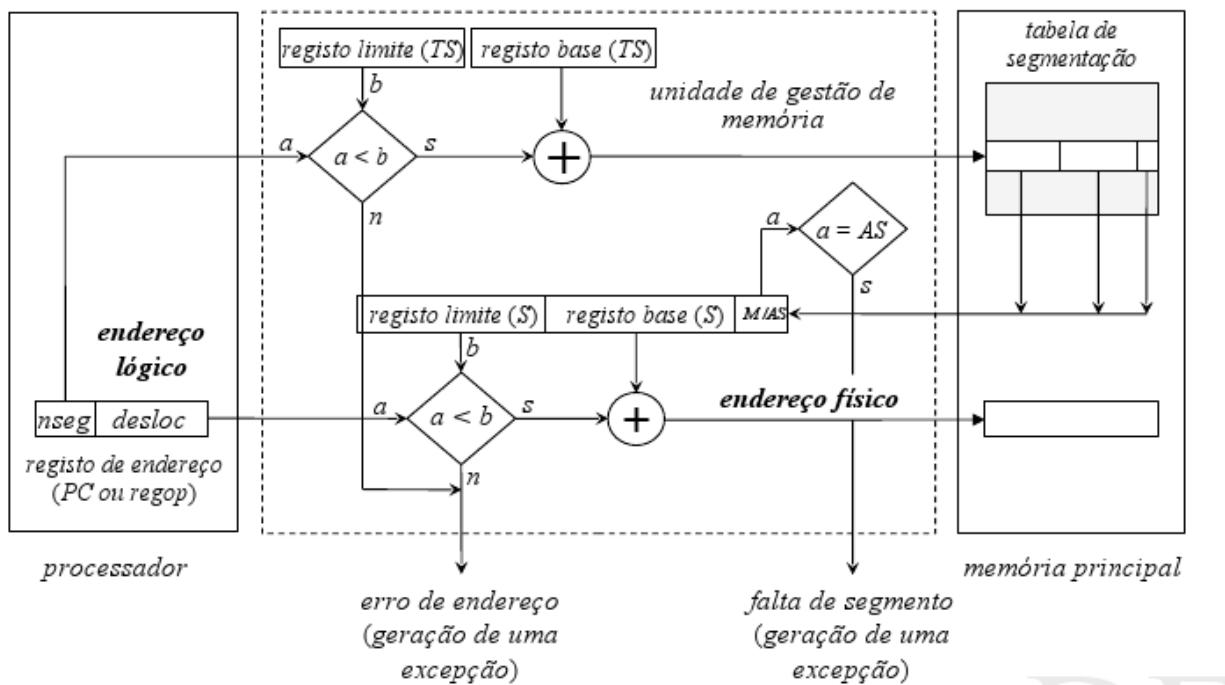
Como é evidente, os blocos (ou segmentos), resultantes da divisão do espaço de endereçamento lógico do processo, são de comprimento variável.

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.



Tradução de um endereço lógico num endereço físico



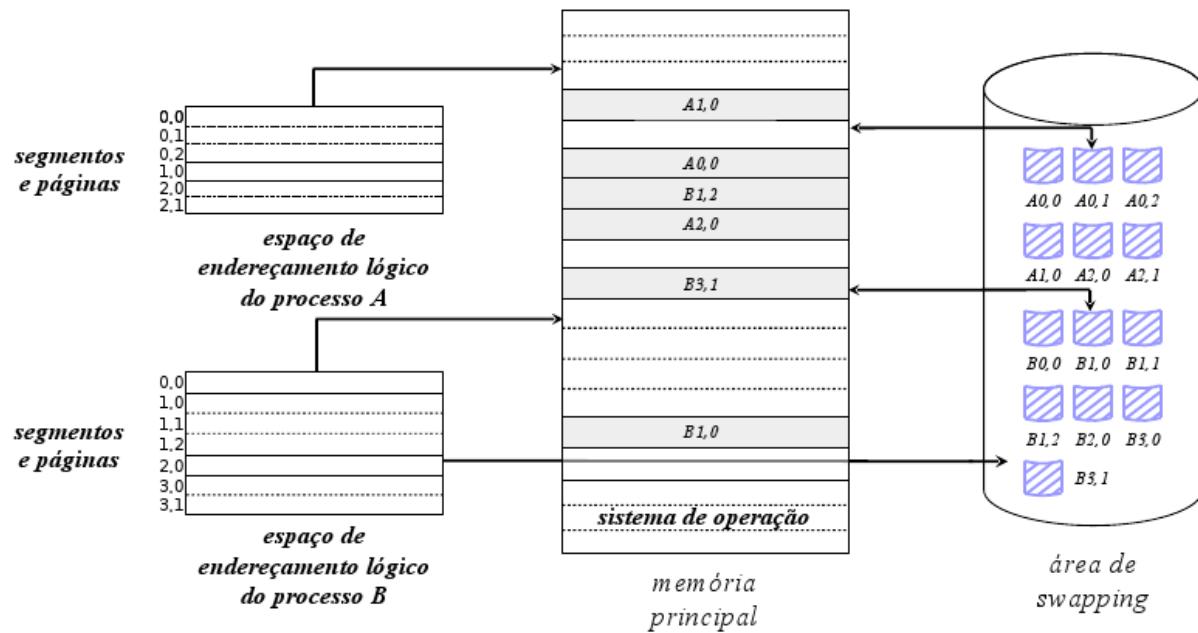
- a **arquitectura segmentada** na sua versão pura tem pouco interesse prático porque, tratando a memória principal como um espaço contínuo, exige a aplicação de técnicas de reserva de espaço para carregamento de um segmento em memória que são em tudo semelhantes às usadas em organizações de memória 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*.

Arquitectura segmentada / paginada

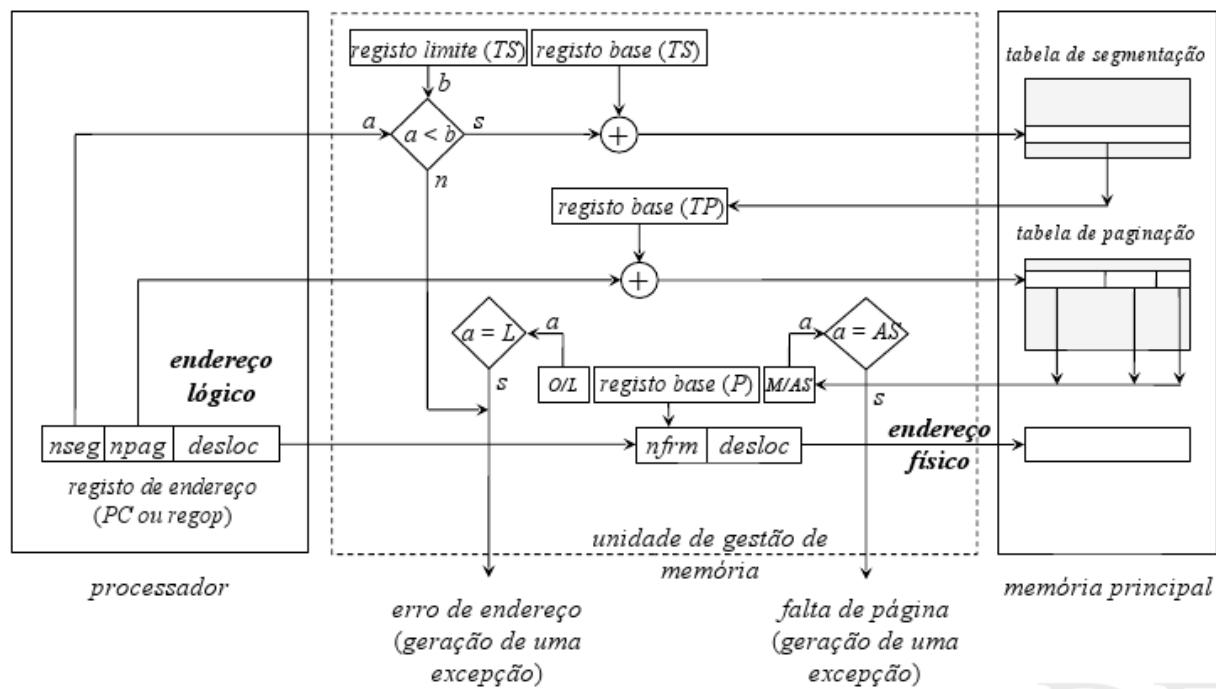
Uma alternativa potencialmente interessante é uma arquitectura mista, designada de arquitectura segmentada e paginada, que é constituída pela combinação das características desejáveis das duas arquitecturas anteriores.

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;
- mas agora 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 arquitectura paginada.



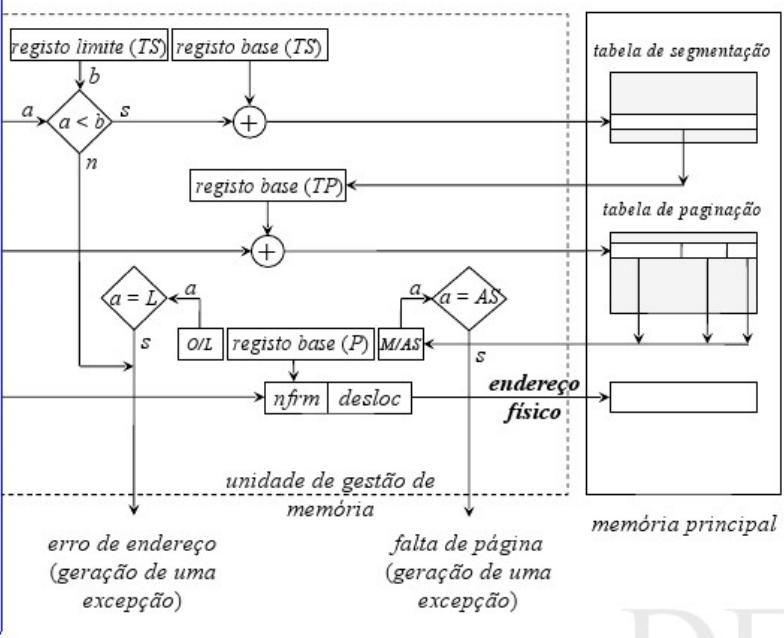
Tradução de um endereço lógico num endereço físico



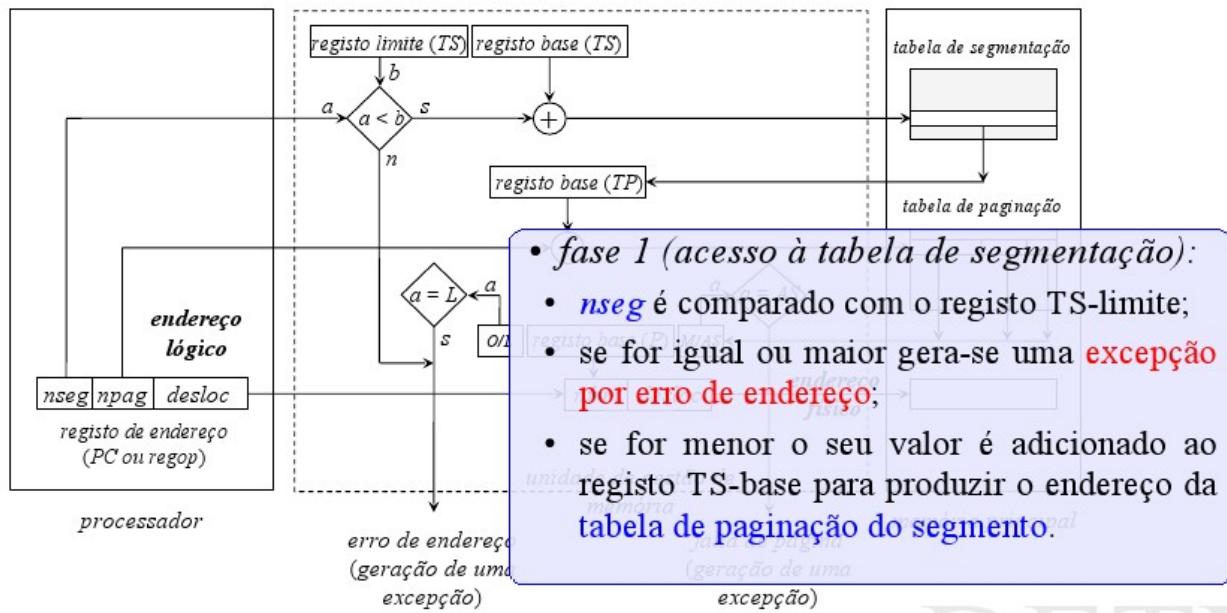
Tradução de um endereço lógico num endereço físico

a unidade de gestão de memória contém aqui **três registos base e um registo limite** associados

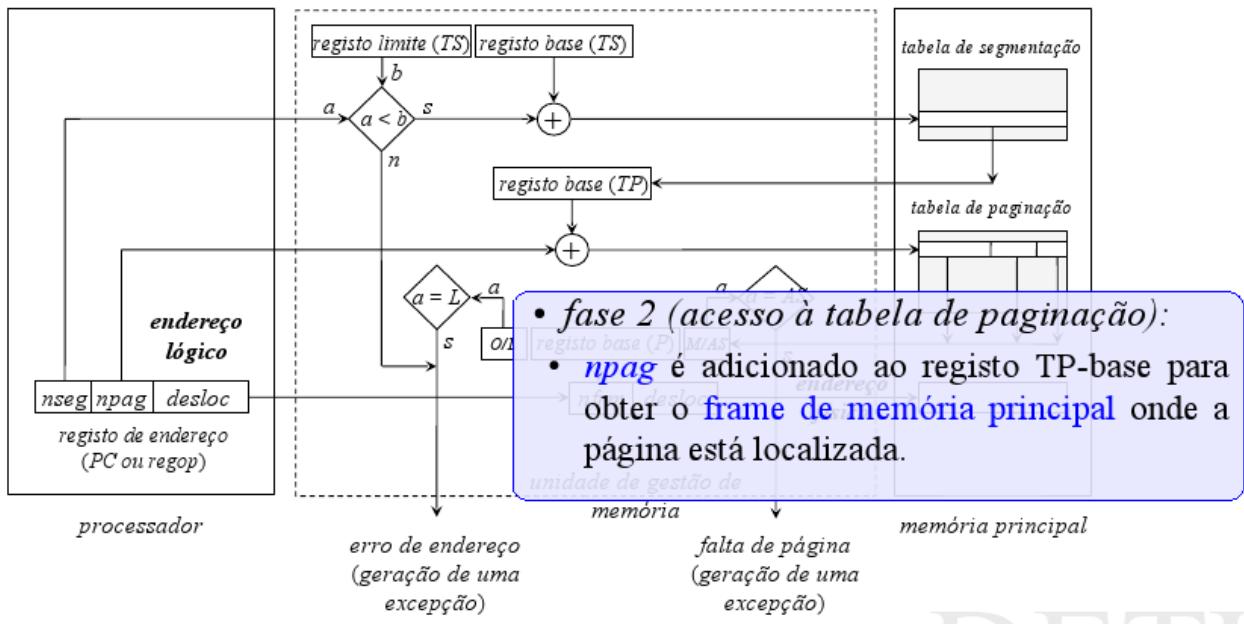
1. base da *tabela de segmentação do processo*
2. base da *tabela de paginação do segmento* que está a ser referenciado
3. base do *frame* de memória principal onde a página está localizada
4. número de entradas da *tabela de segmentação* (o registo limite);



Tradução de um endereço lógico num endereço físico

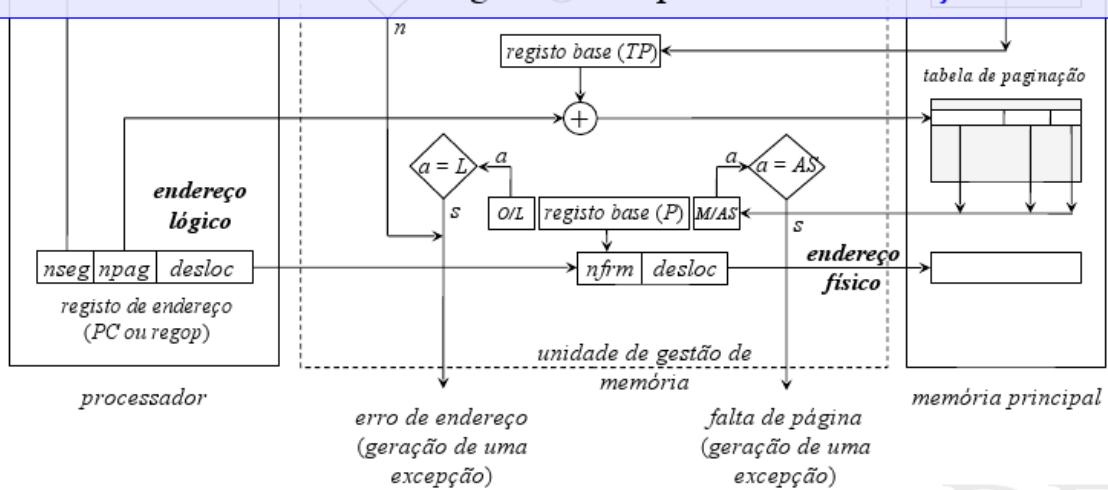


Tradução de um endereço lógico num endereço físico



• fase 3 (geração do endereço físico):

- excepção por erro de endereço** se a página não existe
- excepção por falta de bloco** se página apenas está na área de swap
- desloc** é concatenado com o registo P-base para obter o **endereço físico**.



Conteúdo de cada entrada da tabela de segmentação

<i>Prot</i>	<i>Endereço em memória da tabela de paginação do segmento</i>
-------------	---

Conteúdo de cada entrada da tabela de paginação de cada segmento

<i>O/L</i>	<i>M/AS</i>	<i>Ref</i>	<i>Mod</i>	<i>N. do frame em memória</i>	<i>N. do bloco na área de swapping</i>
------------	-------------	------------	------------	-------------------------------	--

o facto mais saliente é o deslocamento do campo Prot para a descrição de cada segmento – o tipo de acesso é, assim, tratado de uma maneira global.

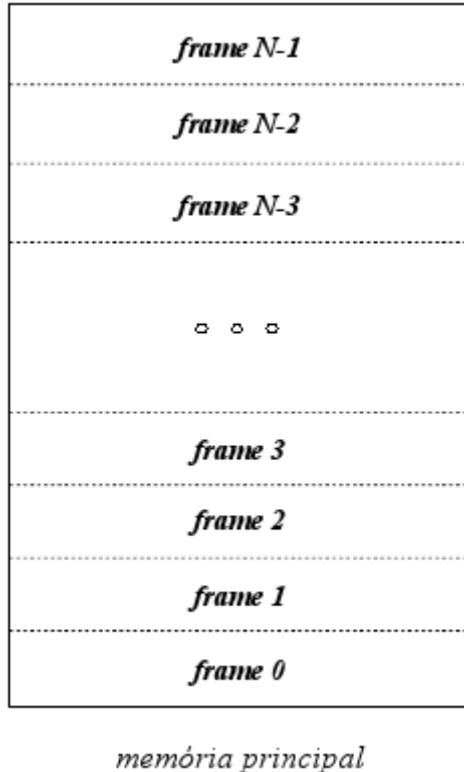
Vantagens

- **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.

Desvantagens

- **exige requisitos especiais de hardware** – ao contrário do Pentium da Intel, nem todos os processadores actuais 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; contudo, este aspecto pode ser minimizado se a unidade de gestão de memória contiver uma memória associativa, o chamado *translation lookaside buffer* (TLB), para armazenamento das entradas das tabelas de paginação recentemente mais referenciadas;
- **operacionalidade muito exigente** – a sua implementação por parte do sistema de operação é ainda mais exigente do que a da arquitectura paginada.

Política de substituição de páginas em memória



Numa arquitectura paginada ou segmentada e paginada, a memória principal é vista como estando dividida operacionalmente em *frames* do tamanho de cada página.

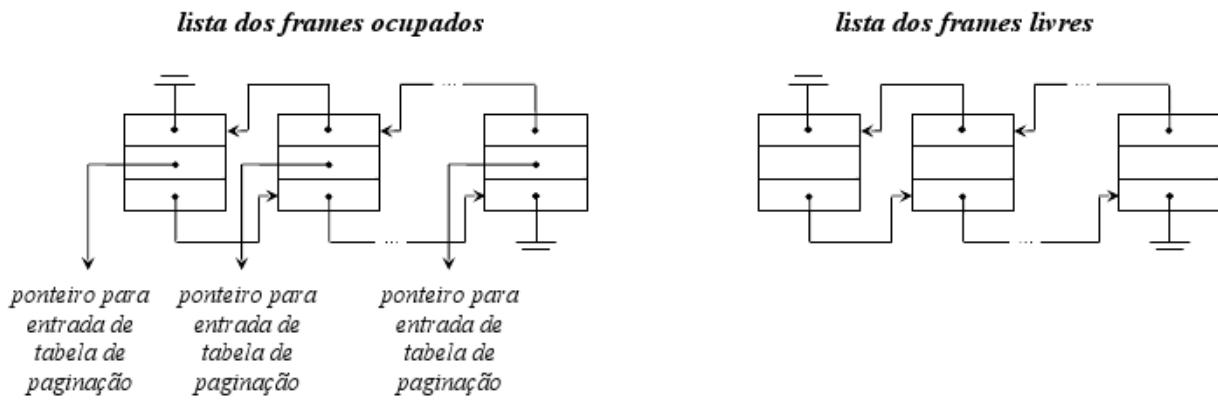
Cada *frame* vai, em princípio, permitir o armazenamento do conteúdo de uma página do espaço de endereçamento lógico de um processo.

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.

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.



Quando ocorre uma falta de página, a situação mais comum é a lista dos *frames* livres estar vazia e, por isso, torna-se necessário seleccionar um *frame* para substituição da lista dos *frames* ocupados.

Alternativamente, pode manter-se sempre na lista dos *frames* livres alguns *frames*, um deles será 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, deveria ser um *frame* que não fosse mais ser referenciado ou, sendo-o, que o fosse mais tarde possível – **princípio da optimalidade**. Minimiza-se deste modo a ocorrência de outras faltas de página.

O **princípio da optimalidade** é, porém, um princípio **não causal** e não pode ser directamente implementado. Procura-se, assim, encontrar estratégias de substituição que sejam realizáveis e que, ao mesmo tempo, se aproximem tanto quanto possível do princípio da optimalidade.

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.

O principal problema com esta estratégia é que apresenta um custo de implementação elevado e é pouco eficiente

- cada referência à memória tem que ser sinalizada com um valor que identifique o instante da sua ocorrência (conteúdo de um *timer* ou de um contador); o que significa que, ou a unidade de gestão de memória tem capacidade para o fazer, o que não é provável, ou hardware específico terá que ser acoplado;
- sempre que ocorra uma falta de página, toda a lista ligada dos *frames* ocupados

terá que ser percorrida para determinar aquele cujo último acesso foi realizado há mais tempo.

Uma aproximação possível ao LRU, de implementação fácil e relativamente eficiente, é a estratégia designada NRU (**Not Recently Used**). Neste caso, são apenas usados os bits **Ref** e **Mod** que são processados tipicamente por uma unidade de gestão de memória convencional (**Ref** é actualizado em cada acesso à página, **Mod** é actualizado 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 enquadraram-se numa das classes seguintes

	Ref	Mod
<i>classe 0</i>	0	0
<i>classe 1</i>	0	1
<i>classe 2</i>	1	0
<i>classe 3</i>	1	1

A selecção da página a substituir será feita entre aquelas pertencentes à classe de ordem mais baixa existente actualmente na lista dos *frames* ocupados.

Algoritmo da segunda oportunidade

Um critério baseado no tempo de estadia das páginas em memória principal pode também ser usado. O pressuposto é que quanto mais tempo as páginas residirem em memória, menos provável será que elas sejam referenciadas a seguir.

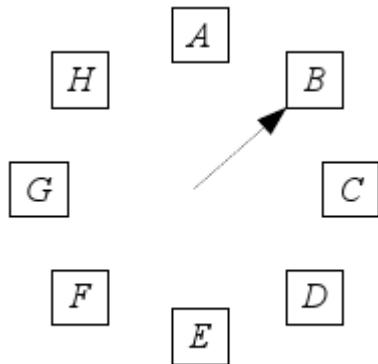
Como é evidente, este pressuposto por si só é extremamente falível (veja-se, por exemplo, as páginas associadas com o código do editor de texto ou do compilador num ambiente de desenvolvimento, ou com bibliotecas de sistema). Contudo, se se juntar ao critério um refinamento, este pressuposto pode tornar-se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

Considere-se, pois, 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 um nó descritivo de um *frame*.
 - Se o seu bit **Ref** estiver a zero, a página associada é escolhida para substituição;
 - caso contrário, coloca-se a zero o bit **Ref** e o nó é reintroduzido no fim do FIFO.

É precisamente este carácter de alternativa que dá o nome ao algoritmo.

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 *fifo_out* e *fifo_in* transformam-se, assim, num mero incremento de um ponteiro (módulo o n.º de elementos da lista).

Quando ocorre uma falta de página, o nó indicado pelo ponteiro é analisado. Se o bit **Ref** do *frame* descrito estiver a zero, a página associada é escolhida para substituição; caso contrário, coloca-se a zero o bit **Ref**. Em qualquer caso, o ponteiro é incrementado de uma unidade e a pesquisa continua até ser encontrado um *frame* adequado.

Working Set

Quando um processo é colocado pela primeira vez na fila de espera dos processos READY-TO-RUN, torna-se necessário decidir que páginas carregar em memória principal.

Uma opção razoável é seleccionar a **primeira** e a **última** páginas do seu espaço de endereçamento (correspondentes, respectivamente, ao **início do código** e ao **topo do stack**). 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 fracção do espaço de endereçamento que o processo está actualmente a referenciar, estão todas presentes em memória principal. Este conjunto de páginas constitui aquilo que se designa pelo nome de **working set** do processo.

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.

- Quando o número de *frames* em memória principal atribuído ao processo é fixo e inferior ao seu *working set* actual, 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.

Estes podem ser de aplicação

- **local** – se a escolha for efectuada entre o conjunto de *frames* atribuído ao processo;
- **global** – se a escolha for efectuada 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.