Sistemas Operativos

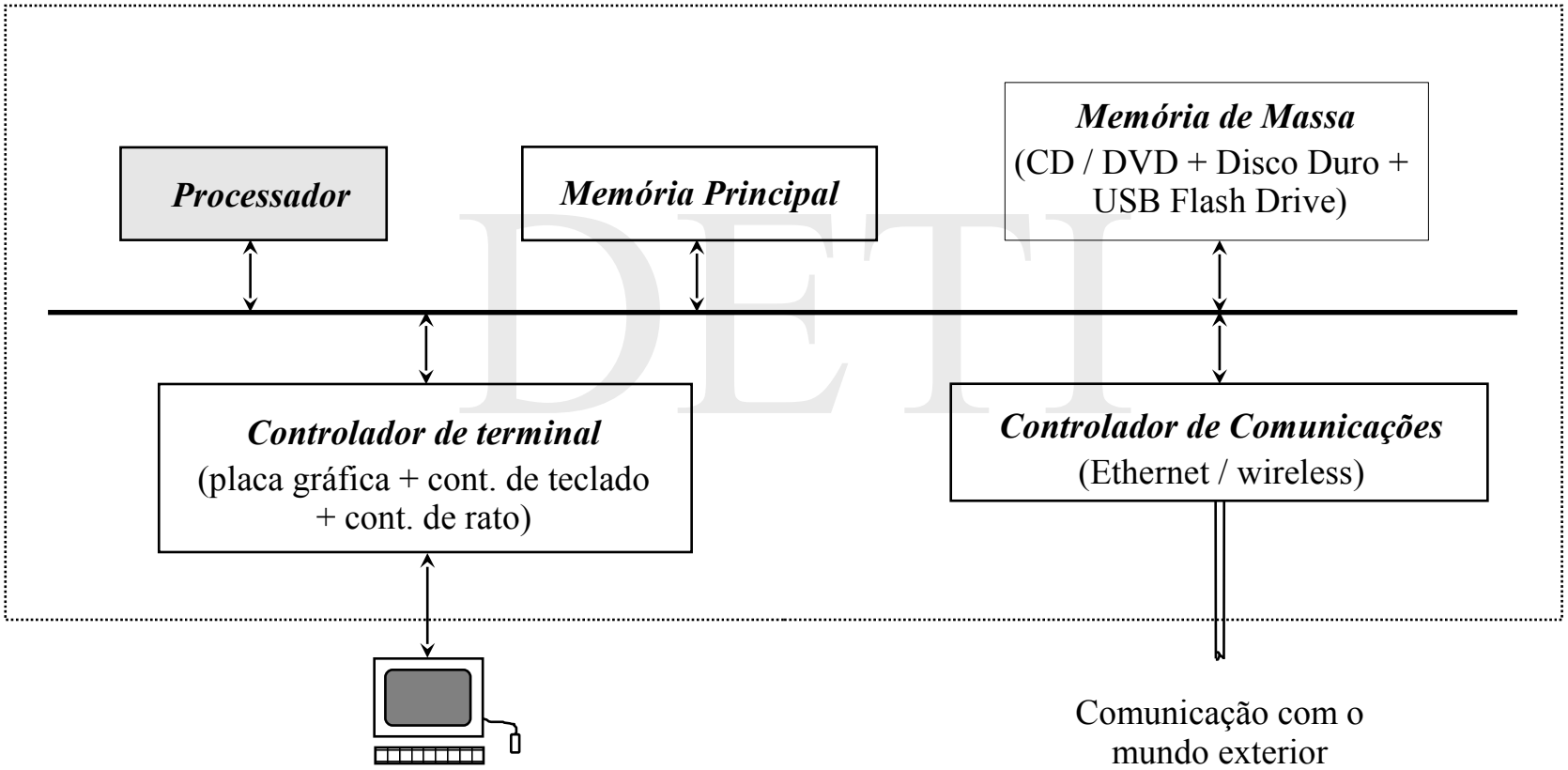
Gestão do Processador

António Rui Borges

Sumário

- *Programa vs. Processo*
- *Caracterização de um ambiente multiprogramado*
- *Gestão de processos em Unix* (*interface de programação*)
- *Implementação de um ambiente multiprogramado*
- *Scheduling do processador*
- *Processos vs. Threads*
- *Caracterização de um ambiente multithreaded*
- *Leituras sugeridas*

Sistema computacional



Programa vs. Processo

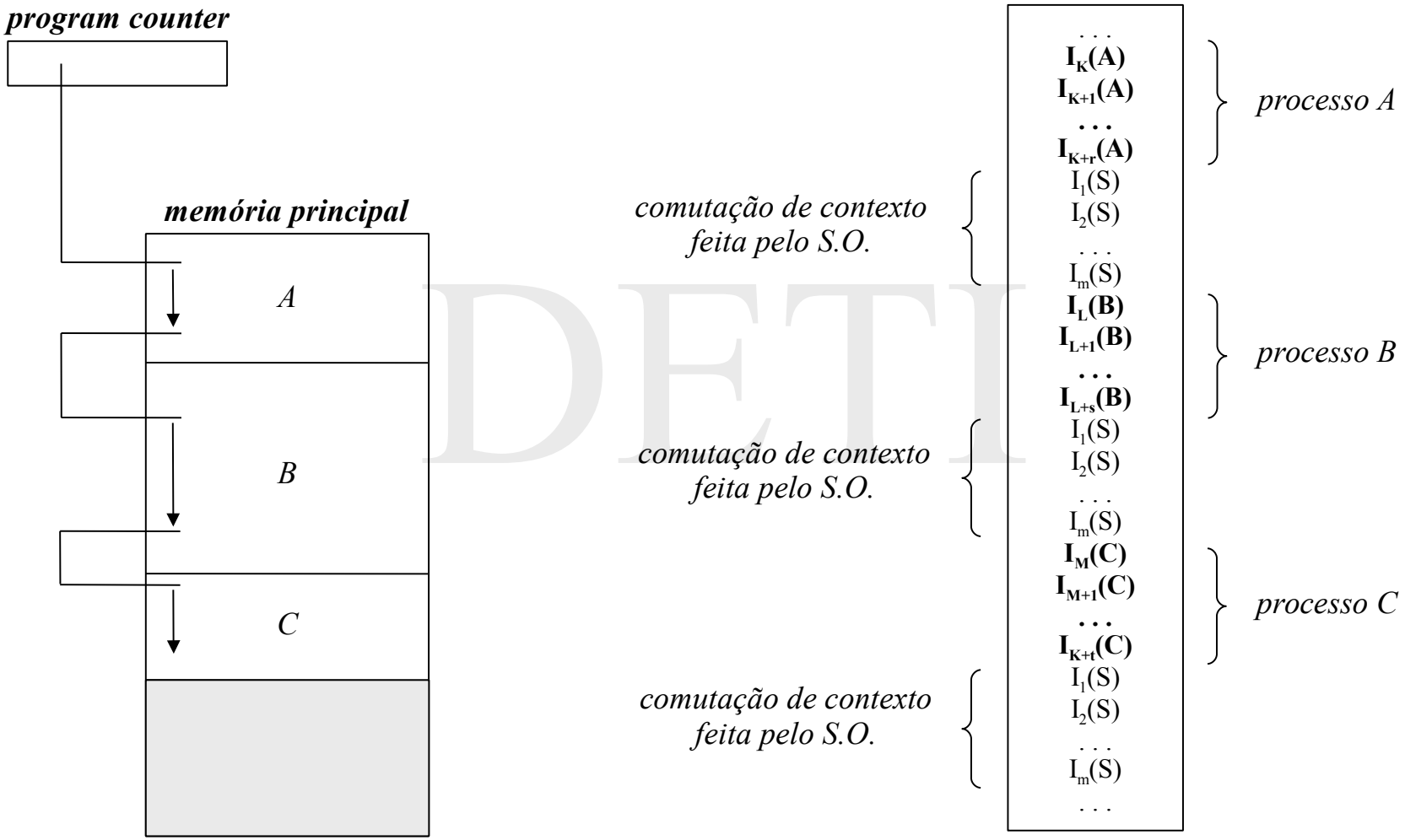
De um modo geral, um *programa* pode ser definido como um conjunto de instruções que descreve a realização de uma determinada tarefa por um computador. Contudo, para que essa tarefa seja *de facto* realizada, o programa correspondente tem que ser executado.

A execução de um programa designa-se de *processo*.

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

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

Tracing de execução num ambiente multiprogramado



Modelação dos processos - 1

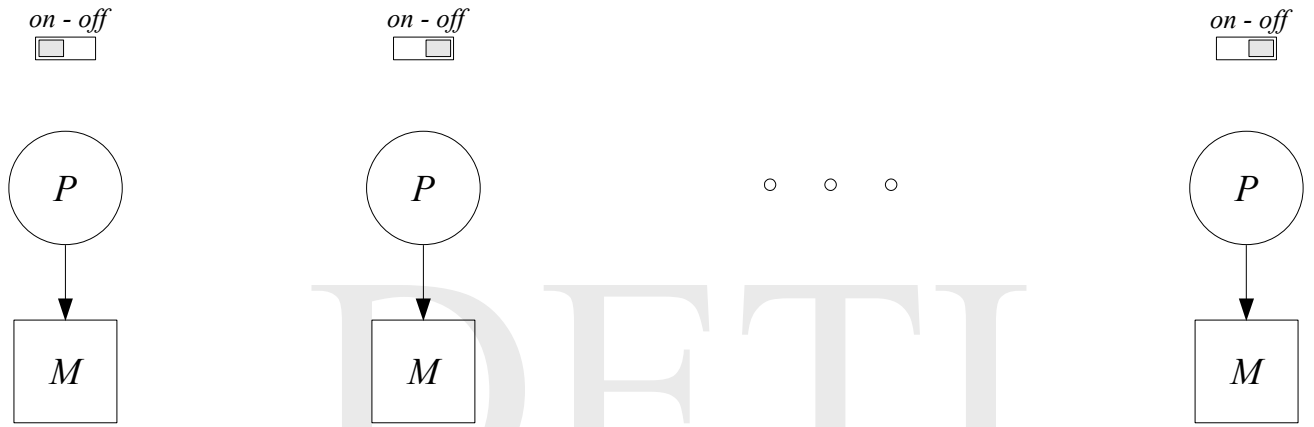
A *multiprogramação*, ao criar uma imagem de aparente simultaneidade na execução de diferentes programas pelo mesmo processador, torna muito complexa a percepção das diferentes actividades que estão em curso.

Esta imagem pode, porém, ser simplificada se, em vez de procurar seguir-se o percurso do processador nas suas constantes comutações entre processos, se supuser a existência de um conjunto de processadores virtuais, um por cada processo que concorrentemente coexiste, e se admitir que os processos associados são executados em paralelo através da activação (*on*) e desactivação (*off*) dos processadores respectivos.

Para que tal modelo seja viável, é preciso garantir que

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

Modelação dos processos - 2



- a *comutação de contexto* é simulada pela activação e desactivação dos processadores virtuais e é controlada pelo seu *estado*;
- num *monoprocessador*, o número de processadores virtuais activos em cada instante é no máximo de um;
- num *multiprocessador*, o número de processadores virtuais activos em cada instante é no máximo igual ao números de processadores existentes.

Diagrama de estados de um processo - 1

Um processo vai encontrar-se em diferentes situações, designadas de *estados*, ao longo da sua existência.

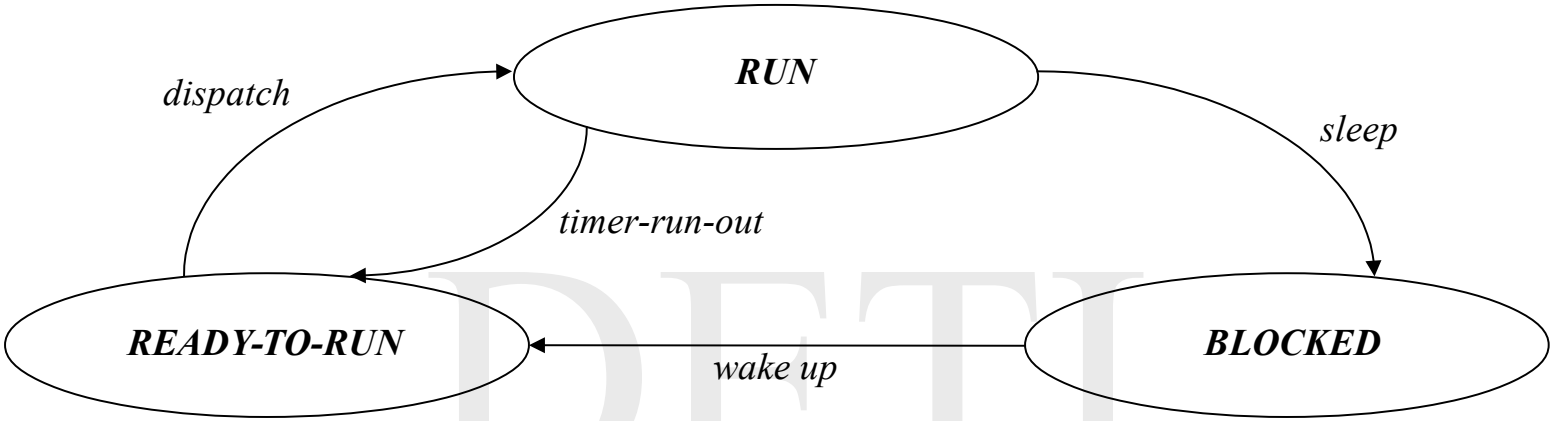
Os estados mais importantes são os seguintes

- *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.).

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*] e constitui parte integrante do seu núcleo central (o *kernel*) que é responsável pelo tratamento das interrupções e por agendar a atribuição do processador e de muitos outros recursos do sistema computacional.

Diagrama de estados de um processo - 2



dispatch – um dos processos da *fila de espera dos processos prontos a serem executados* é seleccionado pelo *scheduler* 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.

Diagrama de estados de um processo - 3

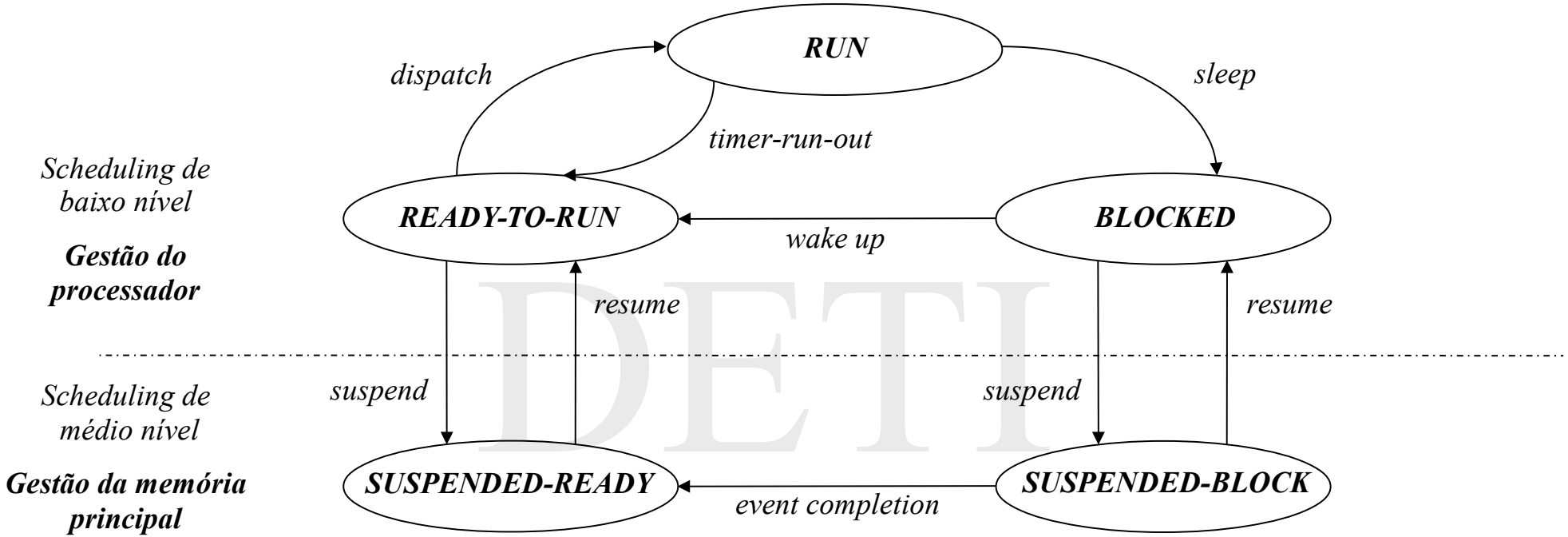
A memória principal, por muito grande que seja, é necessariamente finita. Num ambiente multiprogramado, em que o número de processos que coexistem correntemente, é muito elevado, ela acaba por se tornar um factor limitativo ao seu crescimento.

Para obviar esta situação, costuma criar-se em memória de massa uma extensão à memória principal, comummente designada de *área de swapping*, que funciona como uma região secundária de armazenamento. Liberta-se, assim, espaço em memória principal para a execução de processos que, de outro modo, não poderiam existir e potencia-se, portanto, um 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-block* – correspondente ao estado *blocked*.

Diagrama de estados de um processo - 4



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.

Diagrama de estados de um processo - 5

A situação descrita até agora pressupõe que os processos são eternos, existindo permanentemente enquanto o sistema computacional estiver operacional. À parte de alguns processos de sistema, porém, este não é o caso mais frequente.

Os processos são em geral criados, têm *um tempo de vida* mais ou menos longo e terminam. Torna-se assim necessário controlar o grau de multiprogramação, que deve ser o maior possível, mas garantindo sempre um equilíbrio adequado entre o serviço aos processos que coexistem e a ocupação do processador.

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 manutenção temporária, após a terminação do processo, da informação relativa à história da sua execução; 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.

Diagrama de estados de um processo - 6

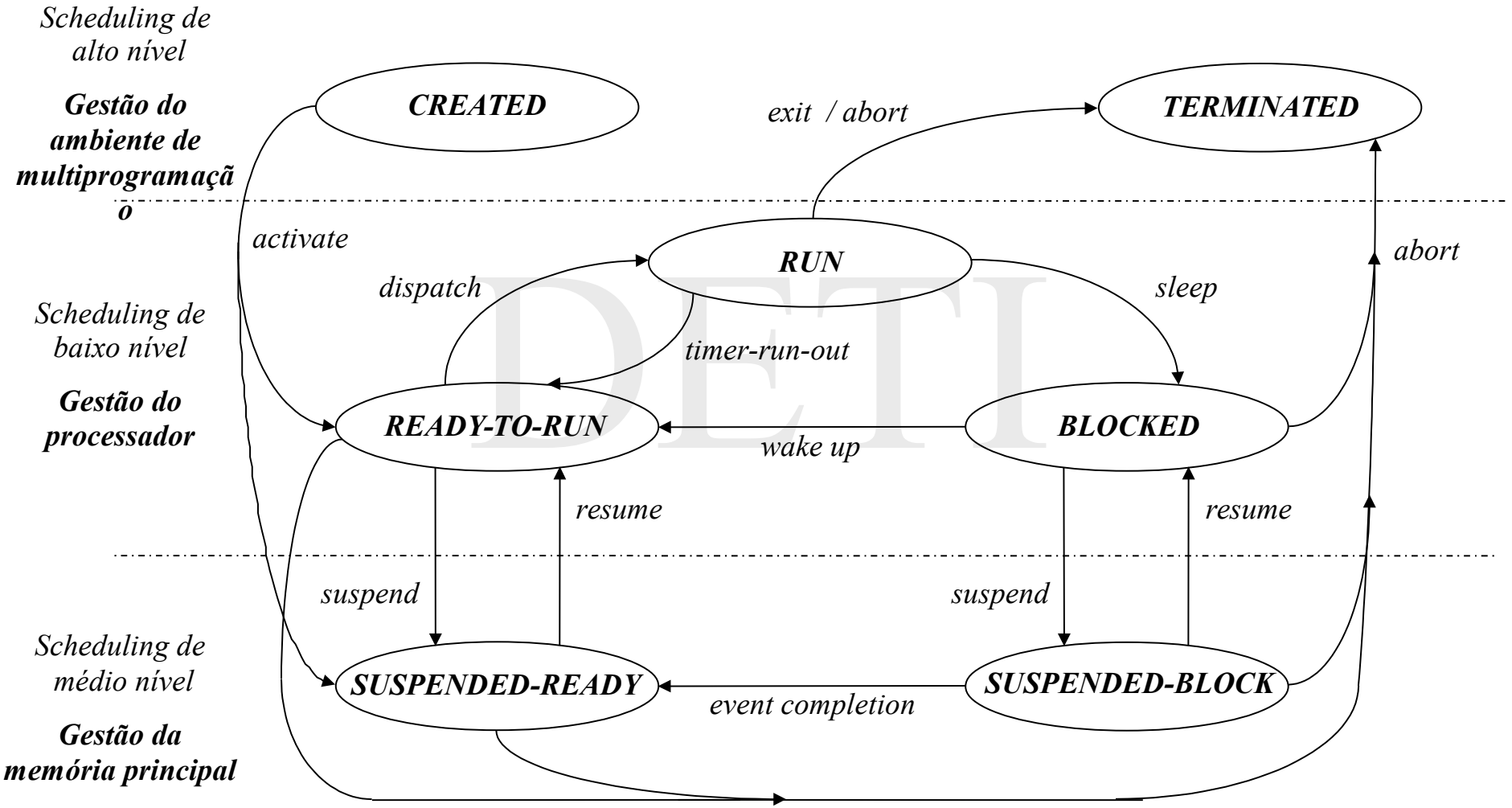


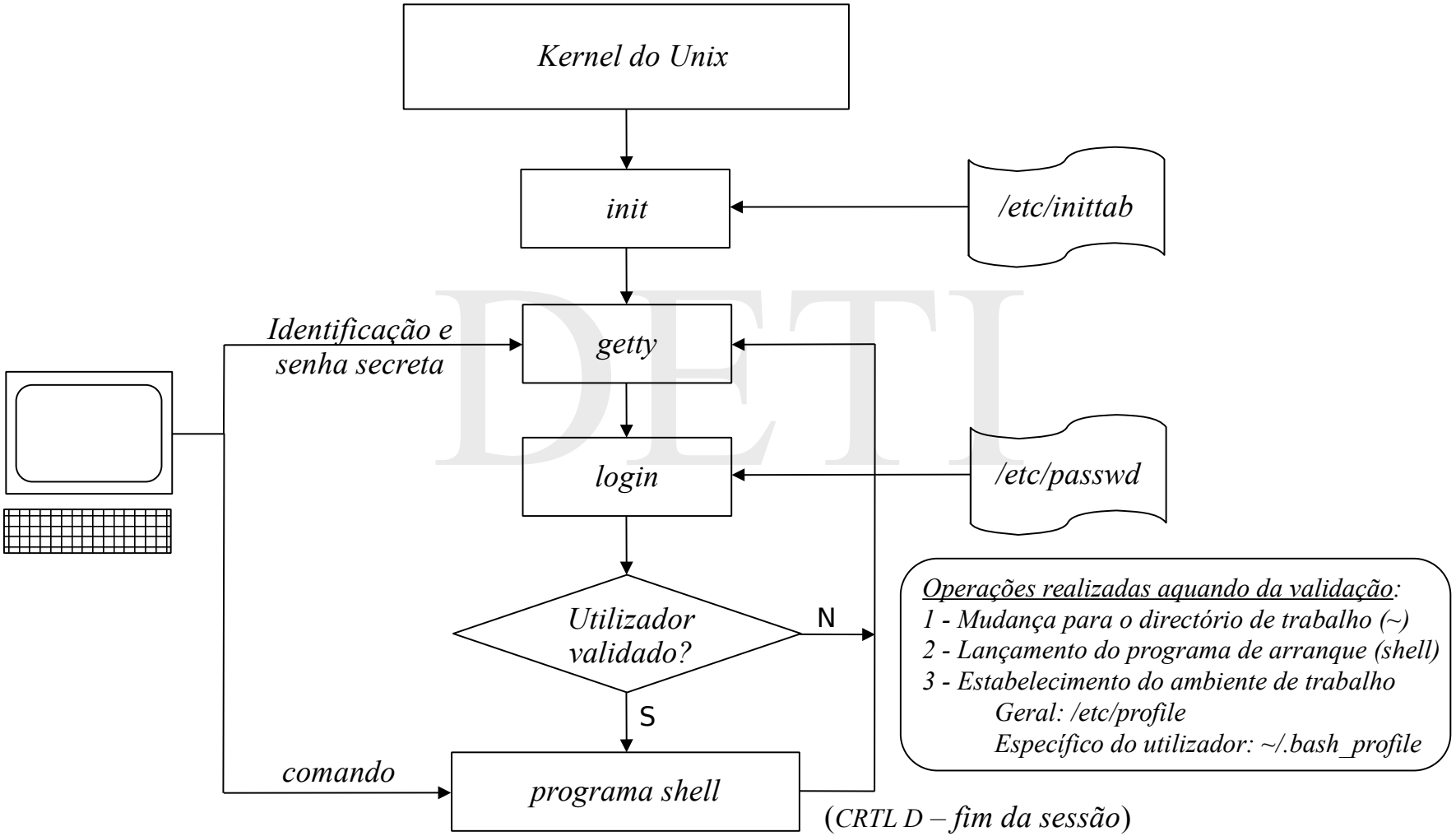
Diagrama de estados de um processo - 7

activate – lançamento do processo em execução; o processo pode ser colocado na *fila de espera dos processos prontos a serem executados*, se houver espaço em memória principal para carregar o seu espaço de endereçamento, ou *suspenso*, em caso contrário;

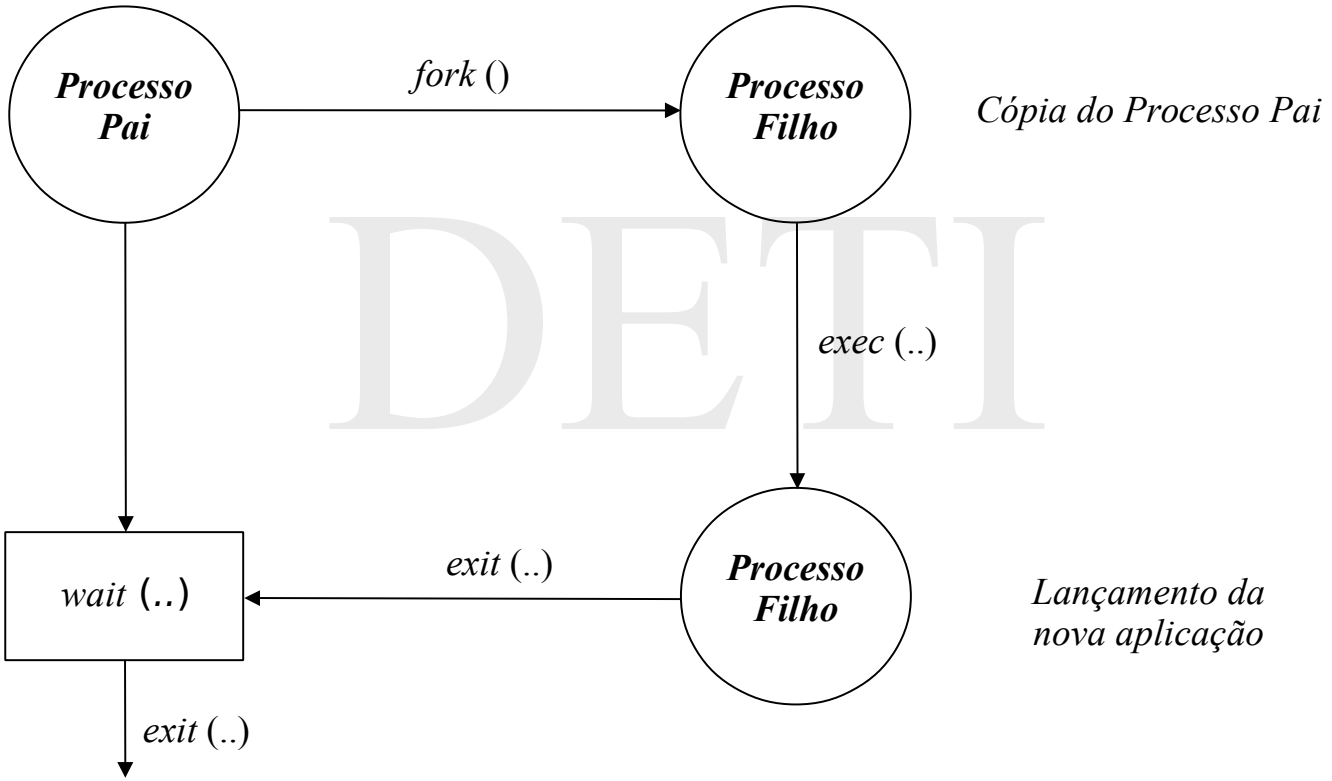
exit – terminação normal do processo;

abort – terminação anormal do processo, provocada pela ocorrência de um erro fatal ou por acção de um processo com autoridade necessária para isso.

Unix – Lançamento da shell



Criação de um processo em Unix - 1



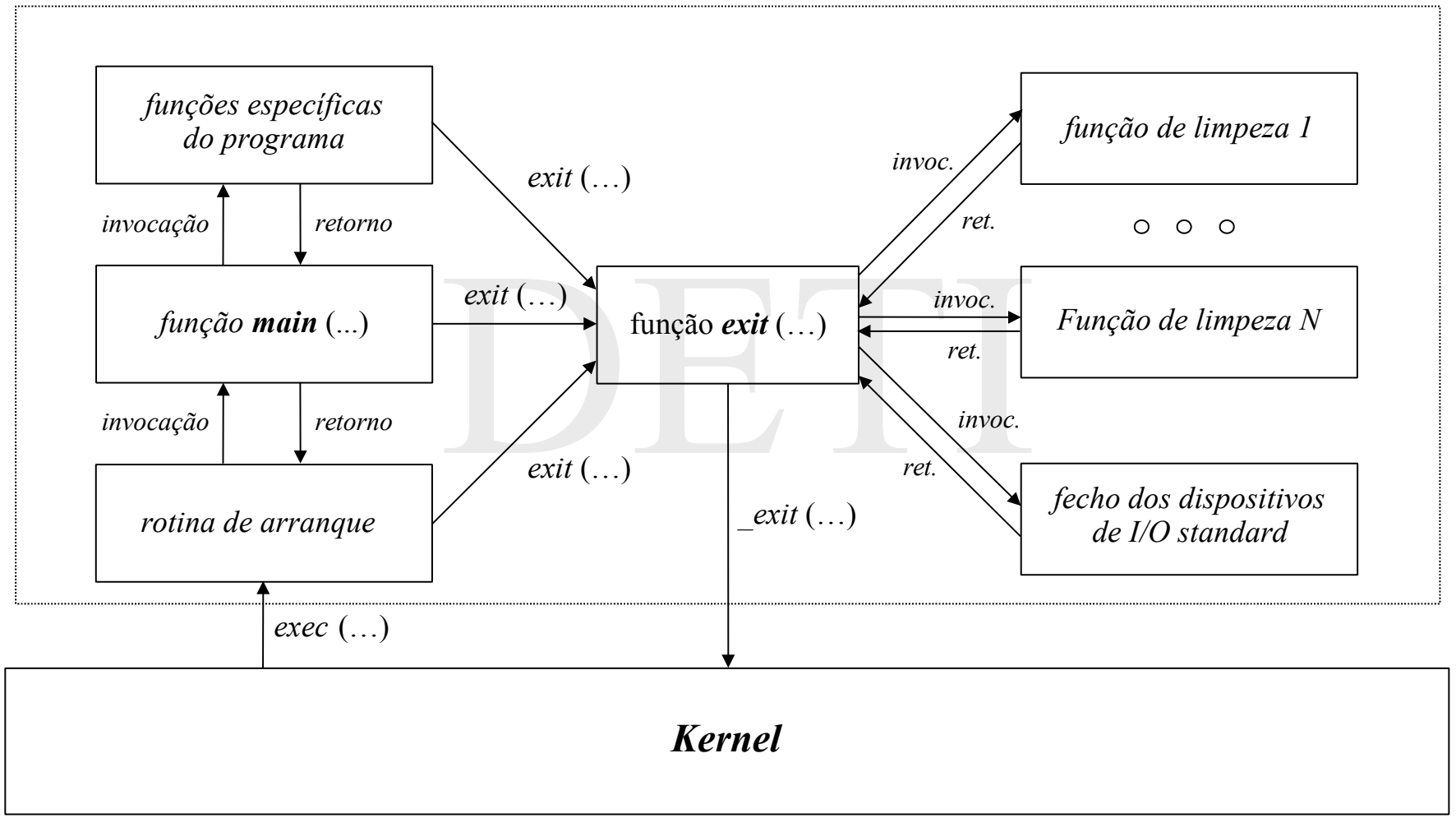
Criação de um processo em Unix - 2

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

void main (int argc, char *argv[])
{
  pid_t pid;                    /* identificador do processo filho */
  char *aplic;                  /* nome da aplicação */
  int status;                   /* 'status' de execução do processo filho */

  /* obtenção do nome da aplicação */

  if (argc != 2)
     { fprintf (stderr, "spawn <nome do ficheiro executável>\n");
       exit (EXIT_FAILURE);
     }
  aplic = argv[1];

  /* duplicação do processo */

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

Criação de um processo em Unix - 3

```
  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, NULL) < 0)
             { perror ("erro no lancamento da aplicacao");
               exit (EXIT_FAILURE);
             }
  exit (EXIT_SUCCESS);
}
```

Ambiente de execução de um programa em C



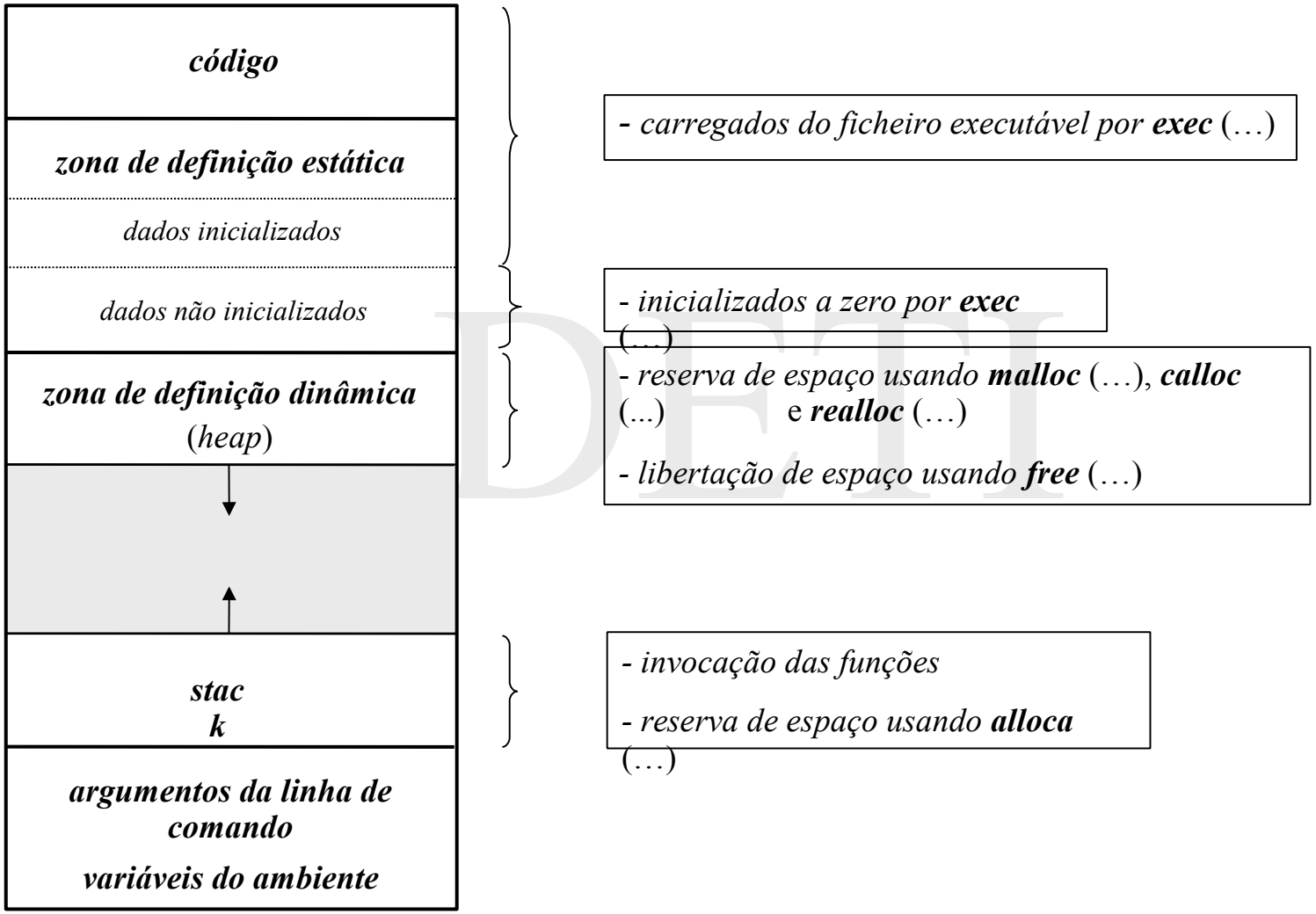
Utilização das funções de limpeza - 1

```
#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);
     }
```

Utilização das funções de limpeza - 2

```
  /* 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

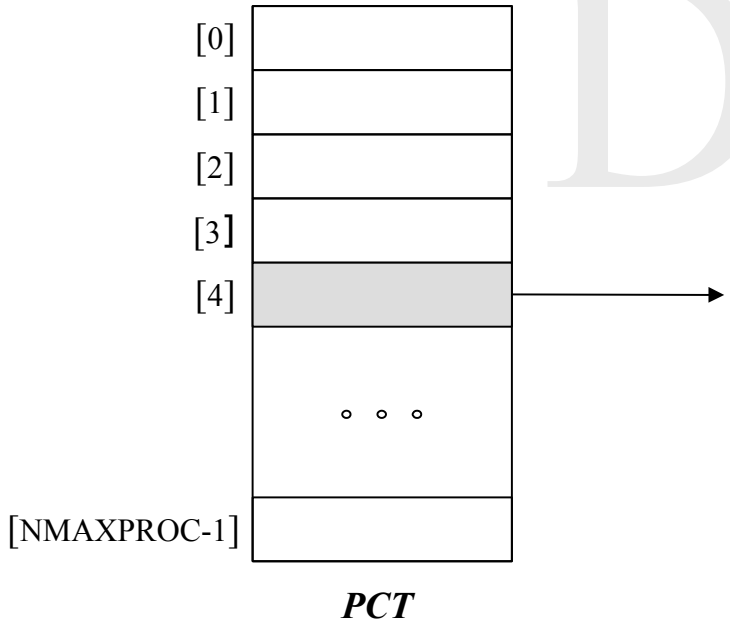


Acesso aos argumentos da linha de comando e às variáveis de ambiente

```
#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. Esta informação é mantida numa tabela, designada pelo nome de *Tabela de Controlo de Processos* (*PCT*), que é usada intensivamente pelo *scheduler* para fazer a gestão do processador e de outros recursos do sistema computacional.



identificadores – do processo, do pai do processo e do utilizador a que o processo pertence;

caracterização do espaço de endereçamento – sua localização (em memória principal ou na *área de swapping*), de acordo com o tipo de organização de memória estabelecido;

contexto do processador – valores de todos os registos internos do processador no momento em que se deu a comutação do processo;

contexto de I/0 – informação sobre os canais de comunicação e *buffers* associados;

informação de estado e de 'scheduling' - estado de execução do processo (de acordo com o diagrama de estados) e outra informação associada.

Comutação de processos - 1

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*. Por razões de segurança, contudo, 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

- um dispositivo externo (*interrupção*);
- a execução de uma instrução ilegal, ou que conduz a um erro (*divisão por zero*, por ex.);
- a execução de uma instrução de tipo *trap* (*interrupção por software*).

Comutação de processos - 2

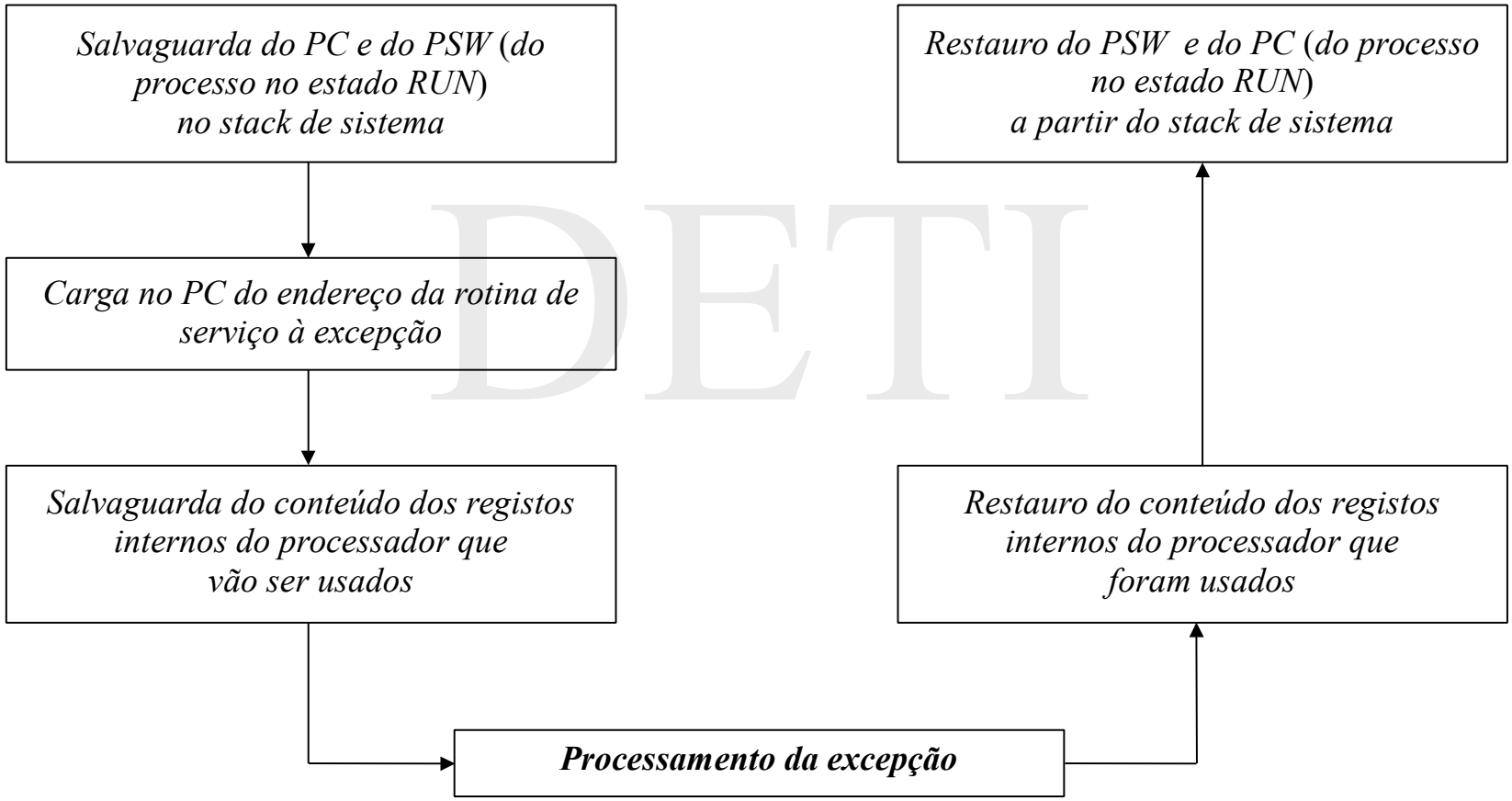
Assim, para que o sistema de operação funcione no modo privilegiado, com total acesso a toda a funcionalidade do processador, as *chamadas ao sistema* associadas, quando não despoletadas pelo próprio *hardware*, são implementadas a partir de instruções *trap*.

Cria-se, portanto, um ambiente operacional uniforme, em que todo o processamento pode ser encarado como o *serviço de excepções*.

Nesta perspectiva, a comutação de processos pode ser visualizada globalmente como uma vulgar rotina de serviço à excepção, apresentando, porém, uma característica peculiar que a distingue de todas as outras: *normalmente, a instrução que vai ser executada, após o serviço da excepção, é diferente daquela cujo endereço foi salvaguardado ao dar-se início ao processamento da excepção*.

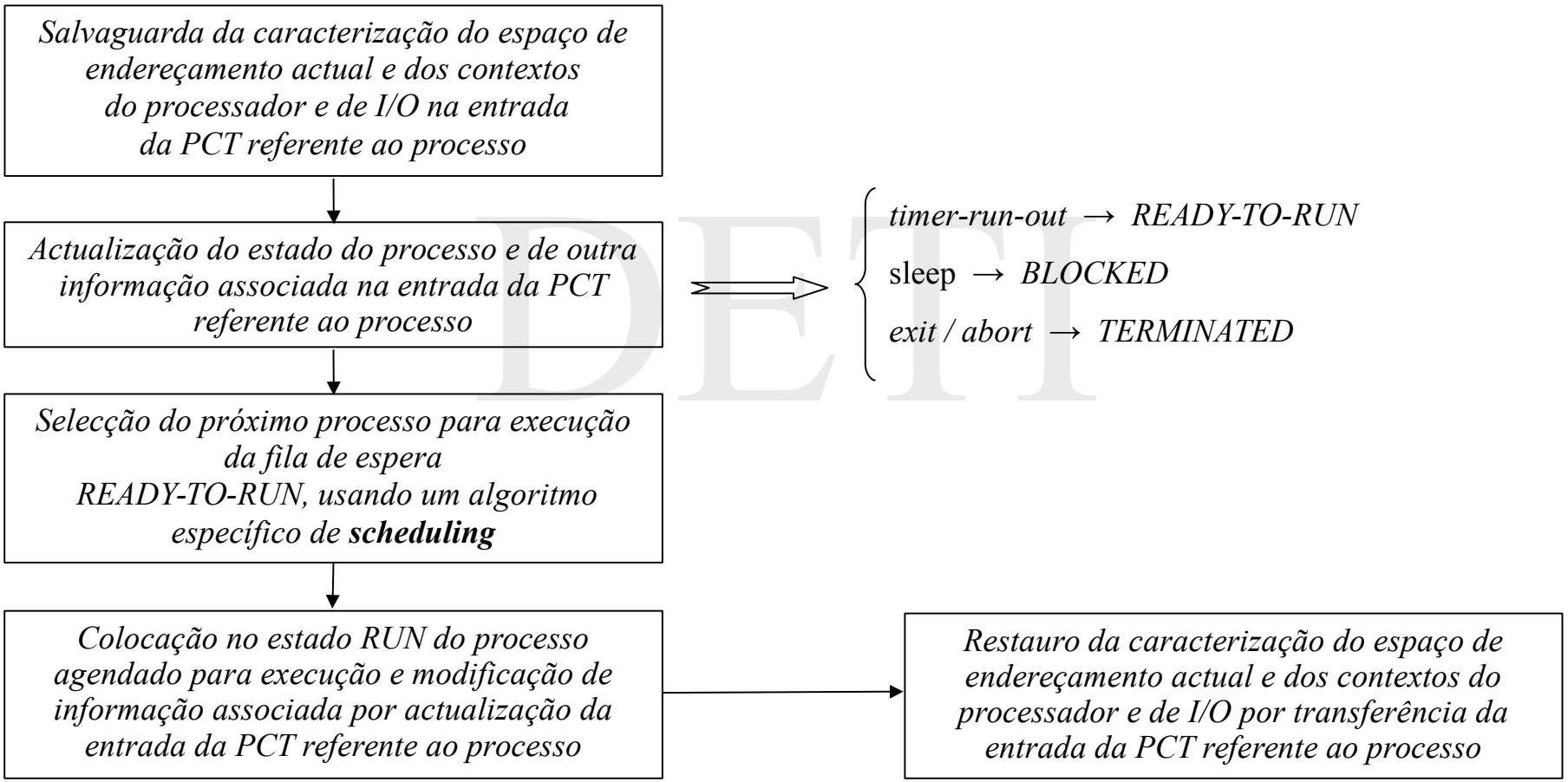
Comutação de processos - 3

Processamento de uma excepção genérica

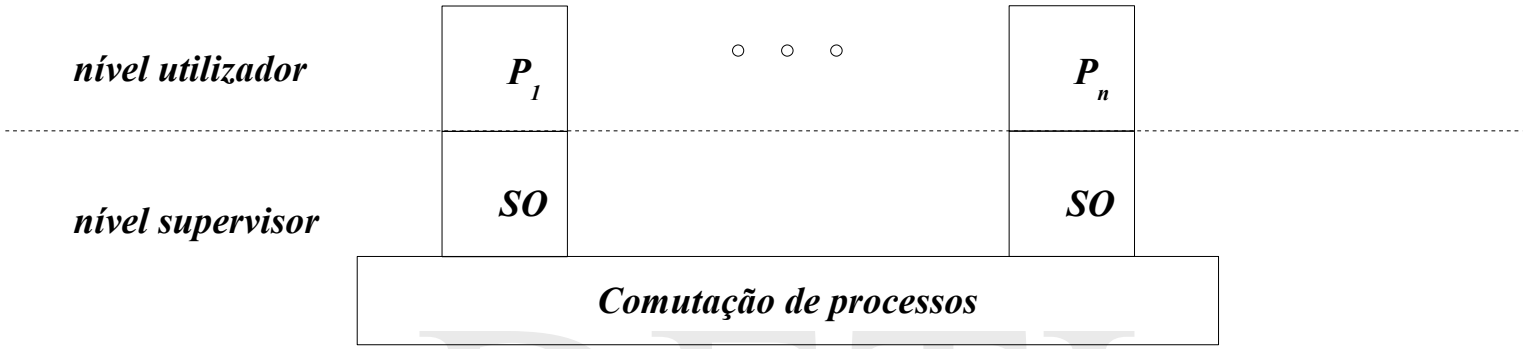


Comutação de processos - 4

Comutação propriamente dita



Comutação de processos - 5



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.

Diagrama de estados de um processo em Unix - 1

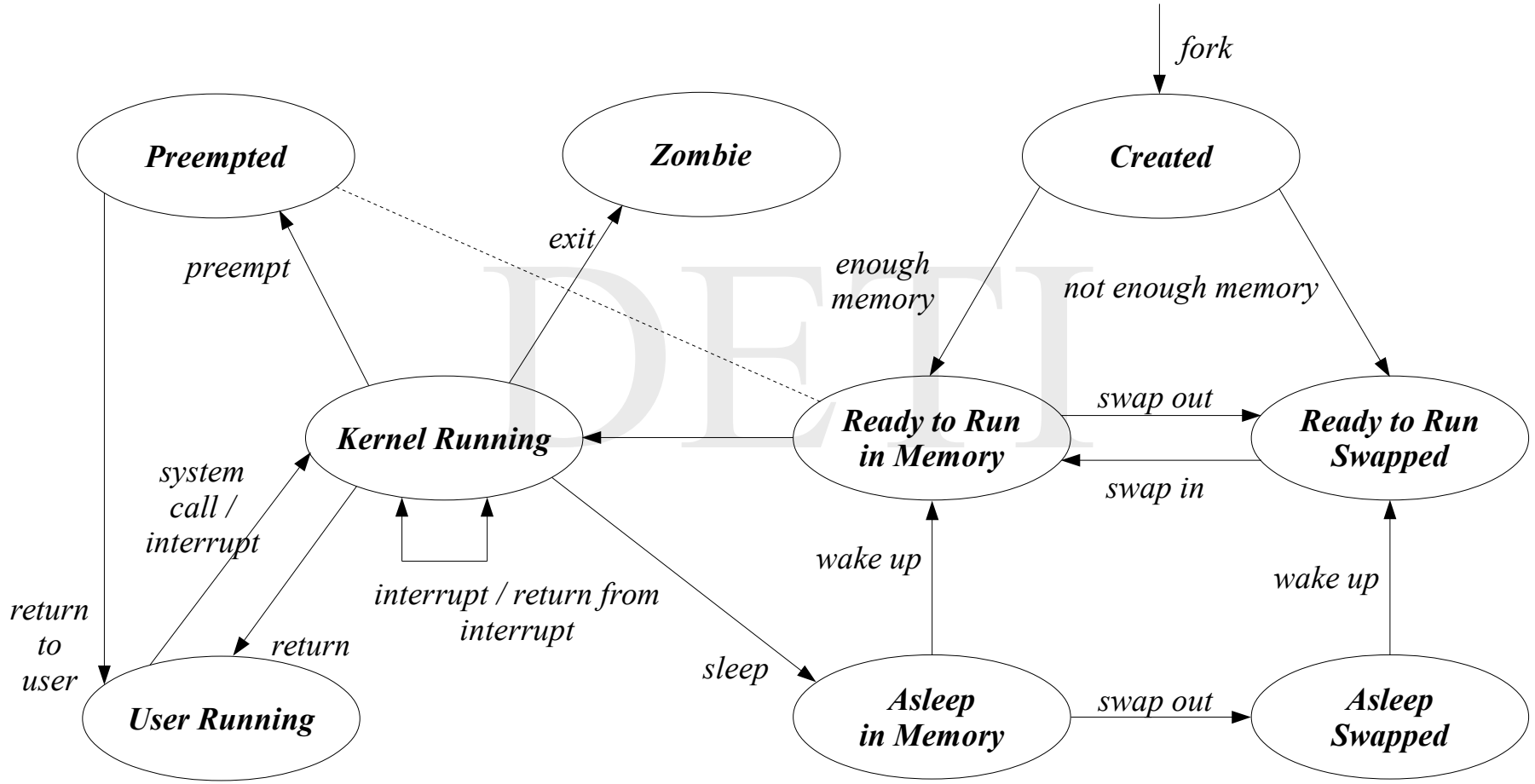


Diagrama de estados de um processo em Unix - 2

- 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*;

Diagrama de estados de um processo em Unix - 3

- 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*.

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 do processador - 1

- *justiça* – todo o processo, ao longo de um intervalo de tempo considerado de referência, deve ter direito à sua fracção 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;
- *throughput* – deve procurar maximizar-se o número de processos terminados por unidade de tempo;
- *tempo de resposta* – deve procurar minimizar-se o tempo de resposta às solicitações feitas pelos processos interactivos;
- *tempo de turnaround* – deve procurar minimizar-se o tempo de espera pelo completamento de um *job* no caso de utilizadores num sistema *batch*;
- *deadlines* – deve procurar garantir-se o máximo de cumprimento possível das metas temporais impostas pelos processos em execução;
- *eficiência* – deve procurar manter-se o processador o mais possível ocupado com a execução dos processos dos utilizadores.

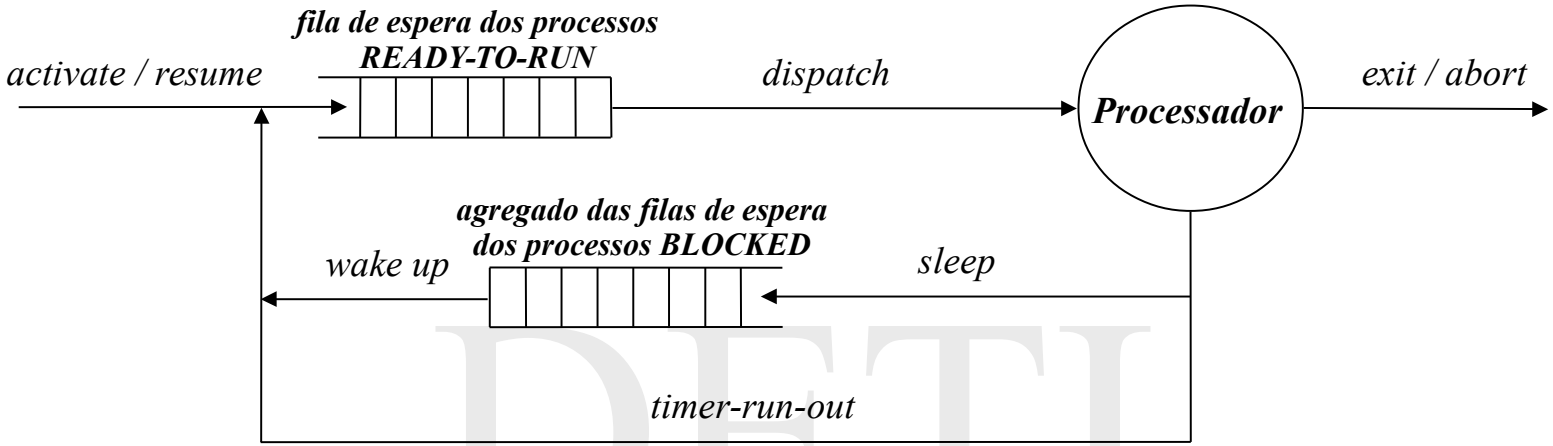
Critérios a satisfazer pelos algoritmos de scheduling do processador - 2

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 passíveis, portanto, 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

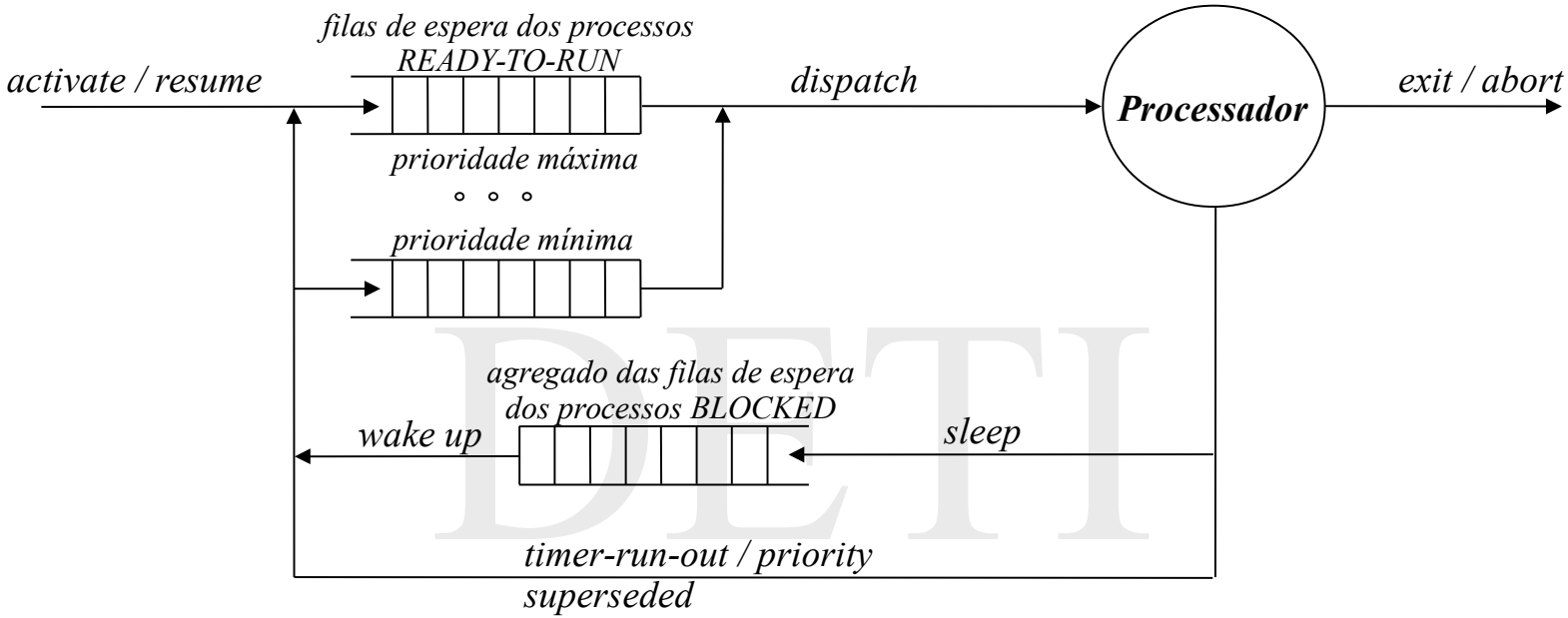
Em muitas circunstâncias concretas, considerar todos os processos em pé de igual-dade não é o procedimento mais adequado. A minimização do tempo de resposta exige, por exemplo, que seja dada preferência a processos *I/O*-intensivos sobre pro-cessos *CPU*-intensivos na calendarização para execução. Em sistemas de tempo real, por outro lado, há processos associados com o tratamento de condições de alarme ou de acções meramente operacionais, cuja execução efectiva tem que ser mantida dentro de intervalos de tempo bem definidos.

Os processos podem, por isso, ser agrupados em níveis de prioridade distinta no 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.

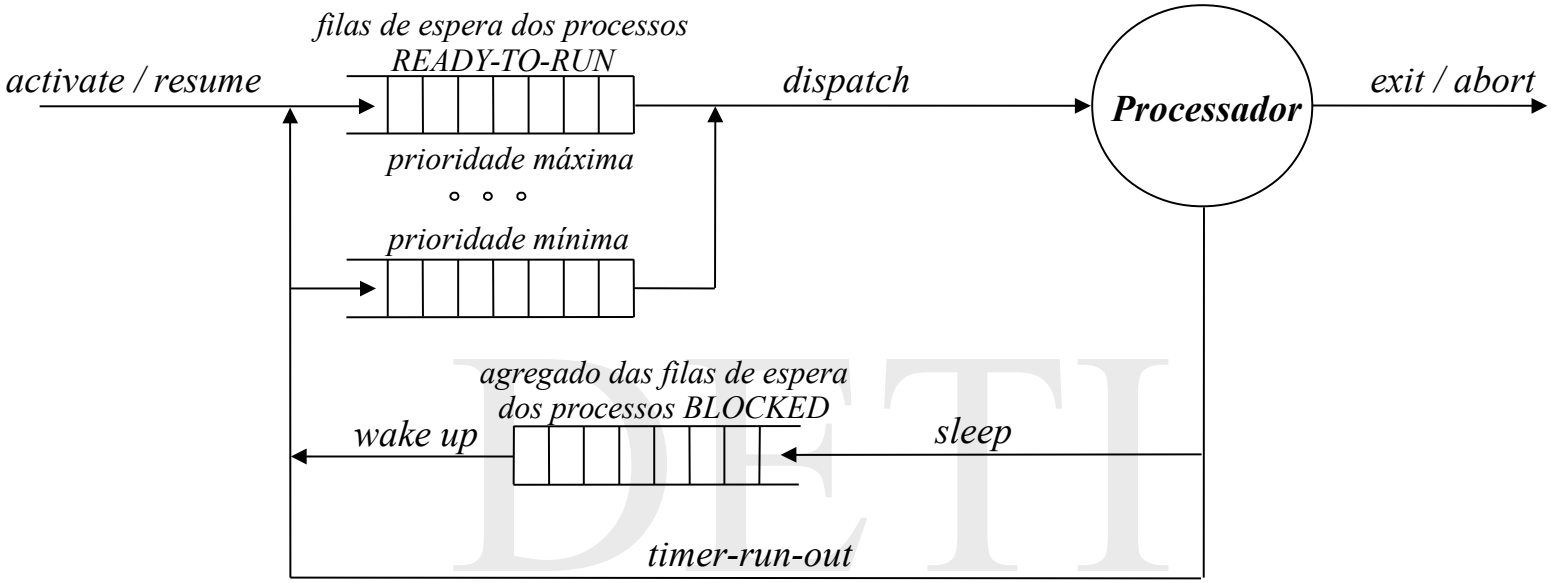
Prioridade estática - 1



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.

Prioridade estática - 2



Aquando da sua criação, é atribuído a cada processo um dado nível de prioridade. Depois, à medida que o processo é calendarizado para execução, a sua prioridade é decrementada de uma unidade, quando esgota a janela de execução atribuída, e incrementada de uma unidade, se bloqueia antes de esgotar a janela, 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 - 1

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.

Prioridade dinâmica - 2

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 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 $n = 1, 2, \ldots, N$. Então, o tempo médio de *turnaround* do grupo vem dado por

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

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*).

Prioridade dinâmica - 3

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 estima-tiva para a fracção de ocupação da próxima janela vem dada por

$$fe_2 = a \cdot fe_1 + (1-a) \cdot f_1 \ , \qquad \text{com } a \in [0, 1] \ ;$$

e, para a N-ésima janela, por

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

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

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

Prioridade dinâmica - 4

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 correcção.

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

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

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 são conhecidas por promoverem o *aging dos processos*.

Scheduling em Linux - 1

- o Linux considera três grandes classes de *scheduling*, cada uma incorporando prioridades múltiplas, que, quando ordenadas por ordem decrescente de prioridades, são
  - *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;

Scheduling em Linux - 2

Algoritmo tradicional

- para a classe *SCHED_OTHER*, o Linux usava 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

$$CPU_j(i) = \frac{CPU_j(i-1)}{2} + PBase_j + nice_j$$

em que $CPU_j(i)$ representa a prioridade do processo *j* (o número de créditos que lhe são atribuídos) no instante de recreditação *i*, $CPU_j(i\text{-}1)$ 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);

Scheduling em Linux - 3

- 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.

Scheduling em Linux - 4

Novo algoritmo

- a partir da versão 2.6.23 do *kernel*, o Linux passou a usar um algoritmo de scheduling para a classe *SCHED_OTHER* conhecido pelo nome de *justiça total* (*total fairness*);

- assuma-se um processador ideal que tem tantos elementos de processamento quantos os processos que correntemente coexistem; assim, se existirem N processos, o processador executá-los-á em paralelo distribuindo a potência de cálculo por todos eles de um modo uniforme (a velocidade de processamento será 1/N da velocidade se existisse um só processo em execução);

- o algoritmo vai procurar modelar este comportamento num processador real;

- são definidas duas variáveis associadas a cada processo:
  - *tempo de execução virtual* – tempo de execução associado ao processo se ele estivesse a ser executado no processador ideal;
  - *tempo de espera* – tempo real que o processo aguarda na fila de espera dos *processos prontos a serem executados* pela atribuição do processador;

Scheduling em Linux - 5

- o *scheduler* organiza os processos numa fila de espera dos *processos prontos a serem executados* única e calendariza para execução o processo cuja diferença entre o *tempo de espera* e o *tempo de execução virtual* é maior, procurando desta forma minimizar o grau de injustiça existente;

- sempre que um processo é calendarizado para execução, o seu *tempo de espera* é decrementado do valor correspondente à janela de execução, o dos restantes presentes na fila de espera é incrementado do mesmo valor, e todos eles têm o *tempo de execução virtual* incrementado do valor de execução virtual;

- os processos bloqueados mantêm os valores destas variáveis inalterados e não entram naturalmente no esquema de selecção enquanto não forem acordados;

- o algoritmo de *scheduling* usa, pois, como elemento central de decisão, a história passada de execução do processo, não havendo propriamente uma distinção entre os processos *I/O*-intensivos e os processos *CPU*-intensivos.

Processos vs. Threads - 1

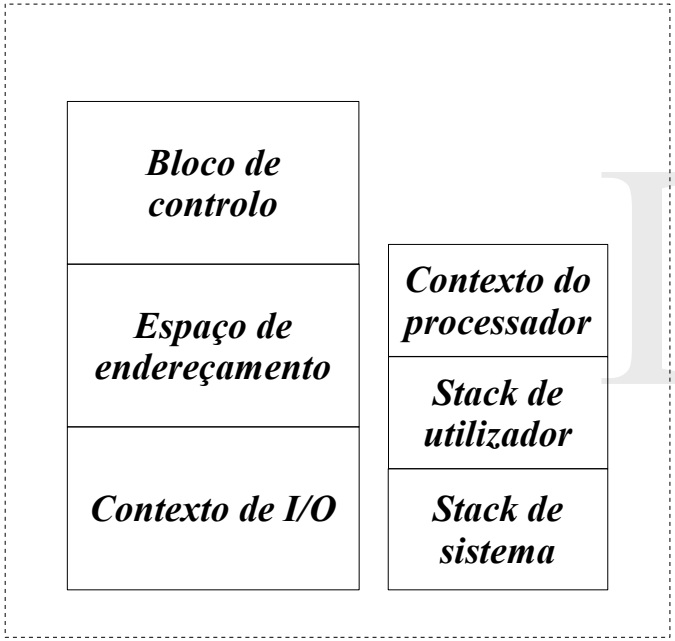
O conceito de *processo* corporiza as propriedades seguintes

- *pertença 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 (um *frame* por cada rotina invocada e que ainda não retornou).

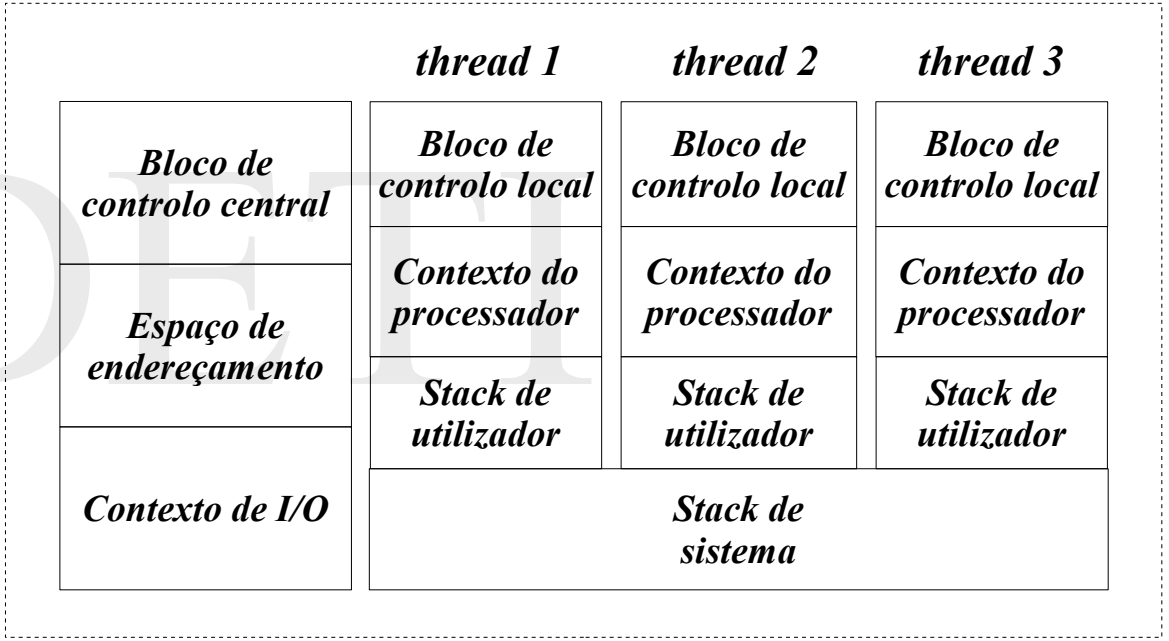
Estas propriedades, embora surjam reunidas num *processo*, podem ser tratadas separadamente pelo sistema de operação. Quando tal acontece, os *processos* dedicam-se a agrupar um conjunto de recursos e os *threads*, também conhecidos por *light weight processes*, constituem entidades executáveis independentes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

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

Processos vs. Threads - 2



Single threading



Multithreading

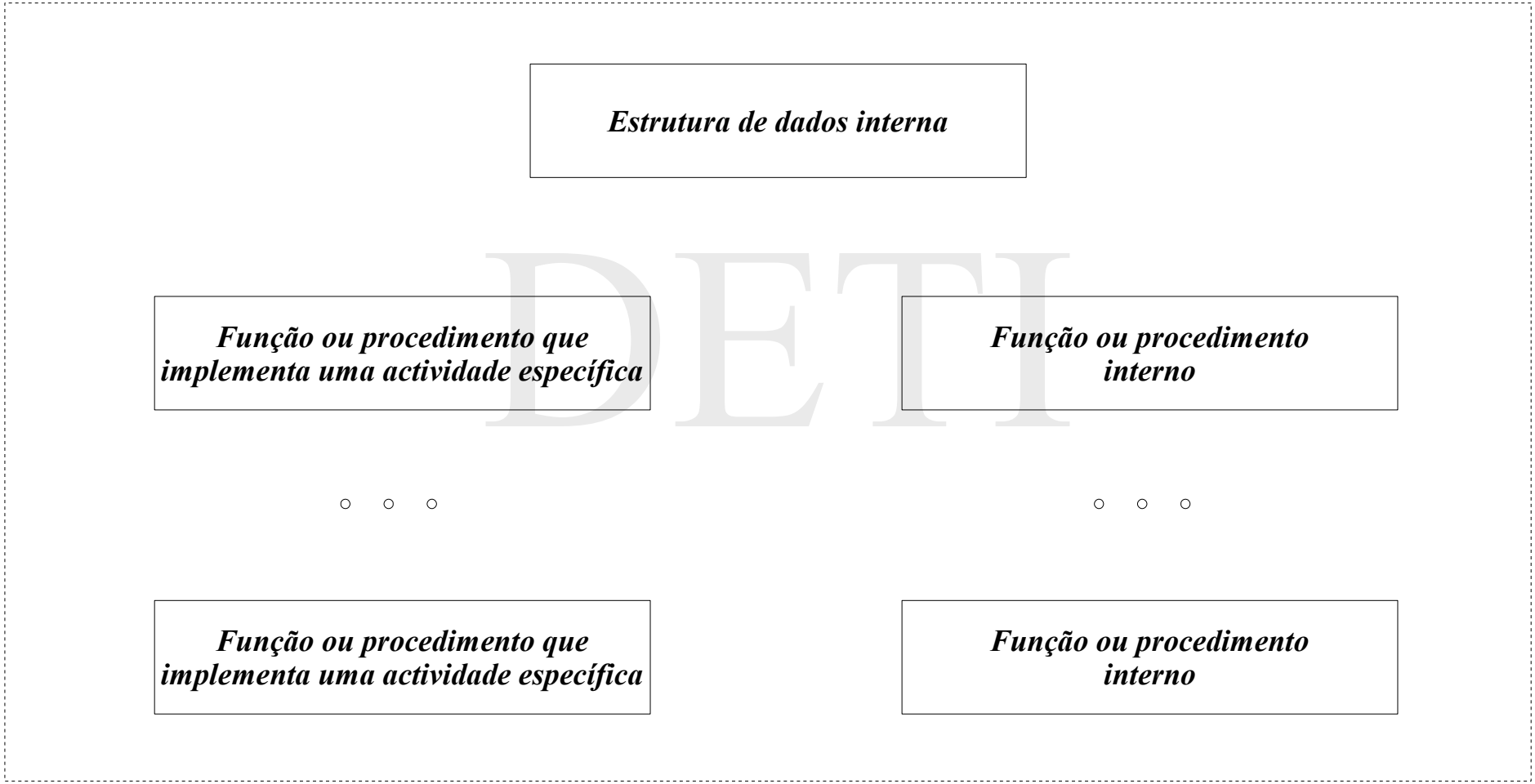
Vantagens de um ambiente multithreaded

- *maior simplicidade na decomposição da solução e maior modularidade na implementação* – programas que envolvem múltiplas actividades e atendem múltiplas solicitações são mais fáceis de conceber e de implementar numa perspectiva concorrencial do que numa perspectiva puramente sequencial

- *melhor gestão de recursos do sistema computacional* – havendo uma partilha do espaço de endereçamento e do contexto de *I/O* entre os *threads* que compõem uma aplicação, torna-se mais simples gerir a ocupação da memória principal e o acesso eficiente aos dispositivos de entrada / saída

- *eficiência e velocidade de execução* – uma decomposição da solução em *threads* por oposição a processos, ao envolver menos recursos por parte do sistema de operação, possibilita que operações como a sua criação e destruição e a mudança de contexto se tornem menos pesadas e, portanto, mais eficientes; além disso, em multiprocessadores simétricos torna-se possível calendarizar para execução em paralelo múltiplos *threads* da mesma aplicação, aumentando assim a velocidade de execução.

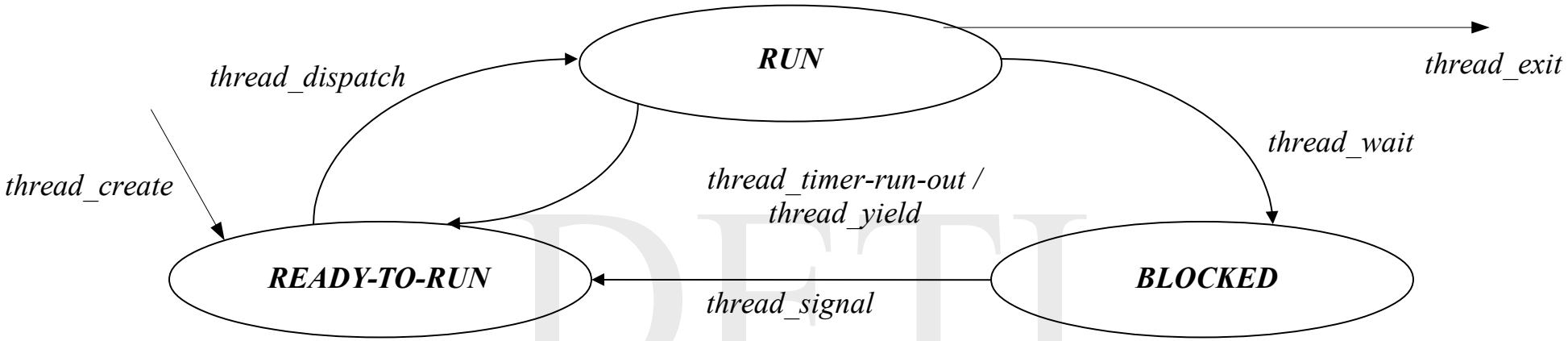
Organização de um programa multithreaded - 1



Organização de um programa multithreaded - 2

- cada *thread* está tipicamente associado à execução de uma *função ou procedimento que implementa uma actividade específica*;
- a comunicação entre os múltiplos *threads* que coexistem num dado instante 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*, representado no diagrama por uma *função ou procedimento que implementa uma actividade específica*, constitui o primeiro *thread* a ser criado e, tipicamente, o último *thread* a ser concluído.

Diagrama de estados de um thread



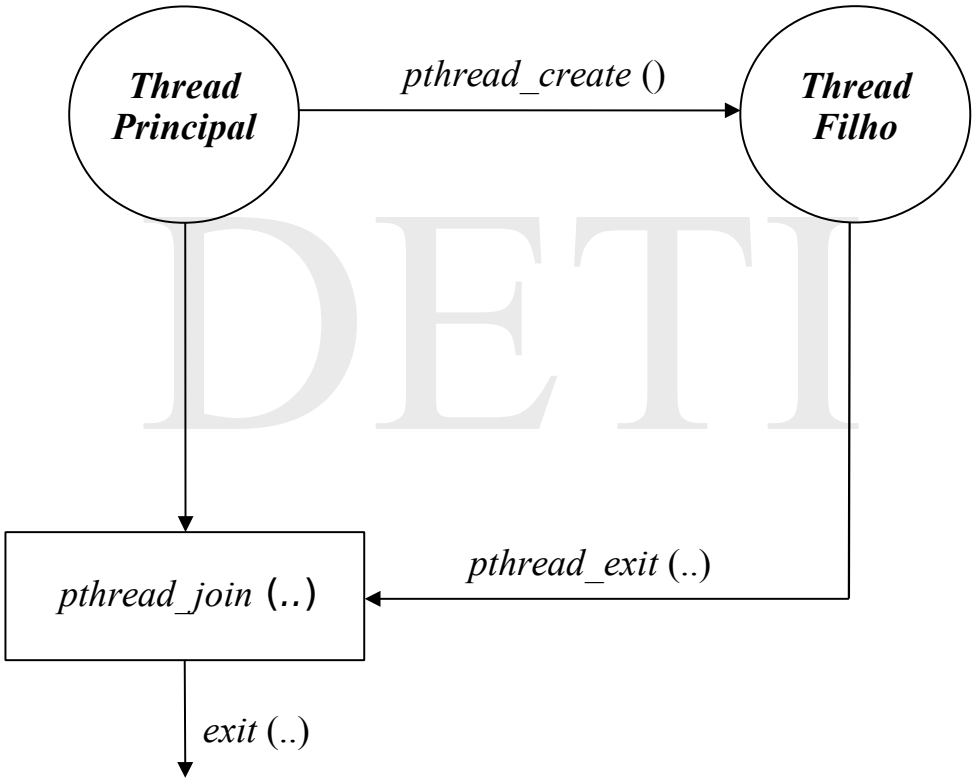
- o diagrama contém apenas os estados correspondentes à *gestão do processador* (*scheduling* de baixo nível), a *gestão da memória principal* (*scheduling* de médio nível) é um problema que se coloca apenas ao nível do processo (o espaço de endereçamento é partilhado) e a *gestão do ambiente de multiprogramação* (*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 level 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 implementação é muito versátil e portável, 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 level 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 versátil 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.

Biblioteca pthread em Unix - 1



Biblioteca pthread em Unix - 2

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

static void *threadSon (void *arg);
static int status;

/*   thread principal: é o primeiro a ser lançado */

int main (void)
{
  pthread_t thr;                              /* identificação do thread filho */
  int *status_p;       /* ponteiro para o status de execução do thread filho */

  /* lançamento do thread filho */

  if (pthread_create (&thr, NULL, threadSon, NULL) != 0)
     { perror ("erro no lançamento do thread filho");
       return EXIT_FAILURE;
     }
```

Biblioteca pthread em Unix - 3

```
  /* aguardar pela terminação do thread filho */
  if (pthread_join (thr, (void **) &status_p) != 0)
     { perror ("erro enquanto aguarda que o thread filho conclua");
       return EXIT_FAILURE;
     }
    printf ("O thread filho terminou: o seu status foi %d.\n", *status_p);
  return EXIT_SUCCESS;
}
/* thread filho */
void *threadSon (void *par)
{
  printf ("Sou o thread filho e estou vivo!\n");
  status = EXIT_SUCCESS;
  pthread_exit (&status);
}
```

Threads em Linux

O Linux lida com a questão da implementação de *threads* de um modo muito artificioso. À parte da *chamada ao sistema fork* que cria um novo processo a partir dum já existente por cópia integral do seu contexto alargado (espaço de endereçamento, contexto de *I/O* e contexto do processador), existe uma outra, *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. O único elemento distintivo é que os diversos *threads* lançados no âmbito do mesmo processo partilham a mesma identificação de grupo, o que permite que a mudança de contexto possa ser diferenciada.

Leituras sugeridas - 1

Operating Systems Concepts, Silberschatz, Galvin, Gagne, John Wiley & Sons, 8th Ed

- Capítulo 3: *Process concept* (Secções 3.1 a 3.3)
  - Construção do ambiente geral de multiprogramação
- Capítulo 4: *Multithread programming*
  - *Threads*
  - Modelo em Linux
- Capítulo 5: *Process scheduling*
  - *Scheduling* do processador

Modern Operating Systems, Tanenbaum, Prentice-Hall International Editions, 3rd Ed

- Capítulo 2: *Processes and Threads*
  - Secção 2.1: Construção do ambiente geral de multiprogramação (muito reduzido)
  - Secção 2.2: *Threads*
  - Secção 2.4: *Scheduling* do processador

Leituras sugeridas - 2

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

- Capítulo 3: *Process Description and Control* (Secções 3.1 a 3.5 e 3.7)
  - Construção do ambiente geral de multiprogramação
- Capítulo 4: *Threads* (Secções 4.1 a 4.2 e 4.6)
  - *Threads*
  - Modelo em Linux
- Capítulo 9: *Uniprocessor Scheduling*
  - *Scheduling* do processador
- Capítulo 10: *Multiprocessor and Real-Time Scheduling* (Secções 10.3 a 10.5 e 10.7)
  - Modelos em Unix SVR4 e FreeBSD e Linux