Capítulo 4

O conceito de tarefa

Um sistema de computação quase sempre tem mais atividades a executar que o número de processadores disponíveis. Assim, é necessário criar métodos para multiplexar o(s) processador(es) da máquina entre as atividades presentes. Além disso, como as diferentes tarefas têm necessidades distintas de processamento e nem sempre a capacidade de processamento existente é suficiente para atender a todos, estratégias precisam ser definidas para que cada tarefa receba uma quantidade de processamento que atenda suas necessidades. Este capítulo apresenta os principais conceitos, estratégias e mecanismos empregados na gestão do processador e das atividades em execução em um sistema de computação.

4.1 Objetivos

Em um sistema de computação, é frequente a necessidade de executar várias tarefas distintas simultaneamente. Por exemplo:

- O usuário de um computador pessoal pode estar editando uma imagem, imprimindo um relatório, ouvindo música e trazendo da Internet um novo software, tudo ao mesmo tempo.
- Em um grande servidor de e-mails, milhares de usuários conectados remotamente enviam e recebem e-mails através da rede.
- Um navegador Web precisa buscar os elementos da página a exibir, analisar e renderizar o código HTML e os gráficos recebidos, animar os elementos da interface e responder aos comandos do usuário.

Por outro lado, um processador convencional somente trata um fluxo de instruções de cada vez. Até mesmo computadores com vários processadores, vários *cores* ou com tecnologia *hyper-threading*, por exemplo, têm mais atividades a executar que o número de processadores disponíveis. Como fazer para atender simultaneamente as múltiplas necessidades de processamento dos usuários?

Uma solução ingênua para esse problema seria equipar o sistema com um processador para cada tarefa, mas essa solução ainda é inviável econômica e tecnicamente. Outra solução seria *multiplexar o processador* entre as várias tarefas que requerem processamento, ou seja, compartilhar o uso do processador entre as várias tarefas, de forma a atendê-las da melhor maneira possível.

Para uma gestão eficaz do processamento, é fundamental compreender o conceito de *tarefa*. O restante deste capítulo aborda o conceito de tarefa, como estas são definidas, quais os seus estados possíveis e como/quando elas mudam de estado.

4.2 O conceito de tarefa

Uma tarefa é definida como sendo a execução de um fluxo sequencial de instruções, construído para atender uma finalidade específica: realizar um cálculo complexo, a edição de um gráfico, a formatação de um disco, etc. Assim, a execução de uma sequência de instruções em linguagem de máquina, normalmente gerada pela compilação de um programa escrito em uma linguagem qualquer, é denominada "tarefa" ou "atividade" (do inglês *task*).

É importante ressaltar as diferenças entre os conceitos de *tarefa* e de *programa*:

Programa: é um conjunto de uma ou mais sequências de instruções escritas para resolver um problema específico, constituindo assim uma aplicação ou utilitário. O programa representa um conceito *estático*, sem um estado interno definido (que represente uma situação específica da execução) e sem interações com outras entidades (o usuário ou outros programas). Os arquivos `C:\Windows\notepad.exe` e `/usr/bin/vi` são exemplos de programas (no caso, para edição de texto).

Tarefa: é a execução sequencial, por um processador, da sequência de instruções definidas em um programa para realizar seu objetivo. Trata-se de um conceito *dinâmico*, que possui um estado interno bem definido a cada instante (os valores das variáveis internas e a posição atual da execução evoluem com o tempo) e interage com outras entidades: o usuário, os dispositivos periféricos e/ou outras tarefas. Tarefas podem ser implementadas de várias formas, como processos (Seção 5.3) ou *threads* (Seção 5.4).

Fazendo uma analogia simples, pode-se dizer que um programa é o equivalente de uma "receita de torta" dentro de um livro de receitas (um diretório) guardado em uma estante (um disco) na cozinha (o computador). Essa receita de torta define os ingredientes necessários (entradas) e o modo de preparo (programa) da torta (saída). Por sua vez, a ação de "executar" a receita, providenciando os ingredientes e seguindo os passos definidos na mesma, é a tarefa propriamente dita. A cada momento, a cozinheira (o processador) está seguindo um passo da receita (posição da execução) e tem uma certa disposição dos ingredientes e utensílios em uso (as entradas e variáveis internas da tarefa).

Assim como uma receita de torta pode definir várias atividades interdependentes para elaborar a torta (preparar a massa, fazer o recheio, assar, decorar, etc.), um programa também pode definir várias sequências de execução interdependentes para atingir seus objetivos. Por exemplo, o programa do navegador Web ilustrado na Figura 4.1 define várias tarefas que uma janela de navegador deve executar simultaneamente, para que o usuário possa navegar na Internet:

1. buscar via rede os vários elementos que compõem a página Web;
2. receber, analisar e renderizar o código HTML e os gráficos recebidos;

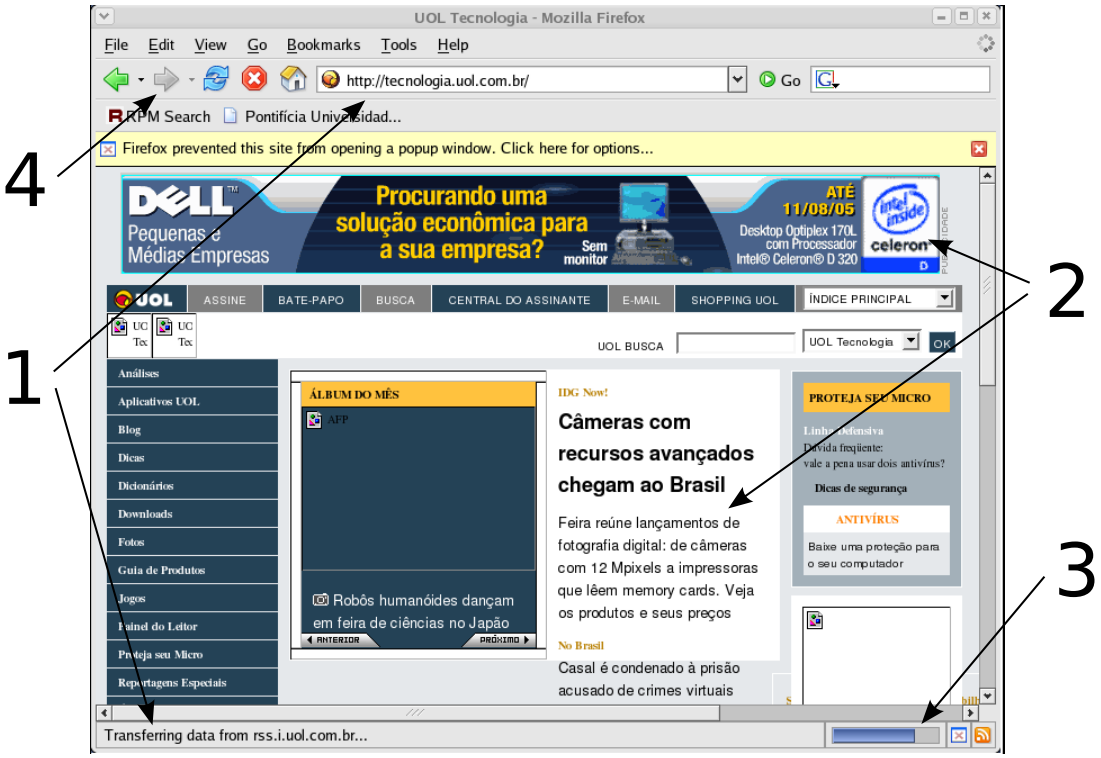


Figura 4.1: Tarefas de um navegador Internet

3. animar os diferentes elementos que compõem a interface do navegador;

4. receber e tratar os eventos do usuário (*clicks*) nos botões do navegador.

Dessa forma, as tarefas definem as atividades a serem realizadas dentro do sistema de computação. Como geralmente há muito mais tarefas a realizar que processadores disponíveis, e as tarefas não têm todas a mesma importância, a gerência de tarefas tem uma grande importância dentro de um sistema operacional.

4.3 A gerência de tarefas

Em um computador, o processador tem de executar todas as tarefas submetidas pelos usuários. Essas tarefas geralmente têm comportamento, duração e importância distintas. Cabe ao sistema operacional organizar as tarefas para executá-las e decidir em que ordem fazê-lo. Nesta seção será estudada a organização básica do sistema de gerência de tarefas e sua evolução histórica.

4.3.1 Sistemas monotarefa

Os primeiros sistemas de computação, nos anos 40, executavam apenas uma tarefa de cada vez. Nestes sistemas, cada programa binário era carregado do disco para a memória e executado até sua conclusão. Os dados de entrada da tarefa eram carregados na memória junto à mesma e os resultados obtidos no processamento eram descarregados de volta no disco após a conclusão da tarefa. Todas as operações de transferência de código e dados entre o disco e a memória eram coordenados por um

operador humano. Esses sistemas primitivos eram usados sobretudo para aplicações de cálculo numérico, muitas vezes com fins militares (problemas de trigonometria, balística, mecânica dos fluidos, etc.).

A Figura 4.2 ilustra um sistema desse tipo. Nessa figura, a etapa 1 corresponde à carga do código na memória, a etapa 2 à carga dos dados de entrada na memória, a etapa 3 ao processamento (execução propriamente dita), consumindo dados e produzindo resultados, e a etapa 4 ao término da execução, com a descarga no disco dos resultados de saída produzidos.

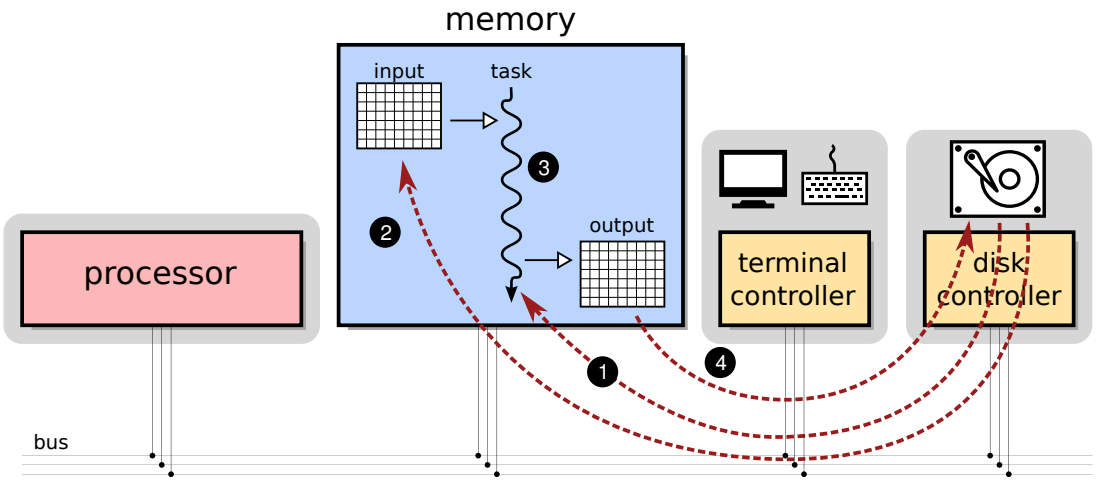


Figura 4.2: Execução de tarefa em um sistema monotarefa.

Nesse método de processamento de tarefas é possível delinear um diagrama de estados para cada tarefa executada pelo sistema, que está representado na Figura 4.3.

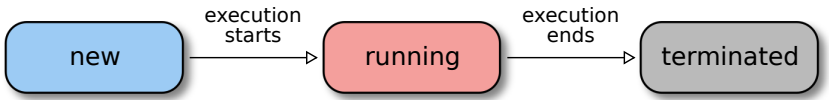


Figura 4.3: Estados de uma tarefa em um sistema monotarefa.

4.3.2 O monitor de sistema

Com a evolução do hardware, as tarefas de carga e descarga de código entre memória e disco, coordenadas por um operador humano, passaram a se tornar críticas: mais tempo era perdido nesses procedimentos manuais que no processamento da tarefa em si. Para resolver esse problema foi construído um *programa monitor*, que era carregado na memória no início da operação do sistema, com a função de gerenciar a execução dos demais programas. O programa monitor executava continuamente os seguintes passos sobre uma fila de programas a executar, armazenada no disco:

1. carregar um programa do disco para a memória;
2. carregar os dados de entrada do disco para a memória;
3. transferir a execução para o programa recém carregado;
4. aguardar o término da execução do programa;

5. escrever os resultados gerados pelo programa no disco.

Percebe-se claramente que a função do monitor é gerenciar uma fila de programas a executar, mantida no disco. Na medida em que os programas são executados pelo processador, novos programas podem ser inseridos na fila pelo operador do sistema. Além de coordenar a execução dos demais programas, o monitor também colocava à disposição destes uma biblioteca de funções para simplificar o acesso aos dispositivos de hardware (teclado, leitora de cartões, disco, etc.). Assim, o monitor de sistema constitui o precursor dos sistemas operacionais.

4.3.3 Sistemas multitarefas

O uso do monitor de sistema agilizou o uso do processador, mas outros problemas persistiam. Como a velocidade de processamento era muito maior que a velocidade de comunicação com os dispositivos de entrada e saída, o processador ficava ocioso durante os períodos de transferência de informação entre disco e memória[1]. Se a operação de entrada/saída envolvesse fitas magnéticas, o processador podia ficar parado vários minutos, aguardando a tranferência de dados. O custo dos computadores e seu consumo de energia eram elevados demais para deixá-los ociosos por tanto tempo.

A solução encontrada para resolver esse problema foi permitir ao monitor suspender a execução da tarefa que espera dados externos e passar a executar outra tarefa. Mais tarde, quando os dados de que a tarefa suspensa necessita estiverem disponíveis, ela pode ser retomada no ponto onde parou. Para tal, é necessário ter mais memória (para poder carregar mais de um programa ao mesmo tempo) e criar mecanismos no monitor para suspender uma tarefa e retomá-la mais tarde.

Uma forma simples de implementar a suspensão e retomada de tarefas de forma transparente consiste no monitor fornecer um conjunto de rotinas padronizadas de entrada/saída à tarefas; essas rotinas implementadas pelo monitor recebem as solicitações de entrada/saída de dados das tarefas e podem suspender uma execução quando for necessário, devolvendo o controle ao monitor.

A Figura 4.4 ilustra o funcionamento de um sistema multitarefa. Os passos numerados representam a execução (*start*), suspensão (*read*), retomada (*resume*) e conclusão (*exit*) de tarefas pelo monitor. Na figura, a tarefa *A* é suspensa ao solicitar uma leitura de dados, sendo retomada mais tarde, após a execução da tarefa *B*.

Essa evolução levou a sistemas mais produtivos (e complexos), nos quais várias tarefas podiam estar em andamento simultaneamente: uma estava ativa (executando) e as demais prontas (esperando pelo processador) ou suspensas (esperando dados ou eventos externos). O diagrama de estados da Figura 4.5 ilustra o comportamento de uma tarefa em um sistema desse tipo:

4.3.4 Sistemas de tempo compartilhado

Solucionado o problema de evitar a ociosidade do processador, restavam no entanto vários outros problemas a resolver. Por exemplo, o programa a seguir contém

[1] Essa diferença de velocidades permanece imensa nos sistemas atuais. Por exemplo, em um computador atual a velocidade de acesso à memória é de cerca de 5 nanossegundos ($5 \times 10^{-9}s$), enquanto a velocidade de acesso a dados em um disco rígido SATA é de cerca de 5 milissegundos ($5 \times 10^{-3}s$), ou seja, um milhão de vezes mais lento!

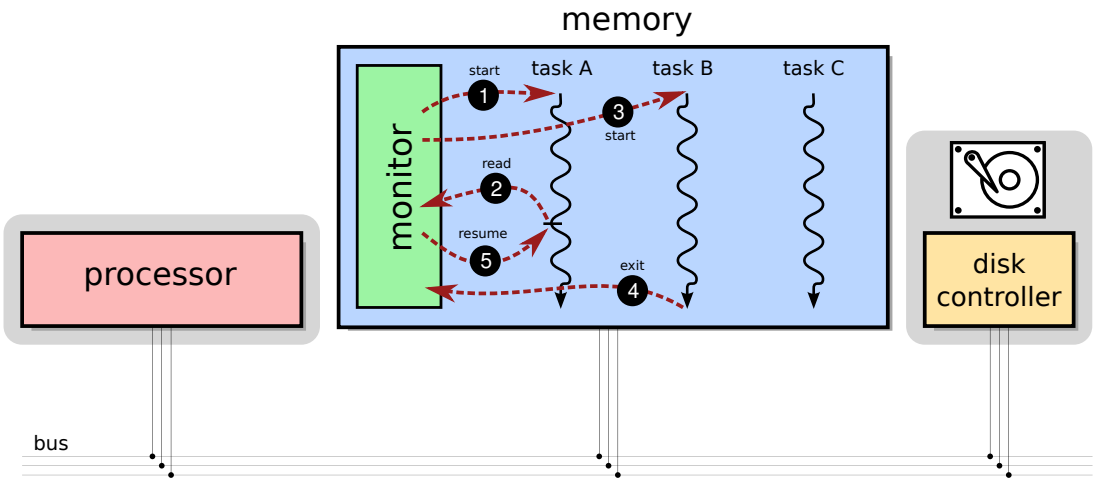


Figura 4.4: Execução de tarefas em um sistema multitarefas.

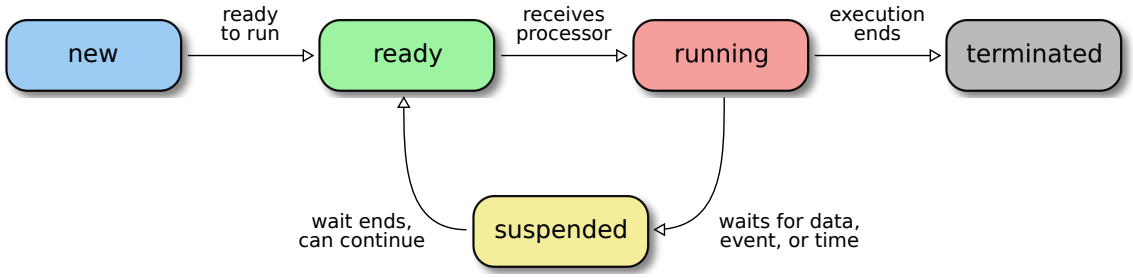


Figura 4.5: Diagrama de estados de uma tarefa em um sistema multitarefas.

um laço infinito; quando uma tarefa executar esse código, ela nunca encerrará nem será suspensa aguardando operações de entrada/saída de dados:

```
1  // calcula a soma dos primeiros 1000 inteiros
2
3  #include <stdio.h>
4
5  int main ()
6  {
7    int i = 0, soma = 0 ;
8
9    while (i <= 1000)
10     soma += i ;       // erro: o contador i não foi incrementado
11
12   printf ("A soma vale %d\n", soma);
13   exit(0) ;
14 }
```

Esse tipo de programa podia inviabilizar o funcionamento do sistema, pois a tarefa em execução nunca termina nem solicita operações de entrada/saída, monopolizando o processador e impedindo a execução das demais tarefas (pois o controle nunca volta ao monitor). Além disso, essa solução não era adequada para a criação de aplicações interativas. Por exemplo, a tarefa de um terminal de comandos pode ser suspensa a cada leitura de teclado, perdendo o processador. Se ela tiver de esperar muito para voltar ao processador, a interatividade com o usuário fica prejudicada.

Para resolver essa questão, foi introduzido no início dos anos 60 um novo conceito: o *compartilhamento de tempo*, ou *time-sharing*, através do sistema CTSS – *Compatible Time-Sharing System* [Corbató, 1963]. Nessa solução, para cada atividade que recebe o processador é definido um prazo de processamento, denominado *fatia de tempo* ou *quantum*[2]. Esgotado seu *quantum*, a tarefa em execução perde o processador e volta para uma fila de tarefas "prontas", que estão na memória aguardando sua oportunidade de executar, e outra tarefa é ativada.

O ato de retirar um recurso "à força" de uma tarefa (neste caso, o processador) é denominado *preempção*. Sistemas que implementam esse conceito são chamados *sistemas preemptivos*. Em um sistema operacional típico, a implementação da preempção por tempo usa as interrupções geradas por um temporizador programável disponível no hardware. Esse temporizador é programado para gerar interrupções em intervalos regulares (a cada milissegundo, por exemplo) que são recebidas por um tratador de interrupção (*interrupt handler*) e encaminhadas ao núcleo; essas ativações periódicas do tratador de interrupção são normalmente chamadas de *ticks*.

Quando uma tarefa recebe o processador, o *núcleo* ajusta um contador de *ticks* que essa tarefa pode usar, ou seja, seu quantum é definido em número de *ticks*. A cada *tick*, esse contador é decrementado; quando ele chegar a zero, a tarefa perde o processador e volta à fila de tarefas prontas. Essa dinâmica de execução está ilustrada na Figura 4.6.

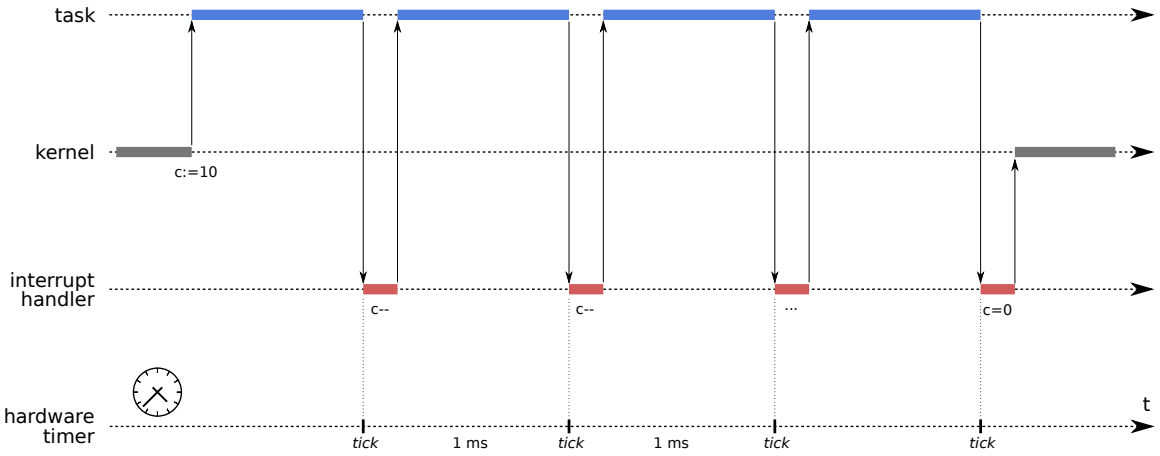


Figura 4.6: Dinâmica da preempção por tempo.

O diagrama de estados das tarefas da Figura 4.5 deve ser portanto reformulado para incluir a preempção por tempo que implementa a estratégia de tempo compartilhado. A Figura 4.7 apresenta esse novo diagrama.

[2] A duração do *quantum* depende muito do tipo de sistema operacional; no Linux ela varia de 10 a 200 milissegundos, dependendo do tipo e prioridade da tarefa [Love, 2010].

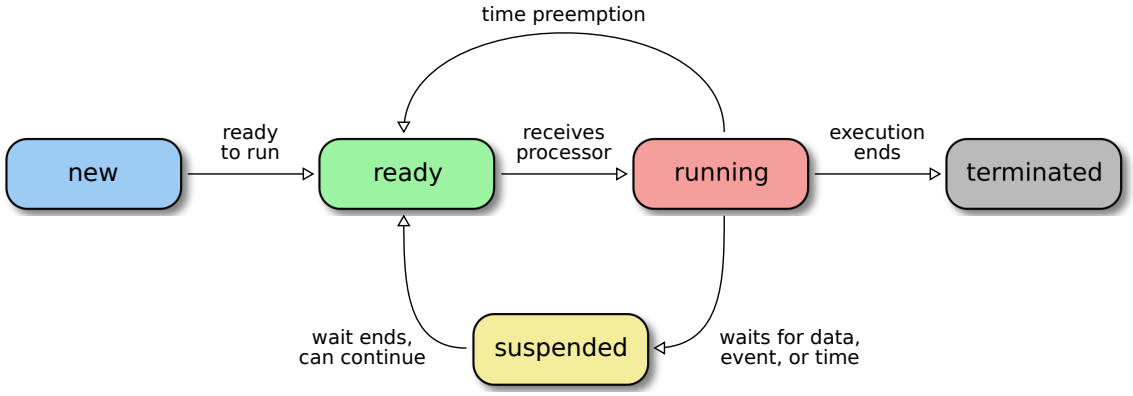


Figura 4.7: Diagrama de estados de uma tarefa em um sistema de tempo compartilhado.

4.4 Ciclo de vida das tarefas

O diagrama apresentado na Figura 4.7 é conhecido na literatura da área como *diagrama de ciclo de vida das tarefas*. Os estados e transições do ciclo de vida têm o seguinte significado:

Nova: A tarefa está sendo criada, i.e. seu código está sendo carregado em memória, junto com as bibliotecas necessárias, e as estruturas de dados do núcleo estão sendo atualizadas para permitir sua execução.

Pronta: A tarefa está em memória, pronta para iniciar ou retomar sua execução, apenas aguardando a disponibilidade do processador. Todas as tarefas prontas são organizadas em uma fila (*fila de prontas*, *ready queue* ou *run queue*), cuja ordem é determinada por algoritmos de escalonamento, que são estudados na Seção 6.

Executando: O processador está dedicado à tarefa, executando suas instruções e fazendo avançar seu estado.

Suspensa: A tarefa não pode executar porque depende de dados externos ainda não disponíveis (do disco ou da rede, por exemplo), aguarda algum tipo de sincronização (o fim de outra tarefa ou a liberação de algum recurso compartilhado) ou simplesmente espera o tempo passar (em uma operação *sleeping*, por exemplo).

Terminada: O processamento da tarefa foi encerrado e ela pode ser removida da memória do sistema.

Tão importantes quanto os estados das tarefas apresentados na Figura 4.7 são as *transições* entre esses estados, que são explicadas a seguir:

$\cdots \rightarrow$ **Nova:** Esta transição ocorre quando uma nova tarefa é admitida no sistema e começa a ser preparada para executar.

Nova $\rightarrow$ **Pronta:** ocorre quando a nova tarefa termina de ser carregada em memória, juntamente com suas bibliotecas e dados, estando pronta para executar.

Pronta $\rightarrow$ **Executando:** esta transição ocorre quando a tarefa é escolhida pelo escalonador para ser executada (ou para continuar sua execução), dentre as demais tarefas prontas.

Executando → Pronta: esta transição ocorre quando se esgota a fatia de tempo destinada à tarefa (ou seja, o fim do *quantum*); como nesse momento a tarefa não precisa de outros recursos além do processador, ela volta à fila de tarefas prontas até recebê-lo novamente.

Executando → Suspensa: caso a tarefa em execução solicite acesso a um recurso não disponível, como dados externos ou alguma sincronização, ela abandona o processador e fica suspensa até o recurso ficar disponível.

Suspensa → Pronta: quando o recurso solicitado pela tarefa se torna disponível, ela pode voltar a executar, portanto volta ao estado de pronta para aguardar o processador (que pode estar ocupado com outra tarefa).

Executando → Terminada: ocorre quando a tarefa encerra sua execução ou é abortada em consequência de algum erro (acesso inválido à memória, instrução ilegal, divisão por zero, etc.). Na maioria dos sistemas a tarefa que deseja encerrar avisa o sistema operacional através de uma chamada de sistema (no Linux é usada a chamada `exit`).

Terminada → ···: Uma tarefa terminada é removida da memória e seus registros e estruturas de controle no núcleo são liberados.

A estrutura do diagrama de ciclo de vida das tarefas pode variar de acordo com a interpretação dos autores. Por exemplo, a forma apresentada neste texto condiz com a apresentada em [Silberschatz et al., 2001] e outros autores. Por outro lado, o diagrama apresentado em [Tanenbaum, 2003] divide o estado "suspenso" em dois subestados separados: "bloqueado", quando a tarefa aguarda a ocorrência de algum evento (tempo, entrada/saída, etc.) e "suspenso", para tarefas bloqueadas que foram movidas da memória RAM para a área de troca pelo mecanismo de paginação em disco (vide Capítulo 17). Todavia, tal distinção de estados não faz mais sentido nos sistemas operacionais atuais baseados em memória paginada, pois neles os processos podem executar mesmo estando somente parcialmente carregados na memória.

Nos sistemas operacionais de mercado é possível consultar o estado das tarefas em execução no sistema. Isso pode ser feito no Windows, por exemplo, através do utilitário "Gerenciador de Tarefas". No Linux, diversos utilitários permitem gerar essa informação, como o comando `top`, cuja saída é mostrada no exemplo a seguir:

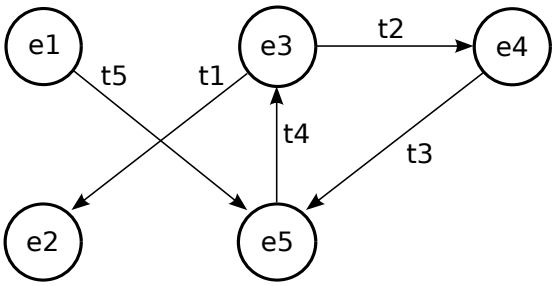
```
top - 16:58:06 up  8:26,  1 user,  load average: 6,04, 2,36, 1,08
Tarefas: 218 total,   7 executando, 211 dormindo,   0 parado,   0 zumbi
%Cpu(s): 49,7 us, 47,0 sy,  0,0 ni,  3,2 id,  0,0 wa,  0,0 hi,  0,1 si,  0,0 st
KiB Mem : 16095364 total,  9856576 free,  3134380 used,  3104408 buff/cache
KiB Swap:        0 total,        0 free,        0 used. 11858380 avail Mem

  PID USUÁRIO   PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
32703 maziero   20   0 2132220 432628 139312 S  44,8  2,7   0:53.64 Web Content
 2192 maziero   20   0 9617080 686444 248996 S  29,8  4,3  20:01.81 firefox
11650 maziero   20   0 2003888 327036 129164 R  24,0  2,0   1:16.70 Web Content
 9844 maziero   20   0 2130164 442520 149508 R  17,9  2,7   1:29.18 Web Content
11884 maziero   20   0   25276   7692   3300 S  15,5  0,0   0:37.18 bash
20425 maziero   20   0   24808   7144   3212 S  14,4  0,0   0:08.39 bash
 1782 maziero   20   0 1788328 235200  77268 S   8,7  1,5  24:12.75 gnome-shell
...
```

Nesse exemplo, existem 218 tarefas no sistema, das quais 7 estão executando (tarefas cuja coluna de Status indica "R" – *running*) e as demais estão suspensas (com Status "S" – *sleeping*). O Linux, como a maioria dos sistemas operacionais, considera que uma tarefa está executando se estiver usando ou esperando por um processador.

Exercícios

1. O que significa *time sharing* e qual a sua importância em um sistema operacional?
2. Como e com base em que critérios é escolhida a duração de um *quantum* de processamento?
3. Considerando o diagrama de estados dos processos apresentado na figura a seguir, complete o diagrama com a transição de estado que está faltando (t_6) e apresente o significado de cada um dos estados e transições.



4. Indique se cada uma das transições de estado de tarefas a seguir definidas é possível ou não. Se a transição for possível, dê um exemplo de situação na qual ela ocorre (*N*: Nova, *P*: pronta, *E*: executando, *S*: suspensa, *T*: terminada).
   - $E \rightarrow P$
   - $E \rightarrow S$
   - $S \rightarrow E$
   - $P \rightarrow N$
   - $S \rightarrow T$
   - $E \rightarrow T$
   - $N \rightarrow S$
   - $P \rightarrow S$
5. Relacione as afirmações abaixo aos respectivos estados no ciclo de vida das tarefas (N: Nova, P: Pronta, E: Executando, S: Suspensa, T: Terminada):

[ ] O código da tarefa está sendo carregado.
[ ] A tarefas são ordenadas por prioridades.
[ ] A tarefa sai deste estado ao solicitar uma operação de entrada/saída.
[ ] Os recursos usados pela tarefa são devolvidos ao sistema.
[ ] A tarefa vai a este estado ao terminar seu *quantum*.
[ ] A tarefa só precisa do processador para poder executar.
[ ] O acesso a um semáforo em uso pode levar a tarefa a este estado.
[ ] A tarefa pode criar novas tarefas.
[ ] Há uma tarefa neste estado para cada processador do sistema.
[ ] A tarefa aguarda a ocorrência de um evento externo.

Referências

F. Corbató. *The Compatible Time-Sharing System: A Programmer's Guide*. MIT Press, 1963.

R. Love. *Linux Kernel Development, Third Edition*. Addison-Wesley, 2010.

A. Silberschatz, P. Galvin, and G. Gagne. *Sistemas Operacionais – Conceitos e Aplicações*. Campus, 2001.

A. Tanenbaum. *Sistemas Operacionais Modernos, 2ª edição*. Pearson – Prentice-Hall, 2003.