Capítulo 2

Gerência de atividades

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 módulo apresenta os principais conceitos, estratégias e mecanismos empregados na gestão do processador e das atividades em execução em um sistema de computação.

2.1 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, centenas 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 o gráficos recebidos, animar os elementos da interface e responder aos comandos do usuário.

No entanto, um processador convencional somente trata um fluxo de instruções de cada vez. Até mesmo computadores com vários processadores (máquinas *Dual Pentium* ou processadores 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 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. Por multiplexar entendemos compartilhar o uso do processador entre as várias tarefas, de forma a atendê-las da melhor maneira possível.

Os principais conceitos abordados neste capítulo compreendem:

- Como as tarefas são definidas;
- Quais os estados possíveis de uma tarefa;
- Como e quando o processador muda de uma tarefa para outra;
- Como ordenar (escalonar) as tarefas para usar o processador.

2.2 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*:

Um 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). Por exemplo, os arquivos `C:\Windows\notepad.exe` e `/usr/bin/nano` são programas de edição de texto.

Uma tarefa é a execução, pelo processador, das sequências 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) e interage com outras entidades: o usuário, os periféricos e/ou outras tarefas. Tarefas podem ser implementadas de várias formas, como processos (Seção 2.4.3) ou *threads* (Seção 2.4.4).

Fazendo uma analogia clássica, 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 e o modo de preparo da torta. Por sua vez, a ação de "executar" a receita, providenciando os ingredientes e seguindo os passos definidos na receita, 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 variáveis internas da tarefa).

Assim como uma receita de torta pode definir várias atividades inter-dependentes para elaborar a torta (preparar a massa, fazer o recheio, decorar, etc.), um programa

também pode definir várias sequências de execução inter-dependentes para atingir seus objetivos. Por exemplo, o programa do navegador Web ilustrado na Figura 2.1 define várias tarefas que uma janela de navegador deve executar simultaneamente, para que o usuário possa navegar na Internet:

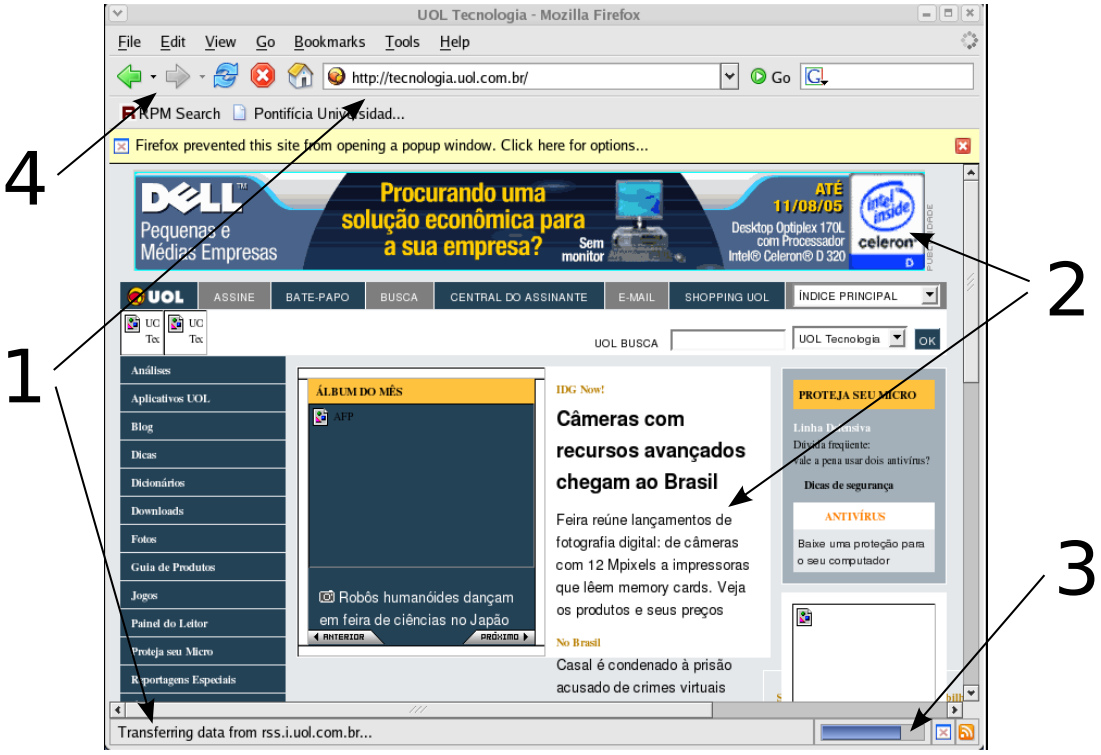

Figura 2.1: Tarefas de um navegador Internet

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

2.3 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.

2.3.1 Sistemas mono-tarefa

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 juntamente com a 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 2.2 a seguir ilustra um sistema desse tipo.

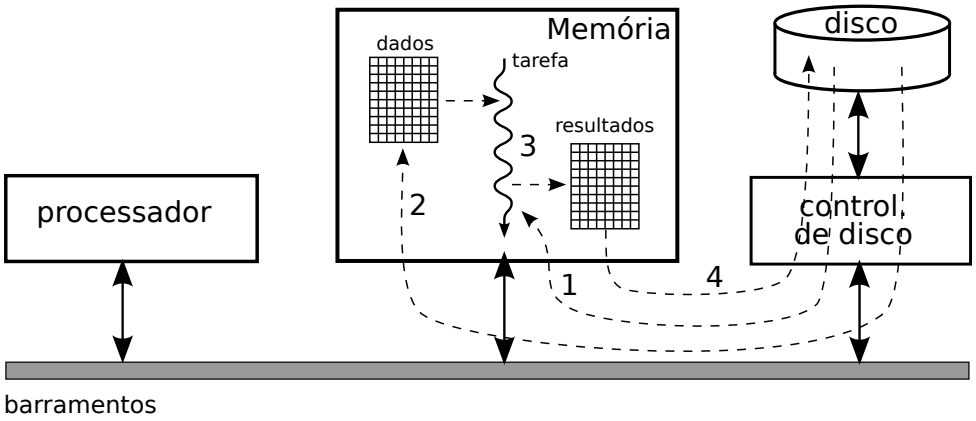

Figura 2.2: Sistema mono-tarefa: 1) carga do código na memória, 2) carga dos dados na memória, 3) processamento, consumindo dados e produzindo resultados, 4) ao término da execução, a descarga dos resultados no disco.

Nesse método de processamento de tarefas é possível delinear um diagrama de estados para cada tarefa executada pelo sistema, que está representado na Figura 2.3.

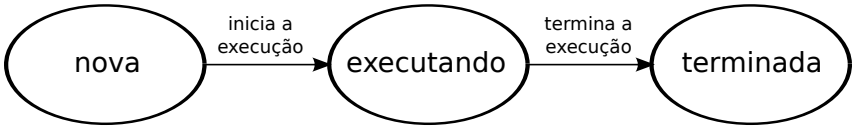

Figura 2.3: Diagrama de estados de uma tarefa em um sistema mono-tarefa.

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 coordenar 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.

2.3.2 Sistemas multi-tarefa

O uso do programa monitor 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[1], o processador ficava ocioso durante os períodos de transferência de informação entre disco e memória. Se a operação de entrada/saída envolvia fitas magnéticas, o processador podia ficar vários minutos parado, esperando. O custo dos computadores era elevado demais (e sua capacidade de processamento muito baixa) para permitir deixá-los ociosos por tanto tempo.

A solução encontrada para resolver esse problema foi permitir ao processador suspender a execução da tarefa que espera dados externos e passar a executar outra tarefa. Mais tarde, quando os dados de que necessita estiverem disponíveis, a tarefa suspensa 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 definir procedimentos para suspender uma tarefa e retomá-la mais tarde. O ato de retirar um recurso de uma tarefa (neste caso o recurso é o processador) é denominado *preempção*. Sistemas que implementam esse conceito são chamados *sistemas preemptivos*.

A adoção da preempção levou a sistemas mais produtivos (e complexos), nos quais várias tarefas podiam estar em andamento simultaneamente: uma estava ativa e as demais suspensas, esperando dados externos ou outras condições. Sistemas que suportavam essa funcionalidade foram denominados *monitores multi-tarefas*. O diagrama de estados da Figura 2.4 ilustra o comportamento de uma tarefa em um sistema desse tipo:

[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 10 nanossegundos ($10 \times 10^{-9} s$), enquanto a velocidade de acesso a dados em um disco rígido IDE é de cerca de 10 milissegundos ($10 \times 10^{-3} s$), ou seja, um milhão de vezes mais lento!

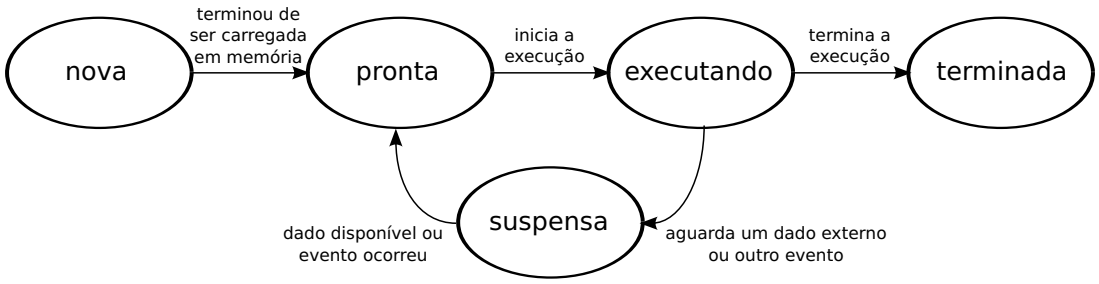

Figura 2.4: Diagrama de estados de uma tarefa em um sistema multi-tarefas.

2.3.3 Sistemas de tempo compartilhado

Solucionado o problema de evitar a ociosidade do processador, restavam no entanto vários outros problemas a resolver. Por exemplo, um programa que contém um laço infinito jamais encerra; como fazer para abortar a tarefa, ou ao menos transferir o controle ao monitor para que ele decida o que fazer? Situações como essa podem ocorrer a qualquer momento, por erros de programação ou intencionalmente, como mostra o exemplo a seguir:

```
1 void main ()
2 {
3   int i = 0, soma = 0 ;
4
5   while (i < 1000)
6     soma += i ; // erro: o contador i não foi incrementado
7
8   printf ("A soma vale %d\n", soma);
9 }
```

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, um terminal de comandos pode ser suspenso a cada leitura de teclado, perdendo o processador. Se ele 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, cada atividade que detém o processador recebe um limite de tempo de processamento, denominado *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.

Em um sistema operacional típico, a implementação da preempção por tempo tem como base as interrupções geradas pelo temporizador programável do hardware. Esse temporizador normalmente é programado para gerar interrupções em intervalos

[2] A duração atual 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, 2004].

regulares (a cada milissegundo, por exemplo) que são recebidas por um tratador de interrupção (*interrupt handler*); as ativações periódicas do tratador de interrupção são normalmente chamadas de *ticks* do relógio. 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*, o contador é decrementado; quando ele chegar a zero, a tarefa perde o processador e volta à fila de prontas. Essa dinâmica de execução está ilustrada na Figura 2.5.

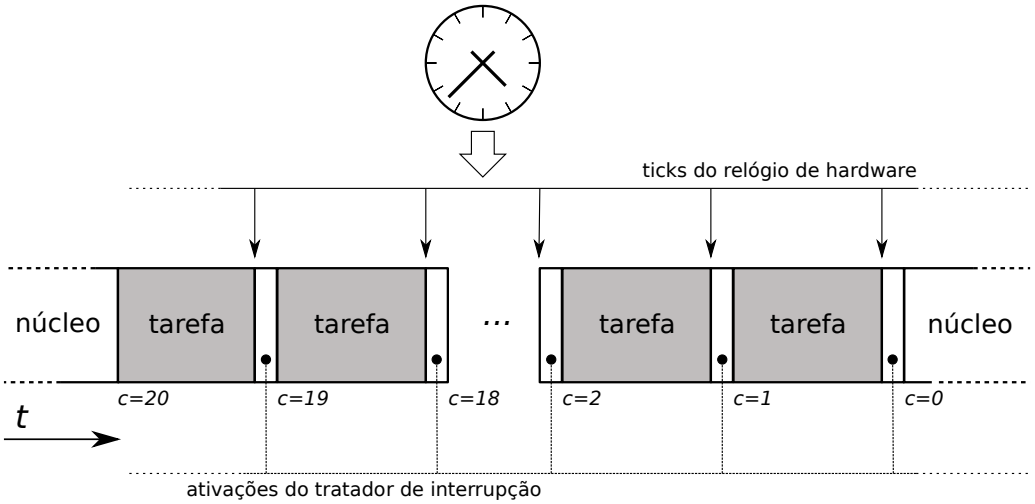

Figura 2.5: Dinâmica da preempção por tempo (*quantum* igual a 20 *ticks*).

O diagrama de estados das tarefas deve ser reformulado para incluir a preempção por tempo que implementa a estratégia de tempo compartilhado. A Figura 2.6 apresenta esse novo diagrama.

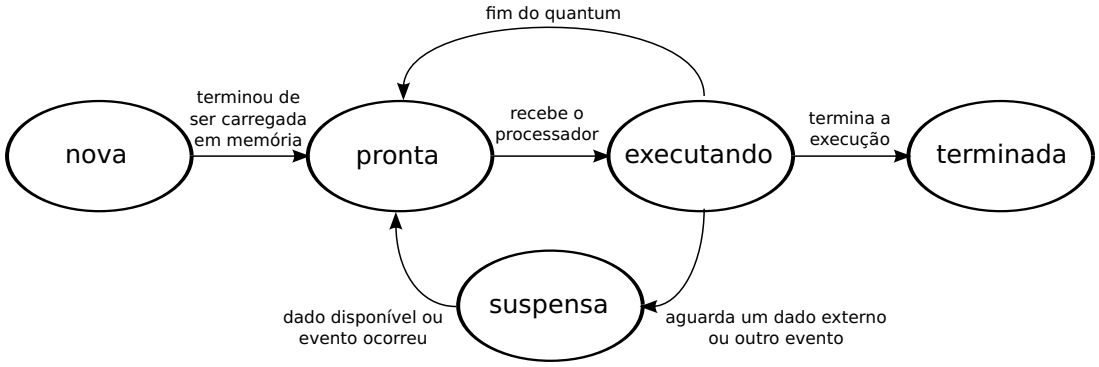

Figura 2.6: Diagrama de estados de uma tarefa em um sistema de tempo compartilhado.

2.3.4 Ciclo de vida das tarefas

O diagrama apresentado na Figura 2.6 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 executar (ou para continuar sua execução), apenas aguardando a disponibilidade do processador. Todas as tarefas prontas são organizadas em uma fila cuja ordem é determinada por algoritmos de escalonamento, que serão estudados na Seção 2.5.

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 2.6 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, dentre as demais tarefas prontas.

Executando $\rightarrow$ **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, para esperar novamente o processador.

Executando $\rightarrow$ **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 $\rightarrow \cdots$: Uma tarefa terminada é removida da memória e seus registros e estruturas de controle no núcleo são apagadas.

Executando $\rightarrow$ **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.

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 memória virtual (vide Seção 5.7). 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.

2.4 Implementação de tarefas

Conforme apresentado, uma tarefa é uma unidade básica de atividade dentro de um sistema. Tarefas podem ser implementadas de várias formas, como processos, *threads*, *jobs* e transações. Nesta seção são descritos os problemas relacionados à implementação do conceito de tarefa em um sistema operacional típico. São descritas as estruturas de dados necessárias para representar uma tarefa e as operações necessárias para que o processador possa comutar de uma tarefa para outra de forma eficiente e transparente.

2.4.1 Contextos

Na Seção 2.2 vimos que uma tarefa possui um estado interno bem definido, que representa sua situação atual: a posição de código que ela está executando, os valores de suas variáveis e os arquivos que ela utiliza, por exemplo. Esse estado se modifica conforme a execução da tarefa evolui. O estado de uma tarefa em um determinado instante é denominado **contexto**. Uma parte importante do contexto de uma tarefa diz respeito ao estado interno do processador durante sua execução, como o valor do contador de programa (PC - *Program Counter*), do apontador de pilha (SP - *Stack Pointer*) e demais registradores. Além do estado interno do processador, o contexto de uma tarefa também inclui informações sobre os recursos usados por ela, como arquivos abertos, conexões de rede e semáforos.

Cada tarefa presente no sistema possui um *descritor* associado, ou seja, uma estrutura de dados que a representa no núcleo. Nessa estrutura são armazenadas as informações relativas ao seu contexto e os demais dados necessários à sua gerência. Essa estrutura de dados é geralmente chamada de TCB (do inglês *Task Control Block*) ou PCB (*Process Control Block*). Um TCB tipicamente contém as seguintes informações:

- Identificador da tarefa (pode ser um número inteiro, um apontador, uma referência de objeto ou um identificador opaco);
- Estado da tarefa (nova, pronta, executando, suspensa, terminada, ...);

- Informações de contexto do processador (valores contidos nos registradores);
- Lista de áreas de memória usadas pela tarefa;
- Listas de arquivos abertos, conexões de rede e outros recursos usados pela tarefa (exclusivos ou compartilhados com outras tarefas);
- Informações de gerência e contabilização (prioridade, usuário proprietário, data de início, tempo de processamento já decorrido, volume de dados lidos/escritos, etc.).

Dentro do núcleo, os descritores das tarefas são organizados em listas ou vetores de TCBs. Por exemplo, normalmente há uma lista de tarefas prontas para executar, uma lista de tarefas aguardando acesso ao disco rígido, etc. Para ilustrar o conceito de TCB, o Apêndice A apresenta o TCB do núcleo Linux (versão 2.6.12), representado pela estrutura `task_struct` definida no arquivo `include/linux/sched.h`.

2.4.2 Trocas de contexto

Para que o processador possa interromper a execução de uma tarefa e retornar a ela mais tarde, sem corromper seu estado interno, é necessário definir operações para salvar e restaurar o contexto da tarefa. O ato de salvar os valores do contexto atual em seu TCB e possivelmente restaurar o contexto de outra tarefa, previamente salvo em outro TCB, é denominado **troca de contexto**. A implementação da troca de contexto é uma operação delicada, envolvendo a manipulação de registradores e flags específicos de cada processador, sendo por essa razão geralmente codificada em linguagem de máquina. No Linux as operações de troca de contexto para a plataforma Intel x86 estão definidas através de diretivas em Assembly no arquivo `arch/i386/kernel/process.c` dos fontes do núcleo.

Durante uma troca de contexto, existem questões de ordem mecânica e de ordem estratégica a serem resolvidas, o que traz à tona a separação entre mecanismos e políticas já discutida anteriormente (vide Seção 1.3). Por exemplo, o armazenamento e recuperação do contexto e a atualização das informações contidas no TCB de cada tarefa são aspectos mecânicos, providos por um conjunto de rotinas denominado **despachante** ou **executivo** (do inglês *dispatcher*). Por outro lado, a escolha da próxima tarefa a receber o processador a cada troca de contexto é estratégica, podendo sofrer influências de diversos fatores, como as prioridades, os tempos de vida e os tempos de processamento restante de cada tarefa. Essa decisão fica a cargo de um componente de código denominado **escalonador** (*scheduler*, vide Seção 2.5). Assim, o despachante implementa os mecanismos da gerência de tarefas, enquanto o escalonador implementa suas políticas.

A Figura 2.7 apresenta um diagrama temporal com os principais passos envolvidos em uma troca de contexto. A realização de uma troca de contexto completa, envolvendo a interrupção de uma tarefa, o salvamento de seu contexto, o escalonamento e a reativação da tarefa escolhida, é uma operação relativamente rápida (de dezenas a centenas de micro-segundos, dependendo do hardware e do sistema operacional).

A parte potencialmente mais demorada de uma troca de contexto é a execução do escalonador; por esta razão, muitos sistemas operacionais executam o escalonador apenas esporadicamente, quando há necessidade de reordenar a fila de tarefas prontas. Nesse caso, o despachante sempre ativa a primeira tarefa da fila.

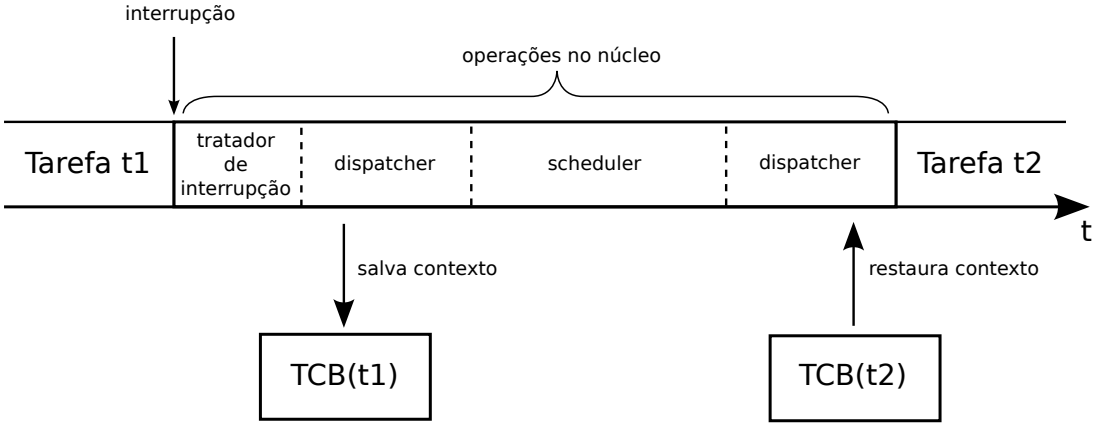

Figura 2.7: Passos de uma troca de contexto.

É importante observar que uma troca de contexto pode ser provocada pelo fim do quantum atual (através de uma interrupção de tempo), por um evento ocorrido em um periférico (também através de uma interrupção do hardware) ou pela execução de uma chamada de sistema pela tarefa corrente (ou seja, por uma interrupção de software) ou até mesmo por algum erro de execução que possa provocar uma exceção no processador.

2.4.3 Processos

Além de seu próprio código executável, cada tarefa ativa em um sistema de computação necessita de um conjunto de recursos para executar e cumprir seu objetivo. Entre esses recursos estão as áreas de memória usadas pela tarefa para armazenar seu código, dados e pilha, seus arquivos abertos, conexões de rede, etc. O conjunto dos recursos alocados a uma tarefa para sua execução é denominado **processo**.

Historicamente, os conceitos de tarefa e processo se confundem, sobretudo porque os sistemas operacionais mais antigos, até meados dos anos 80, somente suportavam uma tarefa para cada processo (ou seja, uma atividade associada a cada contexto). Essa visão vem sendo mantida por muitas referências até os dias de hoje. Por exemplo, os livros [Silberschatz et al., 2001] e [Tanenbaum, 2003] ainda apresentam processos como equivalentes de tarefas. No entanto, quase todos os sistemas operacionais contemporâneos suportam mais de uma tarefa por processo, como é o caso do Linux, Windows XP e os UNIX mais recentes.

Os sistemas operacionais convencionais atuais associam por *default* uma tarefa a cada processo, o que corresponde à execução de um programa sequencial (um único fluxo de instruções dentro do processo). Caso se deseje associar mais tarefas ao mesmo contexto (para construir o navegador Internet da Figura 2.1, por exemplo), cabe ao desenvolvedor escrever o código necessário para tal. Por essa razão, muitos livros ainda

usam de forma equivalente os termos *tarefa* e *processo*, o que não corresponde mais à realidade.

Assim, hoje em dia o processo deve ser visto como uma *unidade de contexto*, ou seja, um contêiner de recursos utilizados por uma ou mais tarefas para sua execução. Os processos são isolados entre si pelos mecanismos de proteção providos pelo hardware (isolamento de áreas de memória, níveis de operação e chamadas de sistema) e pela própria gerência de tarefas, que atribui os recursos aos processos (e não às tarefas), impedindo que uma tarefa em execução no processo p_a acesse um recurso atribuído ao processo p_b. A Figura 2.8 ilustra o conceito de processo, visto como um contêiner de recursos.

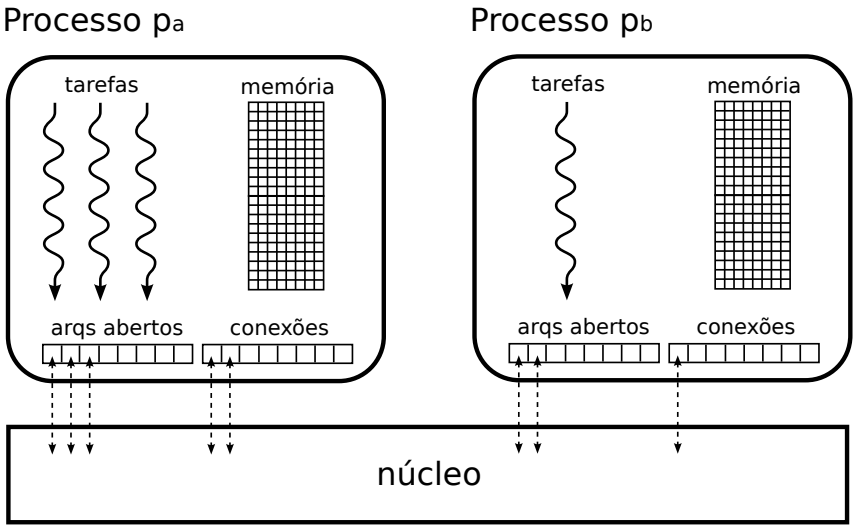

Figura 2.8: O processo visto como um contêiner de recursos.

O núcleo do sistema operacional mantém descritores de processos, denominados PCBs (*Process Control Blocks*), para armazenar as informações referentes aos processos ativos. Cada processo possui um identificador único no sistema, o PID – *Process IDentifier*. Associando-se tarefas a processos, o descritor (TCB) de cada tarefa pode ser bastante simplificado: para cada tarefa, basta armazenar seu identificador, os registradores do processador e uma referência ao processo ao qual a tarefa está vinculada. Disto observa-se também que a troca de contexto entre tarefas vinculadas ao mesmo processo é muito mais simples e rápida que entre tarefas vinculadas a processos distintos, pois somente os registradores do processador precisam ser salvos/restaurados (as áreas de memória e demais recursos são comuns às duas tarefas). Essas questões são aprofundadas na Seção 2.4.4.

Criação de processos

Durante a vida do sistema, processos são criados e destruídos. Essas operações são disponibilizadas às aplicações através de chamadas de sistema; cada sistema operacional tem suas próprias chamadas para a criação e remoção de processos. No caso do UNIX, processos são criados através da chamada de sistema `fork`, que cria uma réplica do

processo solicitante: todo o espaço de memória do processo é replicado, incluindo o código da(s) tarefa(s) associada(s) e os descritores dos arquivos e demais recursos associados ao mesmo. A Figura 2.9 ilustra o funcionamento dessa chamada.

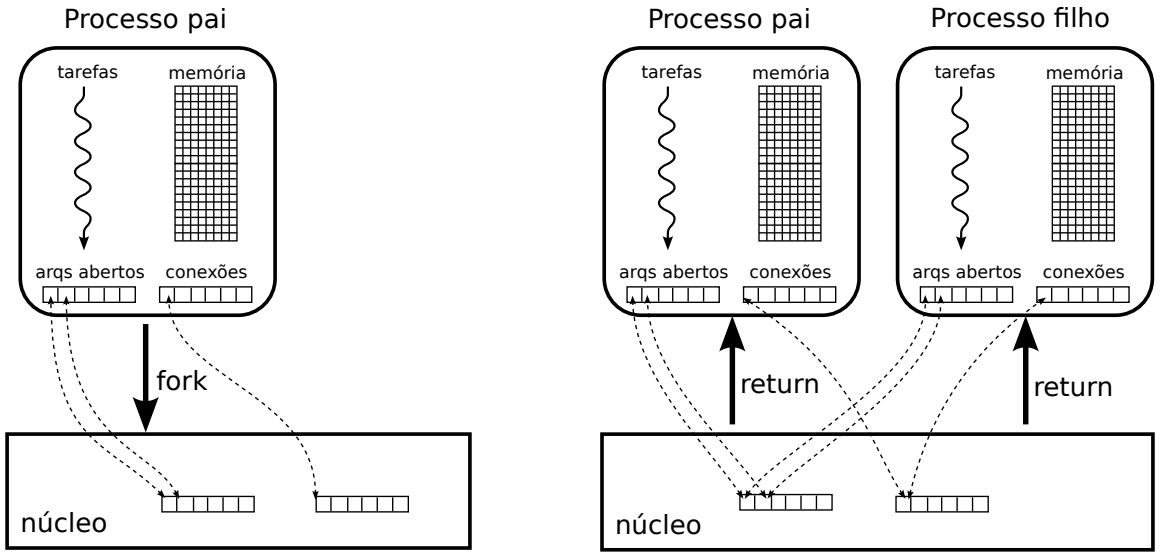

Figura 2.9: A chamada de sistema `fork`: antes (esq) e depois (dir) de sua execução pelo núcleo do sistema UNIX.

A chamada de sistema `fork` é invocada por um processo (o pai), mas dois processos recebem seu retorno: o *processo pai*, que a invocou, e o *processo filho*, recém-criado, que possui o mesmo estado interno que o pai (ele também está aguardando o retorno da chamada de sistema). Ambos os processos têm os mesmos recursos associados, embora em áreas de memória distintas. Caso o processo filho deseje abandonar o fluxo de execução herdado do processo pai e executar outro código, poderá fazê-lo através da chamada de sistema `execve`. Essa chamada substitui o código do processo que a invoca pelo código executável contido em um arquivo informado como parâmetro. A listagem a seguir apresenta um exemplo de uso dessas duas chamadas de sistema:

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

int main (int argc, char *argv[], char *envp[])
{
   int pid ;            /* identificador de processo */

   pid = fork () ;      /* replicação do processo */

   if ( pid < 0 )       /* fork não funcionou */
   {
      perror ("Erro: ") ;
      exit (-1) ;       /* encerra o processo */
   }
   else if ( pid > 0 )  /* sou o processo pai */
   {
      wait (0) ;        /* vou esperar meu filho concluir */
   }
   else                 /* sou o processo filho*/
   {
      /* carrega outro código binário para executar */
      execve ("/bin/date", argv, envp) ;
      perror ("Erro: ") ; /* execve não funcionou */
   }
   printf ("Tchau !\n") ;
   exit(0) ;            /* encerra o processo */
}
```

A chamada de sistema `exit` usada no exemplo acima serve para informar ao núcleo do sistema operacional que o processo em questão não é mais necessário e pode ser destruído, liberando todos os recursos a ele associados (arquivos abertos, conexões de rede, áreas de memória, etc.). Processos podem solicitar ao núcleo o encerramento de outros processos, mas essa operação só é aplicável a processos do mesmo usuário, ou se o processo solicitante pertencer ao administrador do sistema.

Na operação de criação de processos do UNIX aparece de maneira bastante clara a noção de **hierarquia** entre processos. À medida em que processos são criados, forma-se uma *árvore de processos* no sistema, que pode ser usada para gerenciar de forma coletiva os processos ligados à mesma aplicação ou à mesma sessão de trabalho de um usuário, pois irão constituir uma sub-árvore de processos. Por exemplo, quando um processo encerra, seus filhos são informados sobre isso, e podem decidir se também encerram ou se continuam a executar. Por outro, nos sistemas Windows, todos os processos têm o mesmo nível hierárquico, não havendo distinção entre pais e filhos. O comando `pstree` do Linux permite visualizar a árvore de processos do sistema, como mostra a listagem a seguir.

```
init-+-aacraid
     |-ahc_dv_0
     |-atd
     |-avaliacao_horac
     |-bdflush
     |-crond
     |-gpm
     |-kdm-+-X
     |     `-kdm---kdm_greet
     |-keventd
     |-khubd
     |-2*[kjournald]
     |-klogd
     |-ksoftirqd_CPU0
     |-ksoftirqd_CPU1
     |-kswapd
     |-kupdated
     |-lockd
     |-login---bash
     |-lpd---lpd---lpd
     |-5*[mingetty]
     |-8*[nfsd]
     |-nmbd
     |-nrpe
     |-oafd
     |-portmap
     |-rhnsd
     |-rpc.mountd
     |-rpc.statd
     |-rpciod
     |-scsi_eh_0
     |-scsi_eh_1
     |-smbd
     |-sshd-+-sshd---tcsh---top
     |      |-sshd---bash
     |      `-sshd---tcsh---pstree
     |-syslogd
     |-xfs
     |-xinetd
     `-ypbind---ypbind---2*[ypbind]
```

Outro aspecto importante a ser considerado em relação a processos diz respeito à comunicação. Tarefas associadas ao mesmo processo podem trocar informações facilmente, pois compartilham as mesmas áreas de memória. Todavia, isso não é possível entre tarefas associadas a processos distintos. Para resolver esse problema, o núcleo deve prover às aplicações chamadas de sistema que permitam a comunicação inter-processos (IPC – *Inter-Process Communication*). Esse tema será estudado em profundidade no Capítulo 3.

2.4.4 Threads

Conforme visto na Seção 2.4.3, os primeiros sistemas operacionais suportavam apenas uma tarefa por processo. À medida em que as aplicações se tornavam mais complexas, essa limitação se tornou um claro inconveniente. Por exemplo, um editor de textos geralmente executa tarefas simultâneas de edição, formatação, paginação e verificação ortográfica sobre a mesma massa de dados (o texto sob edição). Da mesma forma, processos que implementam servidores de rede (de arquivos, bancos de dados, etc.) devem gerenciar as conexões de vários usuários simultaneamente, que muitas vezes requisitam as mesmas informações. Essas demandas evidenciaram a necessidade de suportar mais de uma tarefa operando no mesmo contexto, ou seja, dentro do mesmo processo.

De forma geral, cada fluxo de execução do sistema, seja associado a um processo ou no interior do núcleo, é denominado **thread**. *Threads* executando dentro de um processo são chamados de **threads de usuário** (*user-level threads* ou simplesmente *user threads*). Cada *thread* de usuário corresponde a uma tarefa a ser executada dentro de um processo. Por sua vez, os fluxos de execução reconhecidos e gerenciados pelo núcleo do sistema operacional são chamados de **threads de núcleo** (*kernel-level threads* ou *kernel threads*). Os *threads* de núcleo representam tarefas que o núcleo deve realizar. Essas tarefas podem corresponder à execução dos processos no espaço de usuário, ou a atividades internas do próprio núcleo, como *drivers* de dispositivos ou tarefas de gerência.

Os sistemas operacionais mais antigos não ofereciam suporte a *threads* para a construção de aplicações. Sem poder contar com o sistema operacional, os desenvolvedores de aplicações contornaram o problema construindo bibliotecas que permitiam criar e gerenciar *threads* dentro de cada processo, sem o envolvimento do núcleo do sistema. Usando essas bibliotecas, uma aplicação pode lançar vários *threads* conforme sua necessidade, mas o núcleo do sistema irá sempre perceber (e gerenciar) apenas um fluxo de execução dentro de cada processo. Por essa razão, esta forma de implementação de *threads* é nomeada **Modelo de Threads N:1**: N *threads* no processo, mapeados em um único *thread* de núcleo. A Figura 2.10 ilustra esse modelo.

O modelo de *threads* N:1 é muito utilizado, por ser leve e de fácil implementação. Como o núcleo somente considera uma *thread*, a carga de gerência imposta ao núcleo é pequena e não depende do número de *threads* dentro da aplicação. Essa característica torna este modelo útil na construção de aplicações que exijam muitos *threads*, como jogos ou simulações de grandes sistemas (a simulação detalhada do tráfego viário de uma cidade grande, por exemplo, pode exigir um *thread* para cada veículo, resultando em centenas de milhares ou mesmo milhões de *threads*). Um exemplo de implementação desse modelo é a biblioteca *GNU Portable Threads* [Engeschall, 2005].

Entretanto, o modelo de *threads* N:1 apresenta problemas em algumas situações, sendo o mais grave deles relacionado às operações de entrada/saída. Como essas operações são intermediadas pelo núcleo, se um *thread* de usuário solicitar uma operação de E/S (recepção de um pacote de rede, por exemplo) o *thread* de núcleo correspondente será suspenso até a conclusão da operação, fazendo com que todos os *threads* de usuário associados ao processo parem de executar enquanto a operação não for concluída.

Outro problema desse modelo diz respeito à divisão de recursos entre as tarefas. O núcleo do sistema divide o tempo do processador entre os fluxos de execução que

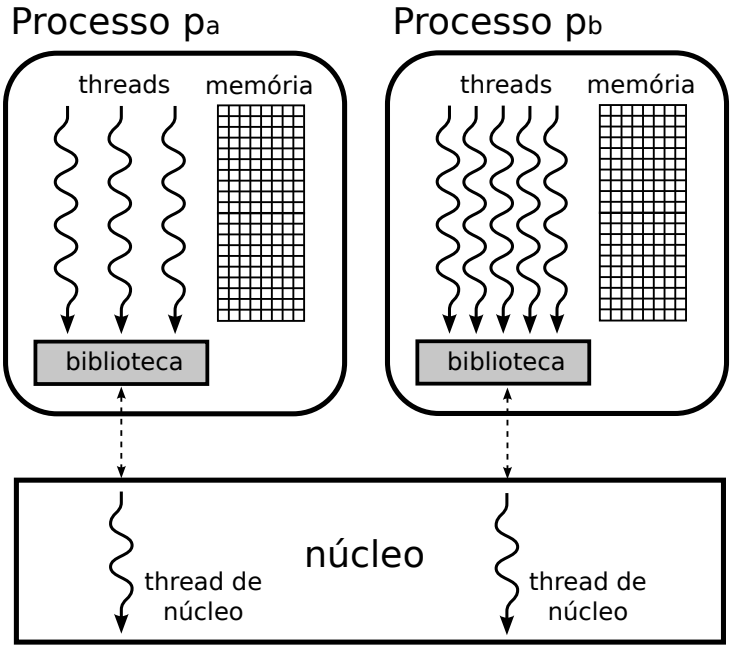

Figura 2.10: O modelo de *threads* N:1.

ele conhece e gerencia: as *threads* de núcleo. Assim, uma aplicação com 100 *threads* de usuário irá receber o mesmo tempo de processador que outra aplicação com apenas um *thread* (considerando que ambas as aplicações têm a mesma prioridade). Cada *thread* da primeira aplicação irá portanto receber 1/100 do tempo que recebe o *thread* único da segunda aplicação, o que não pode ser considerado uma divisão justa desse recurso.

A necessidade de suportar aplicações com vários *threads* (*multithreaded*) levou os desenvolvedores de sistemas operacionais a incorporar a gerência dos *threads* de usuário ao núcleo do sistema. Para cada *thread* de usuário foi então definido um *thread* correspondente dentro do núcleo, suprimindo com isso a necessidade de bibliotecas de *threads*. Caso um *thread* de usuário solicite uma operação bloqueante (leitura de disco ou recepção de pacote de rede, por exemplo), somente seu respectivo *thread* de núcleo será suspenso, sem afetar os demais *threads*. Além disso, caso o hardware tenha mais de um processador, mais *threads* da mesma aplicação podem executar ao mesmo tempo, o que não era possível no modelo anterior. Essa forma de implementação, denominada **Modelo de Threads 1:1** e apresentada na Figura 2.11, é a mais frequente nos sistemas operacionais atuais, incluindo o Windows NT e seus descendentes, além da maioria dos UNIXes.

O modelo de *threads* 1:1 é adequado para a maioria das situações e atende bem às necessidades das aplicações interativas e servidores de rede. No entanto, é pouco escalável: a criação de um grande número de *threads* impõe um carga significativa ao núcleo do sistema, inviabilizando aplicações com muitas tarefas (como grandes servidores Web e simulações de grande porte).

Para resolver o problema da escalabilidade, alguns sistemas operacionais implementam um modelo híbrido, que agrega características dos modelos anteriores. Nesse novo modelo, uma biblioteca gerencia um conjunto de *threads* de usuário (dentro do

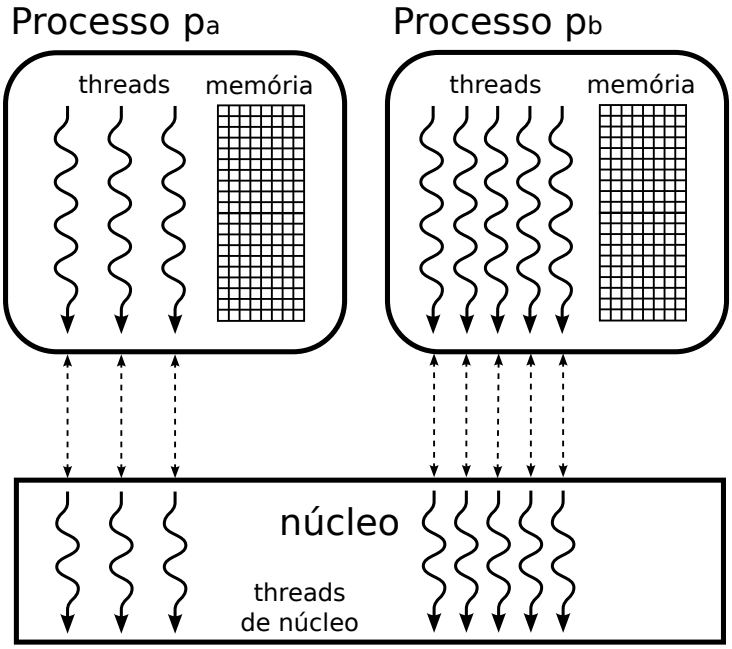

Figura 2.11: O modelo de *threads* 1:1.

processo), que é mapeado em um ou mais *threads* do núcleo. O conjunto de *threads* de núcleo associados a um processo é geralmente composto de um *thread* para cada tarefa bloqueada e mais um *thread* para cada processador disponível, podendo ser ajustado dinamicamente conforme a necessidade da aplicação. Essa abordagem híbrida é denominada **Modelo de Threads N:M**, onde N *threads* de usuário são mapeados em $M \leq N$ *threads* de núcleo. A Figura 2.12 apresenta esse modelo.

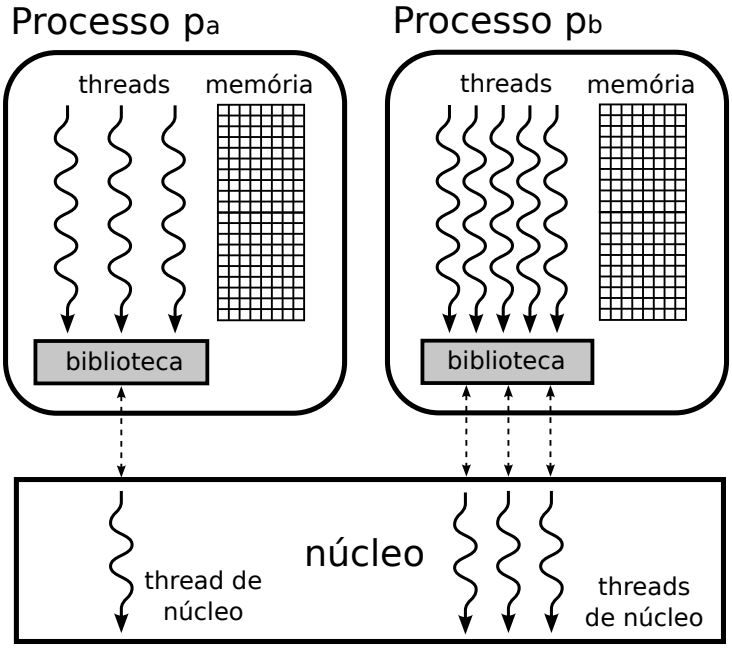

Figura 2.12: O modelo de *threads* N:M.

Modelo	**N:1**	**1:1**	**N:M**
Resumo	Todos os N *threads* do processo são mapeados em um único *thread* de núcleo	Cada *thread* do processo tem um *thread* correspondente no núcleo	Os N *threads* do processo são mapeados em um conjunto de M *threads* de núcleo
Local da implementação	bibliotecas no nível usuário	dentro do núcleo	em ambos
Complexidade	baixa	média	alta
Custo de gerência para o núcleo	nulo	médio	alto
Escalabilidade	alta	baixa	alta
Suporte a vários processadores	não	sim	sim
Velocidade das trocas de contexto entre *threads*	rápida	lenta	rápida entre *threads* no mesmo processo, lenta entre *threads* de processos distintos
Divisão de recursos entre tarefas	injusta	justa	variável, pois o mapeamento *thread*→ processador é dinâmico
Exemplos	GNU Portable Threads	Windows XP, Linux	Solaris, FreeBSD KSE

Tabela 2.1: Quadro comparativo dos modelos de *threads*.

O modelo N:M é implementado pelo Solaris e também pelo projeto KSE (*Kernel-Scheduled Entities*) do FreeBSD [Evans and Elischer, 2003] baseado nas idéias apresentadas em [Anderson et al., 1992]. Ele alia as vantagens de maior interatividade do modelo 1:1 à maior escalabilidade do modelo N:1. Como desvantagens desta abordagem podem ser citadas sua complexidade de implementação e maior custo de gerência dos *threads* de núcleo, quando comparado ao modelo 1:1. A Tabela 2.1 resume os principais aspectos dos três modelos de implementação de *threads* e faz um comparativo entre eles.

No passado, cada sistema operacional definia sua própria interface para a criação de *threads*, o que levou a problemas de portabilidade das aplicações. Em 1995 foi definido o padrão *IEEE POSIX 1003.1c*, mais conhecido como *PThreads* [Nichols et al., 1996], que busca definir uma interface padronizada para a criação e manipulação de *threads* na linguagem C. Esse padrão é amplamente difundido e suportado hoje em dia. A listagem a seguir, extraída de [Barney, 2005], exemplifica o uso do padrão *PThreads* (para compilar deve ser usada a opção de ligação `-lpthread`).

```
1  #include <pthread.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4
5  #define NUM_THREADS 5
6
7  /* cada thread vai executar esta função */
8  void *PrintHello (void *threadid)
9  {
10    printf ("%d: Hello World!\n", (int) threadid);
11    pthread_exit (NULL);
12 }
13
14 /* thread "main" (vai criar as demais threads) */
15 int main (int argc, char *argv[])
16 {
17    pthread_t thread[NUM_THREADS];
18    int status, i;
19
20    /* cria as demais threads */
21    for(i = 0; i < NUM_THREADS; i++)
22    {
23       printf ("Creating thread %d\n", i);
24       status = pthread_create (&thread[i], NULL, PrintHello, (void *) i);
25
26       if (status) /* ocorreu um erro */
27       {
28          perror ("pthread_create");
29          exit (-1);
30       }
31    }
32
33    /* encerra a thread "main" */
34    pthread_exit (NULL);
35 }
```

O conceito de *threads* também pode ser utilizado em outras linguagens de programação, como Java, Python, Perl, C++ e C#. O código a seguir traz um exemplo simples de criação de *threads* em Java (extraído da documentação oficial da linguagem):

```
1  public class MyThread extends Thread {
2     int threadID;
3
4     MyThread (int ID) {
5        threadID = ID;
6     }
7
8     public void run () {
9        int i ;
10
11       for (i = 0; i< 100 ; i++)
12         System.out.println ("Hello from t" + threadID + "!") ;
13    }
14
15    public static void main (String args[]) {
16       MyThread t1 = new MyThread (1);
17       MyThread t2 = new MyThread (2);
18       MyThread t3 = new MyThread (3);
19
20       t1.start ();
21       t2.start ();
22       t3.start ();
23    }
24 }
```

2.5 Escalonamento de tarefas

Um dos componentes mais importantes da gerência de tarefas é o **escalonador** (*task scheduler*), que decide a ordem de execução das tarefas prontas. O algoritmo utilizado no escalonador define o comportamento do sistema operacional, permitindo obter sistemas que tratem de forma mais eficiente e rápida as tarefas a executar, que podem ter características diversas: aplicações interativas, processamento de grandes volumes de dados, programas de cálculo numérico, etc.

Antes de se definir o algoritmo usado por um escalonador, é necessário ter em mente a natureza das tarefas que o sistema irá executar. Existem vários critérios que definem o comportamento de uma tarefa; uma primeira classificação possível diz respeito ao seu comportamento temporal:

Tarefas de tempo real : exigem previsibilidade em seus tempos de resposta aos eventos externos, pois geralmente estão associadas ao controle de sistemas críticos, como processos industriais, tratamento de fluxos multimídia, etc. O escalonamento de tarefas de tempo real é um problema complexo, fora do escopo deste livro e discutido mais profundamente em [Burns and Wellings, 1997, Farines et al., 2000].

Tarefas interativas : são tarefas que recebem eventos externos (do usuário ou através da rede) e devem respondê-los rapidamente, embora sem os requisitos de previsibilidade das tarefas de tempo real. Esta classe de tarefas inclui a maior parte das aplicações dos sistemas *desktop* (editores de texto, navegadores Internet, jogos) e dos servidores de rede (e-mail, web, bancos de dados).