Sistemas Operacionais: Conceitos e Mecanismos

II - Gerência de Tarefas

Prof. Carlos Alberto Maziero
DInf UFPR
http://www.inf.ufpr.br/maziero

4 de agosto de 2017

Este texto está licenciado sob a Licença *Attribution-NonCommercial-ShareAlike 3.0 Unported* da *Creative Commons* (CC). Em resumo, você deve creditar a obra da forma especificada pelo autor ou licenciante (mas não de maneira que sugira que estes concedem qualquer aval a você ou ao seu uso da obra). Você não pode usar esta obra para fins comerciais. Se você alterar, transformar ou criar com base nesta obra, você poderá distribuir a obra resultante apenas sob a mesma licença, ou sob uma licença similar à presente. Para ver uma cópia desta licença, visite http://creativecommons.org/licenses/by-nc-sa/3.0/.

Este texto foi produzido usando exclusivamente software livre: Sistema Operacional *GNU/Linux* (distribuições *Fedora* e *Ubuntu*), compilador de texto LaTeX 2_ε, gerenciador de referências *BibTeX*, editor gráfico *Inkscape*, criadores de gráficos *GNUPlot* e *GraphViz* e processador PS/PDF *GhostScript*, entre outros.

Sumário

1 Objetivos **3**

2 O conceito de tarefa **4**

3 A gerência de tarefas **6**
3.1 Sistemas monotarefa 6
3.2 Sistemas multitarefa 7
3.3 Sistemas de tempo compartilhado 8
3.4 Ciclo de vida das tarefas 9

4 Implementação de tarefas **11**
4.1 Contextos 11
4.2 Trocas de contexto 12
4.3 Processos 13
4.3.1 Criação de processos 15
4.4 Threads 18

5 Escalonamento de tarefas **23**
5.1 Objetivos e métricas 24
5.2 Escalonamento preemptivo e cooperativo 25
5.3 Escalonamento FCFS (*First-Come, First Served*) 26
5.4 Escalonamento SJF (*Shortest Job First*) 29
5.5 Escalonamento por prioridades 30
5.5.1 Definição de prioridades 32
5.5.2 Inanição e envelhecimento de tarefas 33
5.5.3 Inversão e herança de prioridades 35
5.6 Outros algoritmos de escalonamento 38
5.7 Um escalonador real 38

A O *Task Control Block* do Linux **41**

Resumo

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.

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 os 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 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 4.3) ou *threads* (Seção 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 interdependentes para elaborar a torta (preparar a massa, fazer o recheio, 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 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;
3. Animar os diferentes elementos que compõem a interface do navegador;
4. Receber e tratar os eventos do usuário (*clicks*) nos botões do navegador;

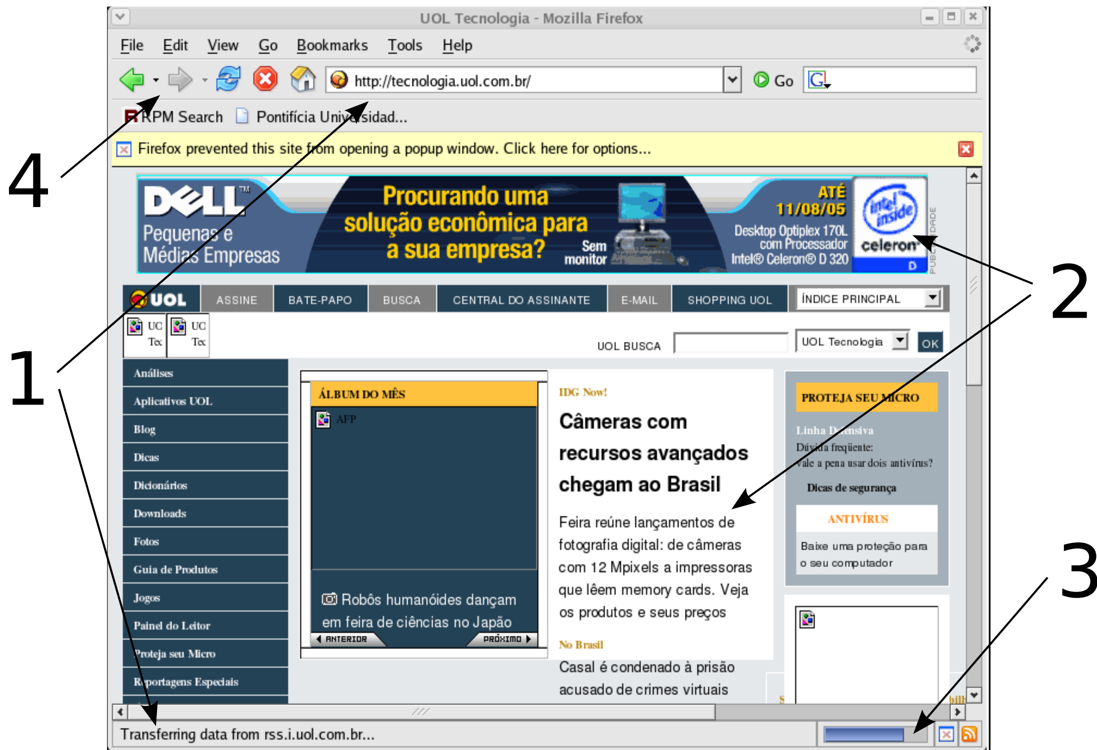

Figura 1: Tarefas de um navegador Internet

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.

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.

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 2 a seguir ilustra um sistema desse tipo.

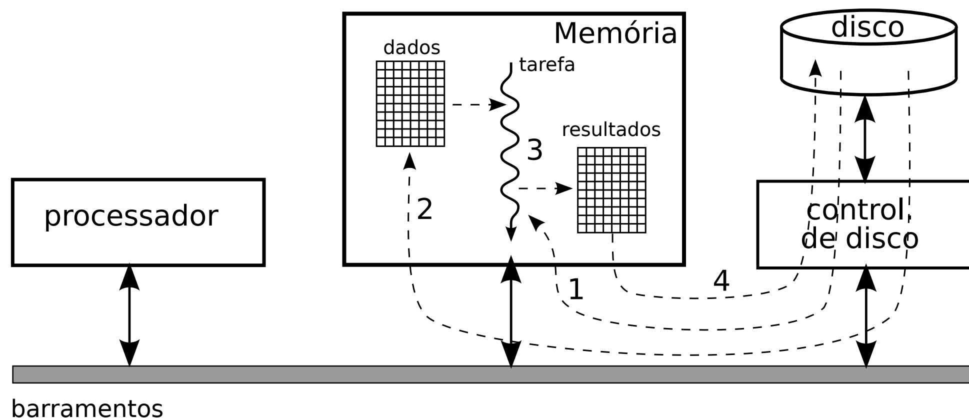

Figura 2: Sistema monotarefa: 1) carga do código na memória, 2) carga dos dados na memória, 3) processamento, consumindo dados e produzindo resultados, 4) ao término da execução, a descarga dos resultados no disco.

Nesse método de processamento de tarefas é possível delinear um diagrama de estados para cada tarefa executada pelo sistema, que está representado na Figura 3.

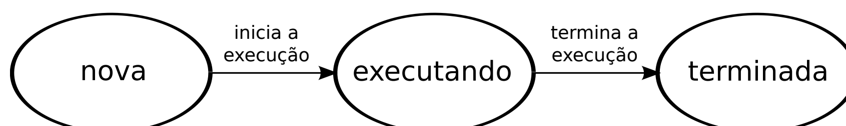

Figura 3: Diagrama de estados de uma tarefa em um sistema monotarefa.

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.

3.2 Sistemas multitarefa

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

[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!

e as demais suspensas, esperando dados externos ou outras condições. Sistemas que suportavam essa funcionalidade foram denominados *monitores multitarefas*. O diagrama de estados da Figura 4 ilustra o comportamento de uma tarefa em um sistema desse tipo:

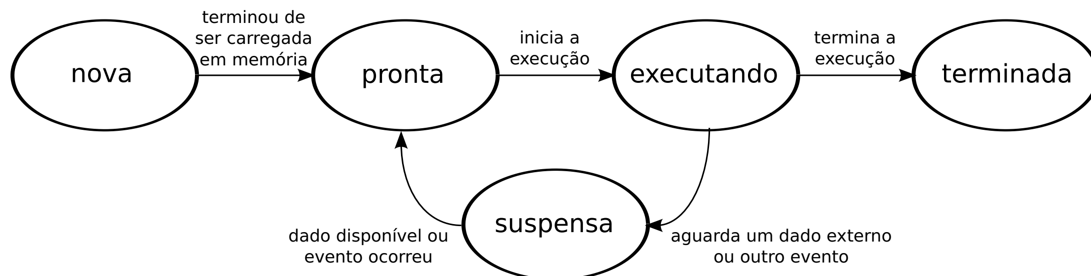

Figura 4: Diagrama de estados de uma tarefa em um sistema multitarefas.

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 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 5.

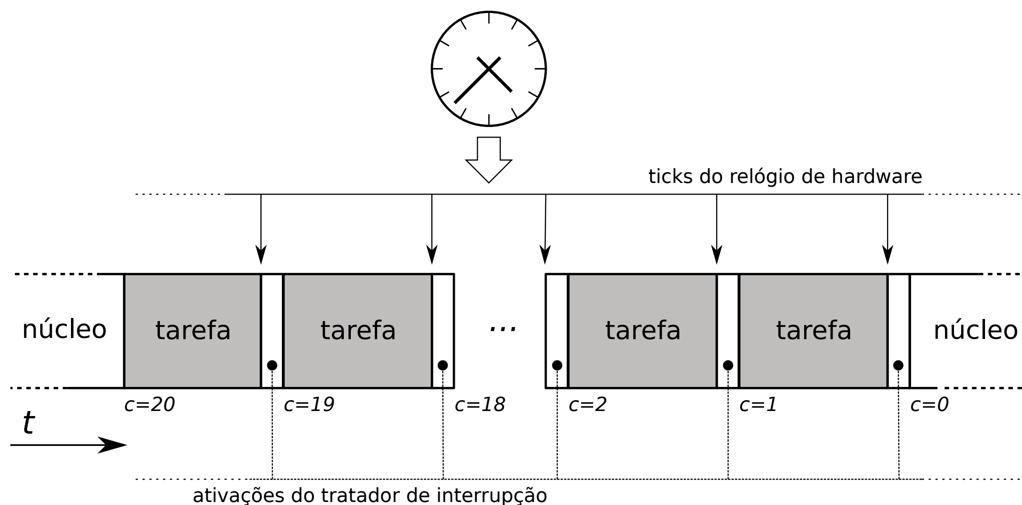

Figura 5: Dinâmica da preempção por tempo (*quantum* igual a 20 *ticks*).

O diagrama de estados das tarefas deve ser reformulado para incluir a preempção por tempo que implementa a estratégia de tempo compartilhado. A Figura 6 apresenta esse novo diagrama.

3.4 Ciclo de vida das tarefas

O diagrama apresentado na Figura 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.

[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].

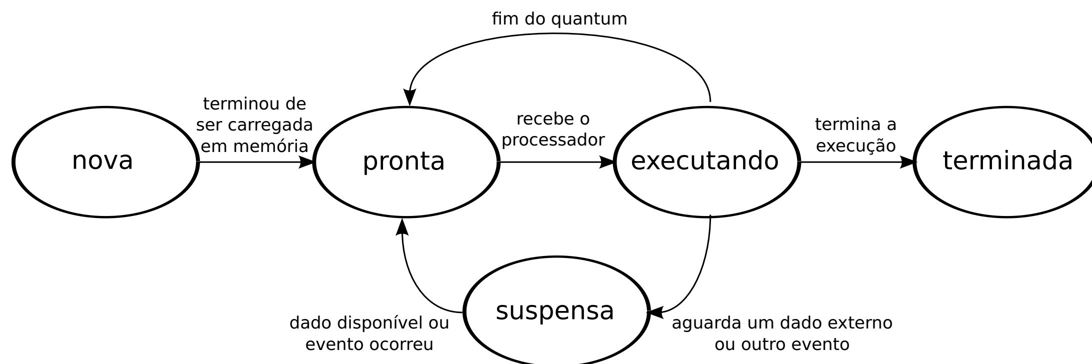

Figura 6: Diagrama de estados de uma tarefa em um sistema de tempo compartilhado.

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 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 6 são as *transições* entre esses estados, que são explicadas a seguir:

··· → **Nova** : Esta transição ocorre quando uma nova tarefa é admitida no sistema e começa a ser preparada para executar.

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

Pronta → Executando : esta transição ocorre quando a tarefa é escolhida pelo escalonador para ser executada, 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, para esperar novamente o processador.

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 apagadas.

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.

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 **??**). 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.

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.

4.1 Contextos

Na Seção 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`.

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 **??**). 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 5). Assim, o despachante implementa os mecanismos da gerência de tarefas, enquanto o escalonador implementa suas políticas.

A Figura 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 microssegundos, 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.

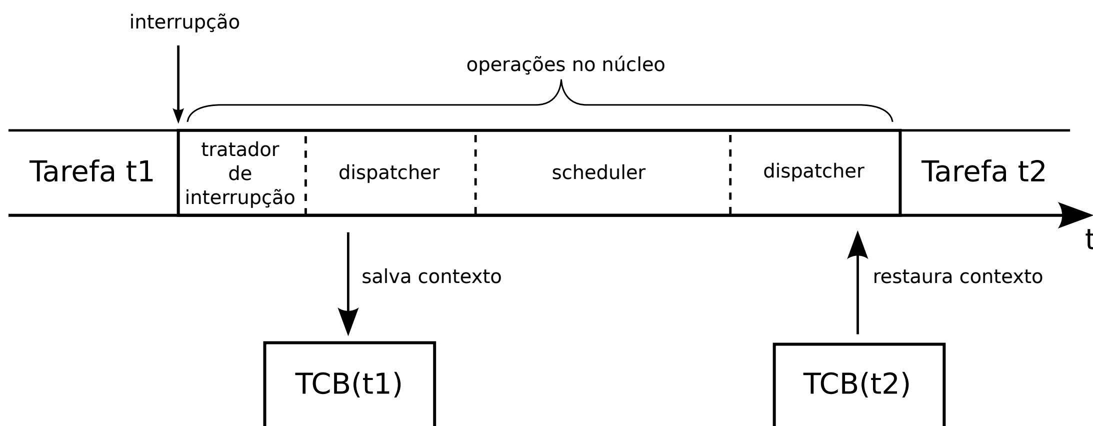

Figura 7: Passos de uma troca de contexto.

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.

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 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 8 ilustra o conceito de processo, visto como um contêiner de recursos.

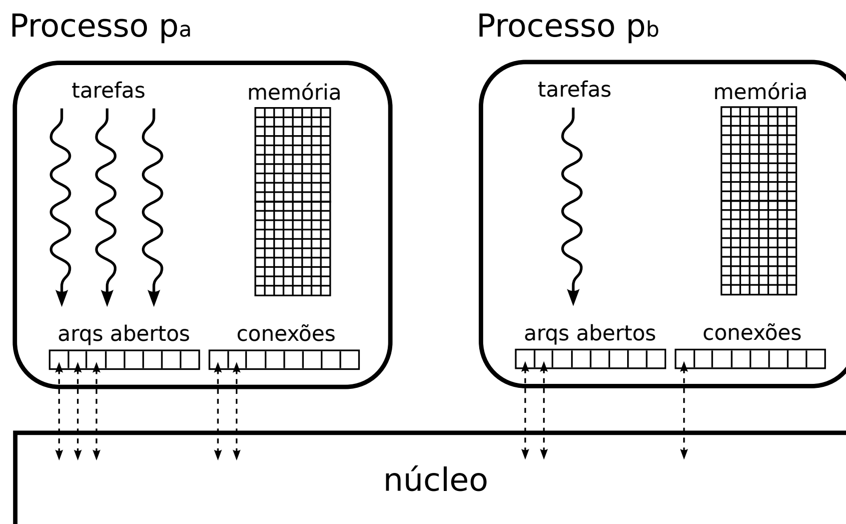

Figura 8: O processo visto como um contêiner de recursos.

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 4.4.

4.3.1 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 9 ilustra o funcionamento dessa chamada.

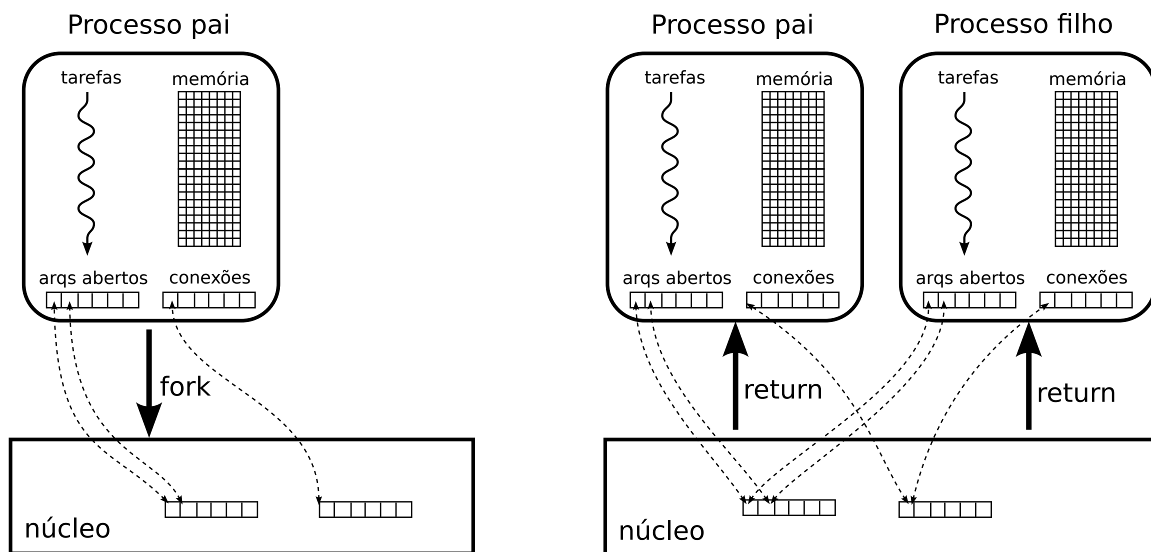

Figura 9: A chamada de sistema `fork`: antes (esq) e depois (dir) de sua execução pelo núcleo do sistema UNIX.

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 interprocessos (IPC – *Inter-Process Communication*). Esse tema será estudado em profundidade no Capítulo **??**.

4.4 Threads

Conforme visto na Seção 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 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

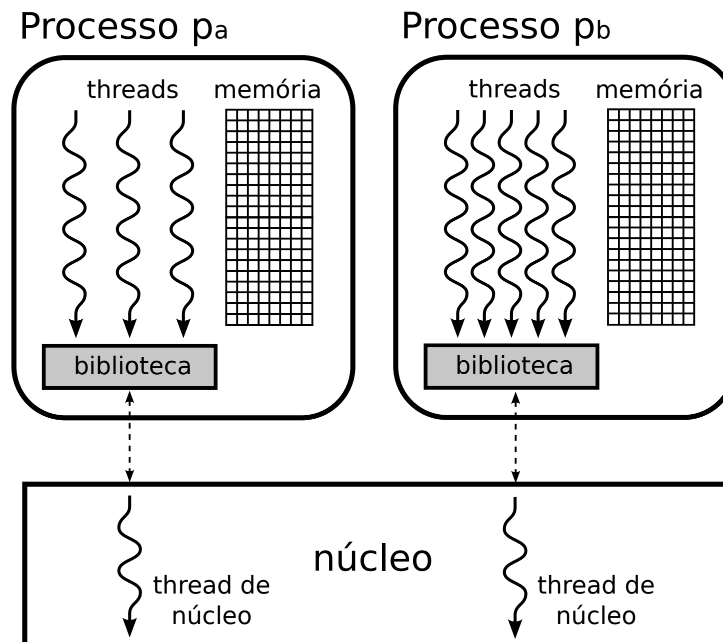

Figura 10: O modelo de *threads* N:1.

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 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 uma 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

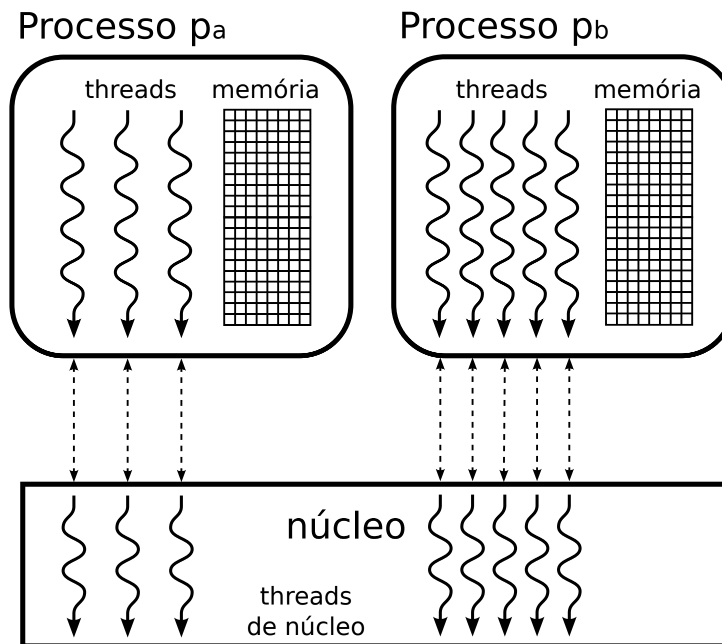

Figura 11: O modelo de *threads* 1:1.

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 12 apresenta esse modelo.

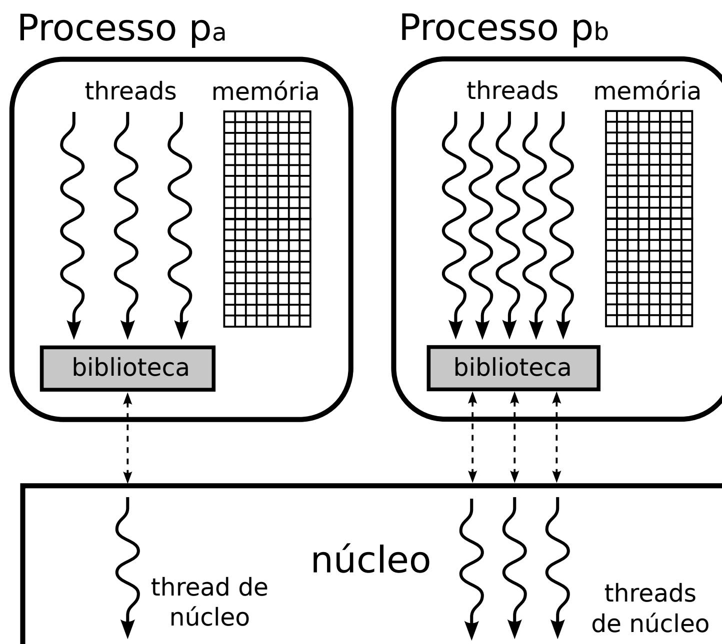

Figura 12: O modelo de *threads* N:M.

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 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 ideias 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 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):

```
public class MyThread extends Thread {
   int threadID;

   MyThread (int ID) {
      threadID = ID;
   }

   public void run () {
      int i ;

      for (i = 0; i< 100 ; i++)
        System.out.println ("Hello from t" + threadID + "!") ;
   }

   public static void main (String args[]) {
      MyThread t1 = new MyThread (1);
      MyThread t2 = new MyThread (2);
      MyThread t3 = new MyThread (3);

      t1.start ();
      t2.start ();
      t3.start ();
   }
}
```

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).

Tarefas em lote (*batch*) : são tarefas sem requisitos temporais explícitos, que normalmente executam sem intervenção do usuário, como procedimentos de *backup*, varreduras de antivírus, cálculos numéricos longos, renderização de animações, etc.

Além dessa classificação, as tarefas também podem ser classificadas de acordo com seu comportamento no uso do processador:

Tarefas orientadas a processamento (*CPU-bound tasks*): são tarefas que usam intensivamente o processador na maior parte de sua existência. Essas tarefas passam a maior parte do tempo nos estados *pronta* ou *executando*. A conversão de arquivos de vídeo e outros processamentos numéricos longos (como os feitos pelo projeto *SETI@home* [Anderson et al., 2002]) são bons exemplos desta classe de tarefas.

Tarefas orientadas a entrada/saída (*IO-bound tasks*): são tarefas que dependem muito mais dos dispositivos de entrada/saída que do processador. Essas tarefas despendem boa parte de suas existências no estado *suspenso*, aguardando respostas às suas solicitações de leitura e/ou escrita de dados nos dispositivos de entrada/saída. Exemplos desta classe de tarefas incluem editores, compiladores e servidores de rede.

É importante observar que uma tarefa pode mudar de comportamento ao longo de sua execução. Por exemplo, um conversor de arquivos de áudio WAV→MP3 alterna constantemente entre fases de processamento e de entrada/saída, até concluir a conversão dos arquivos desejados.

5.1 Objetivos e métricas

Ao se definir um algoritmo de escalonamento, deve-se ter em mente seu objetivo. Todavia, os objetivos do escalonador são muitas vezes contraditórios; o desenvolvedor do sistema tem de escolher o que priorizar, em função do perfil das aplicações a suportar. Por exemplo, um sistema interativo voltado à execução de jogos exige valores de quantum baixos, para que cada tarefa pronta receba rapidamente o processador (provendo maior interatividade). Todavia, valores pequenos de quantum implicam em uma menor eficiência $\mathcal{E}$ no uso do processador, conforme visto na Seção 4.2. Vários critérios podem ser definidos para a avaliação de escalonadores; os mais frequentemente utilizados são:

Tempo de execução ou de vida (*turnaround time*, t_t): diz respeito ao tempo total da "vida" de cada tarefa, ou seja, o tempo decorrido entre a criação da tarefa e seu encerramento, computando todos os tempos de processamento e de espera. É uma medida típica de sistemas em lote, nos quais não há interação direta com os usuários do sistema. Não deve ser confundido com o **tempo de processamento** (t_p), que é o tempo total de uso de processador demandado pela tarefa.

Tempo de espera (*waiting time*, t_w): é o tempo total perdido pela tarefa na fila de tarefas prontas, aguardando o processador. Deve-se observar que esse tempo não inclui os tempos de espera em operações de entrada/saída (que são inerentes à aplicação).

Tempo de resposta (*response time*, t_r): é o tempo decorrido entre a chegada de um evento ao sistema e o resultado imediato de seu processamento. Por exemplo, o tempo decorrido entre apertar uma tecla e o caractere correspondente aparecer na tela, em um editor de textos. Essa medida de desempenho é típica de sistemas interativos, como sistemas desktop e de tempo real; ela depende sobretudo da rapidez no tratamento das interrupções de hardware pelo núcleo e do valor do *quantum* de tempo, para permitir que as tarefas cheguem mais rápido ao processador quando saem do estado suspenso.

Justiça : este critério diz respeito à distribuição do processador entre as tarefas prontas: duas tarefas de comportamento similar devem receber tempos de processamento similares e ter durações de execução similares.

Eficiência : a eficiência $\mathcal{E}$, conforme definido na Seção 4.2, indica o grau de utilização do processador na execução das tarefas do usuário. Ela depende sobretudo da rapidez da troca de contexto e da quantidade de tarefas orientadas a entrada/saída no sistema (tarefas desse tipo geralmente abandonam o processador antes do fim do *quantum*, gerando assim mais trocas de contexto que as tarefas orientadas a processamento).

5.2 Escalonamento preemptivo e cooperativo

O escalonador de um sistema operacional pode ser preemptivo ou cooperativo (não-cooperativo):

Sistemas preemptivos : nestes sistemas uma tarefa pode perder o processador caso termine seu *quantum* de tempo, execute uma chamada de sistema ou caso ocorra uma interrupção que acorde uma tarefa mais prioritária (que estava suspensa aguardando um evento). A cada interrupção, exceção ou chamada de sistema, o escalonador pode reavaliar todas as tarefas da fila de prontas e decidir se mantém ou substitui a tarefa atualmente em execução.

Sistemas cooperativos : a tarefa em execução permanece no processador tanto quanto possível, só abandonando o mesmo caso termine de executar, solicite uma operação de entrada/saída ou libere explicitamente o processador, voltando à fila de tarefas prontas (isso normalmente é feito através de uma chamada de sistema `sched_yield()` ou similar). Esses sistemas são chamados de *cooperativos* por exigir a cooperação das tarefas entre si na gestão do processador, para que todas possam executar.

Atualmente a maioria dos sistemas operacionais de uso geral atuais é preemptiva. Sistemas mais antigos, como o Windows 3.*, PalmOS 3 e MacOS 8 e 9 operavam de forma cooperativa.

Em um sistema preemptivo, normalmente as tarefas só são interrompidas quando o processador está no modo usuário; a *thread* de núcleo correspondente a cada tarefa não sofre interrupções. Entretanto, os sistemas mais sofisticados implementam a preempção de tarefas também no modo núcleo. Essa funcionalidade é importante para sistemas de

tempo real, pois permite que uma tarefa de alta prioridade chegue mais rapidamente ao processador quando for reativada. Núcleos de sistema que oferecem essa possibilidade são denominados **núcleos preemptivos**; Solaris, Linux 2.6 e Windows NT são exemplos de núcleos preemptivos.

5.3 Escalonamento FCFS (*First-Come, First Served*)

A forma de escalonamento mais elementar consiste em simplesmente atender as tarefas em sequência, à medida em que elas se tornam prontas (ou seja, conforme sua ordem de chegada na fila de tarefas prontas). Esse algoritmo é conhecido como FCFS – *First Come - First Served* – e tem como principal vantagem sua simplicidade.

Para dar um exemplo do funcionamento do algoritmo FCFS, consideremos as tarefas na fila de tarefas prontas, com suas durações previstas de processamento e datas de ingresso no sistema, descritas na tabela a seguir:

tarefa	t_1	t_2	t_3	t_4
ingresso	0	0	1	3
duração	5	2	4	3

O diagrama da Figura 13 mostra o escalonamento do processador usando o algoritmo FCFS cooperativo (ou seja, sem *quantum* ou outras interrupções). Os quadros sombreados representam o uso do processador (observe que em cada instante apenas uma tarefa ocupa o processador). Os quadros brancos representam as tarefas que já ingressaram no sistema e estão aguardando o processador (tarefas prontas).

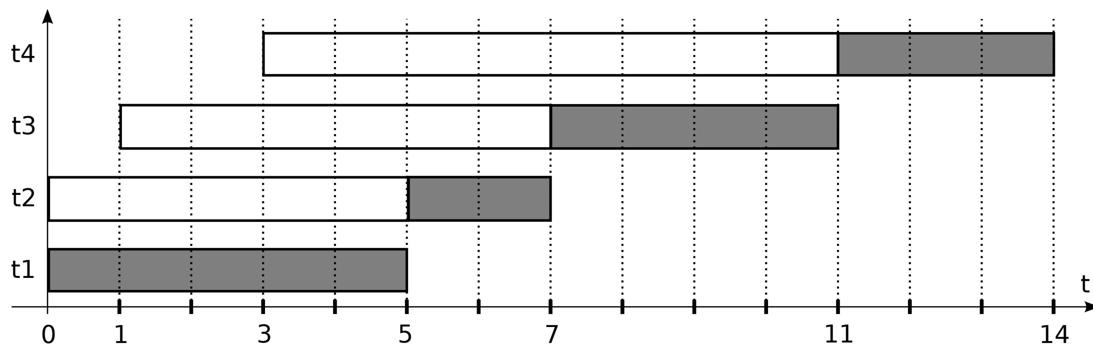

Figura 13: Escalonamento FCFS.

Calculando o tempo médio de execução (T_t, a média de $t_t(t_i)$) e o tempo médio de espera (T_w, a média de $t_w(t_i)$) para o algoritmo FCFS, temos:

$$
\begin{aligned}
T_t &= \frac{t_t(t_1) + t_t(t_2) + t_t(t_3) + t_t(t_4)}{4} = \frac{(5-0) + (7-0) + (11-1) + (14-3)}{4} \\
&= \frac{5 + 7 + 10 + 11}{4} = \frac{33}{4} = 8.25s
\end{aligned}
$$

$$
\begin{aligned}
T_w &= \frac{t_w(t_1) + t_w(t_2) + t_w(t_3) + t_w(t_4)}{4} = \frac{(0-0) + (5-0) + (7-1) + (11-3)}{4} \\
&= \frac{0+5+6+8}{4} = \frac{19}{4} = 4.75s
\end{aligned}
$$

A adição da preempção por tempo ao escalonamento FCFS dá origem a outro algoritmo de escalonamento bastante popular, conhecido como **escalonamento por revezamento**, ou *Round-Robin*. Considerando as tarefas definidas na tabela anterior e um quantum $t_q = 2s$, seria obtida a sequência de escalonamento apresentada na Figura 14.

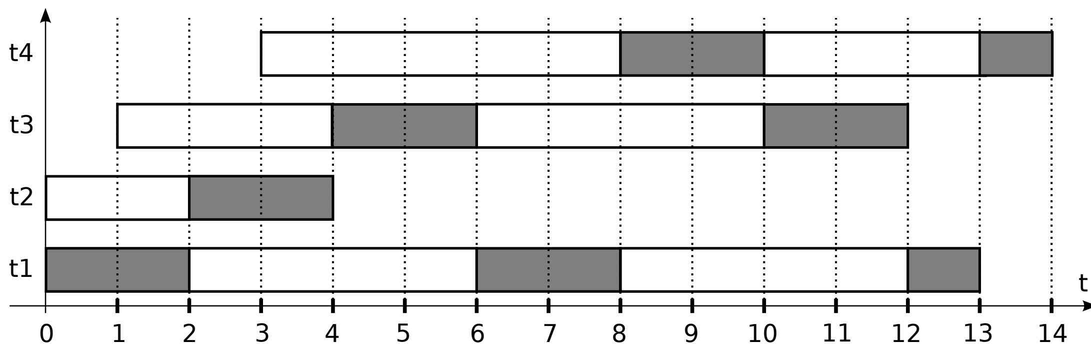

Figura 14: Escalonamento *Round-Robin*.

Na Figura 14, é importante observar que a execução das tarefas não obedece uma sequência óbvia como $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \rightarrow t_1 \rightarrow t_2 \rightarrow \ldots$, mas uma sequência bem mais complexa: $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_1 \rightarrow t_4 \rightarrow t_3 \rightarrow \ldots$. Isso ocorre por causa da ordem das tarefas na fila de tarefas prontas. Por exemplo, a tarefa t_1 para de executar e volta à fila de tarefas prontas no instante $t = 2$, antes de t_4 ter entrado no sistema (em $t = 3$). Por isso, t_1 retorna ao processador antes de t_4 (em $t = 6$). A Figura 15 detalha a evolução da fila de tarefas prontas ao longo do tempo, para esse exemplo.

Calculando o tempo médio de execução T_t e o tempo médio de espera T_w para o algoritmo *round-robin*, temos:

$$
\begin{aligned}
T_t &= \frac{t_t(t_1) + t_t(t_2) + t_t(t_3) + t_t(t_4)}{4} = \frac{(13-0) + (4-0) + (12-1) + (14-3)}{4} \\
&= \frac{13+4+11+11}{4} = \frac{39}{4} = 9.75s
\end{aligned}
$$

$$
T_w = \frac{t_w(t_1) + t_w(t_2) + t_w(t_3) + t_w(t_4)}{4} = \frac{8+2+7+8}{4} = \frac{25}{4} = 6.25s
$$

Observa-se o aumento nos tempos T_t e T_w em relação ao algoritmo FCFS simples, o que mostra que o algoritmo *round-robin* é menos eficiente para a execução de tarefas em lote. Entretanto, por distribuir melhor o uso do processador entre as tarefas ao longo do tempo, ele pode proporcionar tempos de resposta bem melhores às aplicações interativas.

O aumento da quantidade de trocas de contexto também tem um impacto negativo na eficiência do sistema operacional. Quanto menor o número de trocas de contexto

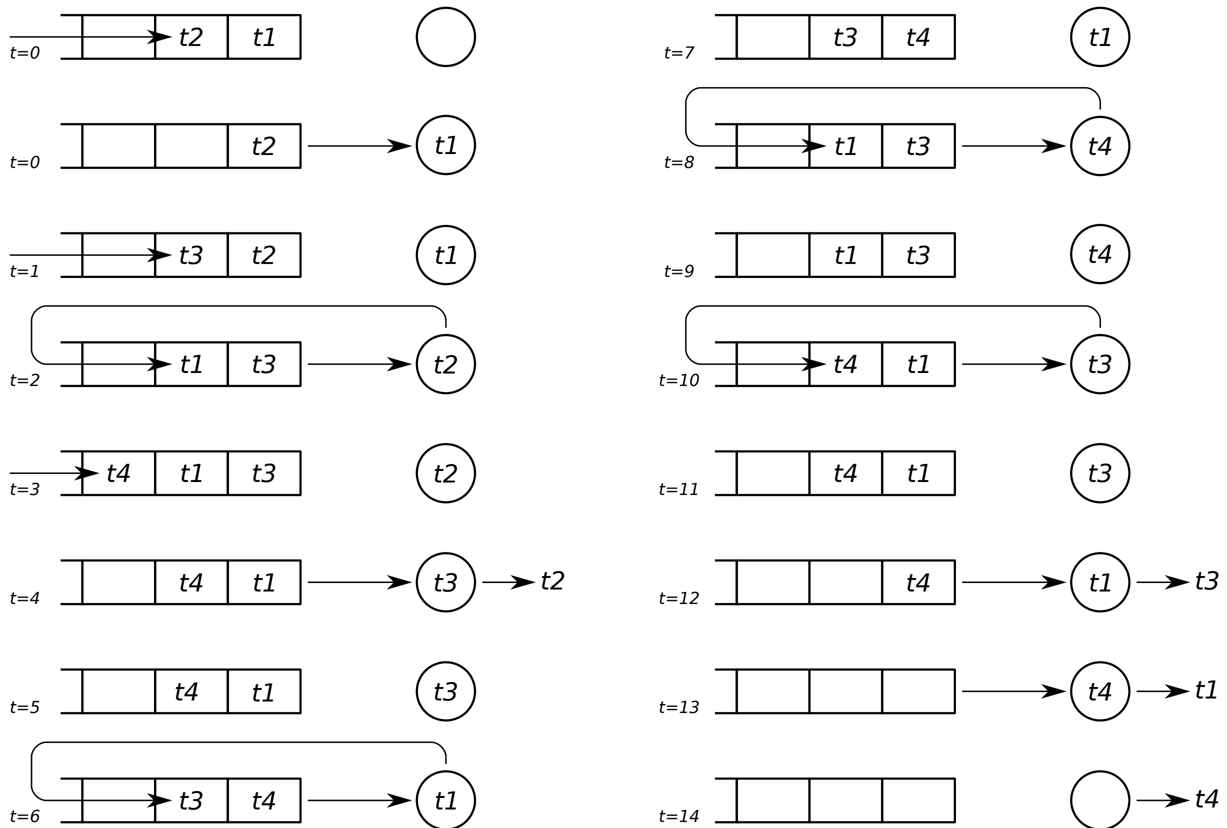

Figura 15: Evolução da fila de tarefas prontas no escalonamento *Round-Robin*.

e menor a duração de cada troca, mais tempo sobrará para a execução das tarefas em si. Assim, é possível definir uma **medida de eficiência** $\mathcal{E}$ do uso do processador, em função das durações médias do *quantum* de tempo t_q e da troca de contexto t_{tc}:

$$\mathcal{E} = \frac{t_q}{t_q + t_{tc}}$$

Por exemplo, um sistema no qual as trocas de contexto duram $1ms$ e cujo *quantum* médio é de $20ms$ terá uma eficiência $\mathcal{E} = \frac{20}{20+1} = 95,2\%$. Caso a duração do quantum seja reduzida para $2ms$, a eficiência cairá para $\mathcal{E} = \frac{2}{2+1} = 66,7\%$. A eficiência final da gerência de tarefas é influenciada por vários fatores, como a carga do sistema (mais tarefas ativas implicam em mais tempo gasto pelo escalonador, aumentando t_{tc}) e o perfil das aplicações (aplicações que fazem muita entrada/saída saem do processador antes do final de seu *quantum*, diminuindo o valor médio de t_q).

Deve-se observar que os algoritmos de escalonamento FCFS e RR não levam em conta a importância das tarefas nem seu comportamento em relação ao uso dos recursos. Por exemplo, nesses algoritmos as tarefas orientadas a entrada/saída irão receber menos tempo de processador que as tarefas orientadas a processamento (pois as primeiras geralmente não usam integralmente seus *quanta* de tempo), o que pode ser prejudicial para aplicações interativas.

5.4 Escalonamento SJF (*Shortest Job First*)

O algoritmo de escalonamento que proporciona os menores tempos médios de execução e de espera é conhecido como *menor tarefa primeiro*, ou SJF (*Shortest Job First*). Como o nome indica, ele consiste em atribuir o processador à menor (mais curta) tarefa da fila de tarefas prontas. Pode ser provado matematicamente que esta estratégia sempre proporciona os menores tempos médios de espera. Aplicando-se este algoritmo às tarefas da tabela anterior, obtém-se o escalonamento apresentado na Figura 16.

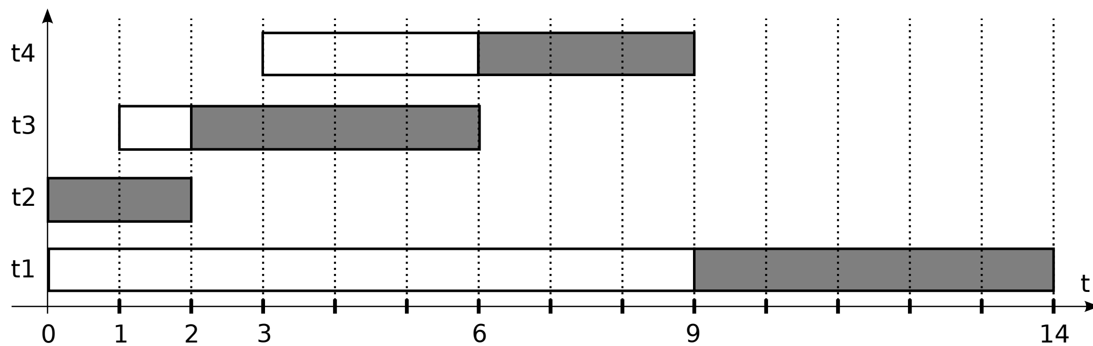

Figura 16: Escalonamento SJF.

Calculando o tempo médio de execução T_t e o tempo médio de espera T_w para o algoritmo SJF, temos:

$$
\begin{aligned}
T_t &= \frac{t_t(t_1) + t_t(t_2) + t_t(t_3) + t_t(t_4)}{4} = \frac{(14-0) + (2-0) + (6-1) + (9-3)}{4} \\
&= \frac{14+2+5+6}{4} = \frac{27}{4} = 6.75s
\end{aligned}
$$

$$
\begin{aligned}
T_w &= \frac{t_w(t_1) + t_w(t_2) + t_w(t_3) + t_w(t_4)}{4} = \frac{(9-0) + (0-0) + (2-1) + (6-3)}{4} \\
&= \frac{9+0+1+3}{4} = \frac{13}{4} = 3.25s
\end{aligned}
$$

Deve-se observar que o comportamento expresso na Figura 16 corresponde à versão cooperativa do algoritmo SJF: o escalonador aguarda a conclusão de cada tarefa para decidir quem irá receber o processador. No caso preemptivo, o escalonador deve comparar a duração prevista de cada nova tarefa que ingressa no sistema com o tempo restante de processamento das demais tarefas presentes, inclusive aquela que está executando no momento. Essa abordagem é denominada por alguns autores de *menor tempo restante primeiro* (SRTF – *Short Remaining Time First*) [Tanenbaum, 2003].

A maior dificuldade no uso do algoritmo SJF consiste em estimar a priori a duração de cada tarefa, ou seja, antes de sua execução. Com exceção de algumas tarefas em lote ou de tempo real, essa estimativa é inviável; por exemplo, como estimar por quanto tempo um editor de textos irá ser utilizado? Por causa desse problema, o algoritmo SJF puro é pouco utilizado. No entanto, ao associarmos o algoritmo SJF à preempção por

tempo, esse algoritmo pode ser de grande valia, sobretudo para tarefas orientadas a entrada/saída.

Suponha uma tarefa orientada a entrada/saída em um sistema preemptivo com $t_q = 10ms$. Nas últimas 3 vezes em que recebeu o processador, essa tarefa utilizou $3ms$, $4ms$ e $4.5ms$ de cada quantum recebido. Com base nesses dados históricos, é possível estimar qual a duração da execução da tarefa na próxima vez em que receber o processador. Essa estimativa pode ser feita por média simples (cálculo mais rápido) ou por extrapolação (cálculo mais complexo, podendo influenciar o tempo de troca de contexto t_{tc}).

A estimativa de uso do próximo quantum assim obtida pode ser usada como base para a aplicação do algoritmo SJF, o que irá priorizar as tarefas orientadas a entrada/saída, que usam menos o processador. Obviamente, uma tarefa pode mudar de comportamento repentinamente, passando de uma fase de entrada/saída para uma fase de processamento, ou vice-versa. Nesse caso, a estimativa de uso do próximo *quantum* será incorreta durante alguns ciclos, mas logo voltará a refletir o comportamento atual da tarefa. Por essa razão, apenas a história recente da tarefa deve ser considerada (3 a 5 últimas ativações).

Outro problema associado ao escalonamento SJF é a possibilidade de *inanição* (*starvation*) das tarefas mais longas. Caso o fluxo de tarefas curtas chegando ao sistema seja elevado, as tarefas mais longas nunca serão escolhidas para receber o processador e vão literalmente "morrer de fome", esperando na fila sem poder executar. Esse problema pode ser resolvido através de técnicas de envelhecimento de tarefas, como a apresentada na Seção 5.5.2.

5.5 Escalonamento por prioridades

Vários critérios podem ser usados para ordenar a fila de tarefas prontas e escolher a próxima tarefa a executar; a data de ingresso da tarefa (usada no FCFS) e sua duração prevista (usada no SJF) são apenas dois deles. Inúmeros outros critérios podem ser especificados, como o comportamento da tarefa (em lote, interativa ou de tempo real), seu proprietário (administrador, gerente, estagiário), seu grau de interatividade, etc.

No escalonamento por prioridades, a cada tarefa é associada uma prioridade, geralmente na forma de um número inteiro. Os valores de prioridade são então usados para escolher a próxima tarefa a receber o processador, a cada troca de contexto. O algoritmo de escalonamento por prioridades define um modelo genérico de escalonamento, que permite modelar várias abordagens, entre as quais o FCFS e o SJF.

Para ilustrar o funcionamento do escalonamento por prioridades, serão usadas as tarefas descritas na tabela a seguir, que usam uma escala de prioridades positiva (ou seja, onde valores maiores indicam uma prioridade maior):

tarefa	t_1	t_2	t_3	t_4
ingresso	0	0	1	3
duração	5	2	4	3
prioridade	2	3	1	4

O diagrama da Figura 17 mostra o escalonamento do processador usando o algoritmo por prioridades em modo cooperativo (ou seja, sem *quantum* ou outras interrupções).

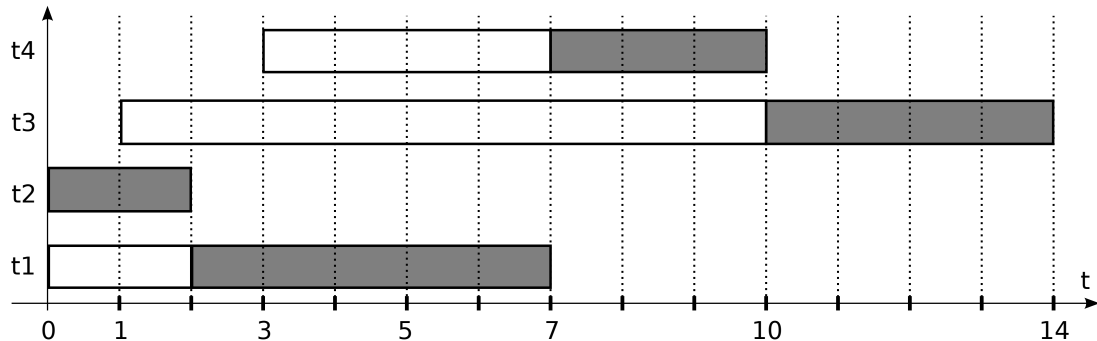

Figura 17: Escalonamento por prioridades (cooperativo).

Calculando o tempo médio de execução T_t e o tempo médio de espera T_w para esse algoritmo, temos:

$$\begin{aligned} T_t &= \frac{t_t(t_1) + t_t(t_2) + t_t(t_3) + t_t(t_4)}{4} = \frac{(7-0) + (2-0) + (14-1) + (10-3)}{4} \\ &= \frac{7+2+13+7}{4} = \frac{29}{4} = 7.25s \end{aligned}$$

$$\begin{aligned} T_w &= \frac{t_w(t_1) + t_w(t_2) + t_w(t_3) + t_w(t_4)}{4} = \frac{(2-0) + (0-0) + (10-1) + (7-3)}{4} \\ &= \frac{2+0+9+4}{4} = \frac{15}{4} = 3.75s \end{aligned}$$

Quando uma tarefa de maior prioridade se torna disponível para execução, o escalonador pode decidir entregar o processador a ela, trazendo a tarefa atual de volta para a fila de prontas. Nesse caso, temos um escalonamento por prioridades *preemptivo*, cujo comportamento é apresentado na Figura 18 (observe que, quando t_4 ingressa no sistema, ela recebe o processador e t_1 volta a esperar na fila de prontas).

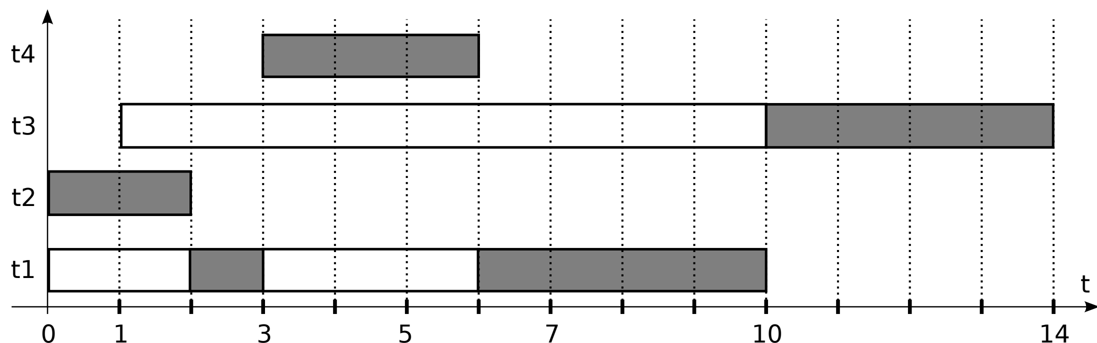

Figura 18: Escalonamento por prioridades (preemptivo).

Calculando o tempo médio de execução T_t e o tempo médio de espera T_w para esse algoritmo, temos:

$$
\begin{aligned}
T_t &= \frac{t_t(t_1) + t_t(t_2) + t_t(t_3) + t_t(t_4)}{4} = \frac{(10-0) + (2-0) + (14-1) + (6-3)}{4} \\
&= \frac{10 + 2 + 13 + 3}{4} = \frac{28}{4} = 7s \\
T_w &= \frac{t_w(t_1) + t_w(t_2) + t_w(t_3) + t_w(t_4)}{4} = \frac{5 + 0 + 9 + 0}{4} = \frac{14}{4} = 3.5s
\end{aligned}
$$

5.5.1 Definição de prioridades

A definição da prioridade de uma tarefa é influenciada por diversos fatores, que podem ser classificados em dois grandes grupos:

Fatores externos : são informações providas pelo usuário ou o administrador do sistema, que o escalonador não conseguiria estimar sozinho. Os fatores externos mais comuns são a classe do usuário (administrador, diretor, estagiário) o valor pago pelo uso do sistema (serviço básico, serviço *premium*) e a importância da tarefa em si (um detector de intrusão, um *script* de reconfiguração emergencial, etc.).

Fatores internos : são informações que podem ser obtidas ou estimadas pelo escalonador, com base em dados disponíveis no sistema local. Os fatores internos mais utilizados são a idade da tarefa, sua duração estimada, sua interatividade, seu uso de memória ou de outros recursos, etc.

Todos esses fatores devem ser combinados para produzir um valor de prioridade para cada tarefa. Todos os fatores externos são expressos por valor inteiro denominado **prioridade estática** (ou *prioridade de base*), que resume a "opinião" do usuário ou administrador sobre aquela tarefa. Os fatores internos mudam continuamente e devem ser recalculados periodicamente pelo escalonador. A combinação da prioridade estática com os fatores internos resulta na **prioridade dinâmica** ou final, que é usada pelo escalonador para ordenar as tarefas prontas. A Figura 19 resume esse procedimento.

Em geral, cada família de sistemas operacionais define sua própria escala de prioridades estáticas. Alguns exemplos de escalas comuns são:

Windows 2000 e sucessores : processos e *threads* são associados a *classes de prioridade* (6 classes para processos e 7 classes para *threads*); a prioridade final de uma *thread* depende de sua prioridade de sua própria classe de prioridade e da classe de prioridade do processo ao qual está associada, assumindo valores entre 0 e 31. As prioridades do processos, apresentadas aos usuários no *Gerenciador de Tarefas*, apresentam os seguintes valores *default*:

- 4: *baixa* ou *ociosa*
- 6: *abaixo do normal*
- 8: *normal*

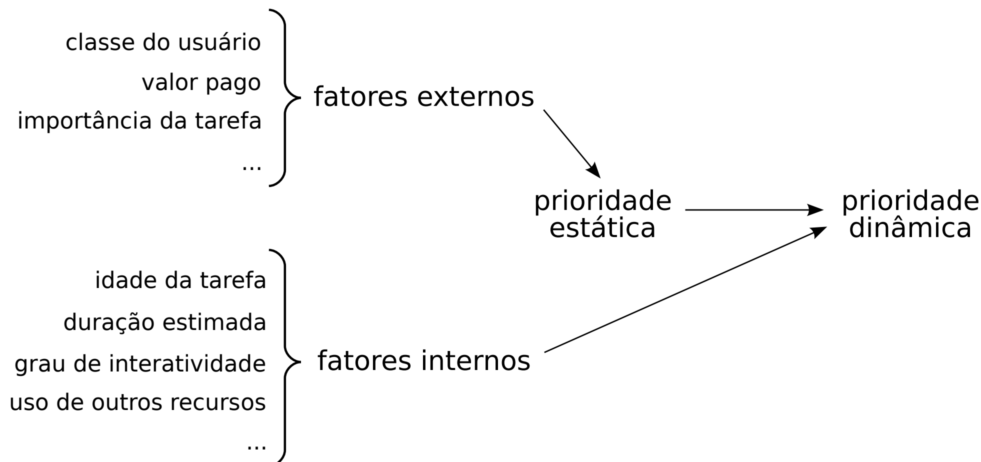

Figura 19: Composição da prioridade dinâmica.

- 10: *acima do normal*
- 13: *alta*
- 24: *tempo real*

Geralmente a prioridade da tarefa responsável pela janela ativa recebe um incremento de prioridade (+1 ou +2, conforme a configuração do sistema).

No Linux (núcleo 2.4 e sucessores) há duas escalas de prioridades:

- *Tarefas interativas*: a escala de prioridades é negativa: a prioridade de cada tarefa vai de -20 (mais importante) a +19 (menos importante) e pode ser ajustada através dos comandos `nice` e `renice`. Esta escala é padronizada em todos os sistemas UNIX.
- *Tarefas de tempo real*: a prioridade de cada tarefa vai de 1 (mais importante) a 99 (menos importante). As tarefas de tempo real têm precedência sobre as tarefas interativas e são escalonadas usando políticas distintas. Somente o administrador pode criar tarefas de tempo real.

5.5.2 Inanição e envelhecimento de tarefas

No escalonamento por prioridades básico, as tarefas de baixa prioridade só recebem o processador na ausência de tarefas de maior prioridade. Caso existam tarefas de maior prioridade frequentemente ativas, as de baixa prioridade podem sofrer de inanição (*starvation*), ou seja, nunca ter acesso ao processador.

Além disso, em sistemas de tempo compartilhado, as prioridades estáticas definidas pelo usuário estão intuitivamente relacionadas à *proporcionalidade* na divisão do tempo de processamento. Por exemplo, se um sistema recebe duas tarefas iguais com a mesma prioridade, espera-se que cada uma receba 50% do processador e que ambas concluam ao mesmo tempo. Caso o sistema receba três tarefas: t_1 com prioridade 1, t_2 com prioridade 2 e t_3 com prioridade 3, espera-se que t_3 receba mais o processador que t_2, e esta mais

que t_1 (assumindo uma escala de prioridades positiva). Entretanto, se aplicarmos o algoritmo de prioridades básico, as tarefas irão executar de forma sequencial, sem distribuição proporcional do processador. Esse resultado indesejável ocorre porque, a cada fim de quantum, sempre a mesma tarefa é escolhida para processar: a mais prioritária. Essa situação está ilustrada na Figura 20.

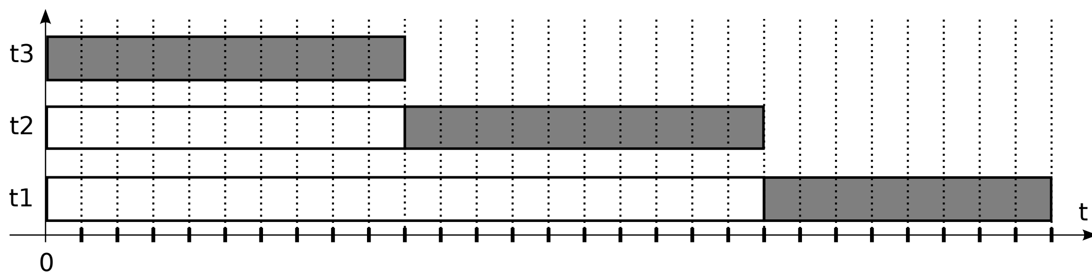

Figura 20: Escalonamento por prioridades.

Para evitar a inanição e garantir a proporcionalidade expressa através das prioridades estáticas, um fator interno denominado **envelhecimento** (*task aging*) deve ser definido. O envelhecimento indica há quanto tempo uma tarefa está aguardando o processador e aumenta sua prioridade proporcionalmente. Dessa forma, o envelhecimento evita a inanição dos processos de baixa prioridade, permitindo a eles obter o processador periodicamente. Uma forma simples de implementar o envelhecimento está resumida no seguinte algoritmo (que considera uma escala de prioridades positiva):

Definições:
- t_i : tarefa i
- pe_i : prioridade estática de t_i
- pd_i : prioridade dinâmica de t_i
- N : número de tarefas no sistema

Quando uma tarefa nova t_n ingressa no sistema:
- $pe_n \leftarrow$ *prioridade inicial default*
- $pd_n \leftarrow pe_n$

Para escolher a próxima tarefa a executar t_p:
- escolher $t_p \mid pd_p = max_{i=1}^{N}(pd_i)$
- $pd_p \leftarrow pe_p$
- $\forall i \neq p : pd_i \leftarrow pd_i + \alpha$

Em outras palavras, a cada turno o escalonador escolhe como próxima tarefa (t_p) aquela com a maior prioridade dinâmica (pd_p). A prioridade dinâmica dessa tarefa é igualada à sua prioridade estática ($pd_p \leftarrow pe_p$) e então ela recebe o processador. A prioridade dinâmica das demais tarefas é aumentada de α, ou seja, elas "envelhecem" e no próximo turno terão mais chances de ser escolhidas. A constante α é conhecida como *fator de envelhecimento*.

Usando o algoritmo de envelhecimento, a divisão do processador entre as tarefas se torna proporcional às suas prioridades. A Figura 21 ilustra essa proporcionalidade na execução das três tarefas t_1, t_2 e t_3 com $p(t_1) < p(t_2) < p(t_3)$, usando a estratégia de envelhecimento. Nessa figura, percebe-se que todas as três tarefas recebem o processador periodicamente, mas que t_3 recebe mais tempo de processador que t_2, e que t_2 recebe mais que t_1.

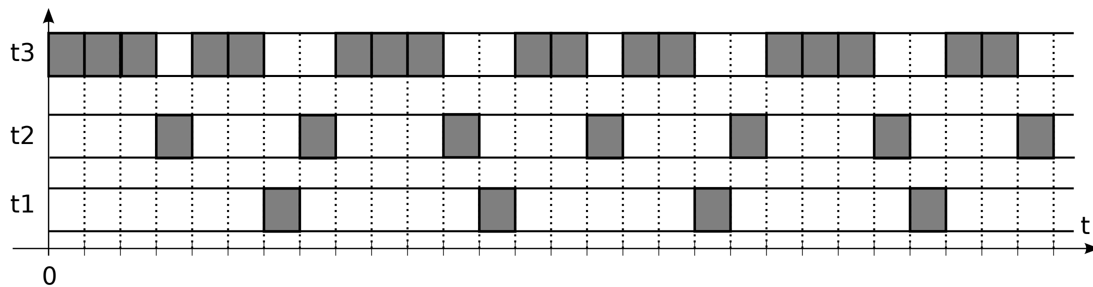

Figura 21: Escalonamento por prioridades com envelhecimento.

5.5.3 Inversão e herança de prioridades

Outro problema relevante que pode ocorrer em sistemas baseados em prioridades é a *inversão de prioridades* [Sha et al., 1990]. Este tipo de problema é mais complexo que o anterior, pois envolve o conceito de *exclusão mútua*: alguns recursos do sistema devem ser usados por um processo de cada vez, para evitar problemas de consistência de seu estado interno. Isso pode ocorrer com arquivos, portas de entrada saída e conexões de rede, por exemplo. Quando um processo obtém acesso a um recurso com exclusão mútua, os demais processos que desejam usá-lo ficam esperando no estado suspenso, até que o recurso esteja novamente livre. As técnicas usadas para implementar a exclusão mútua são descritas no Capítulo **??**.

A inversão de prioridades consiste em processos de alta prioridade serem impedidos de executar por causa de um processo de baixa prioridade. Para ilustrar esse problema, pode ser considerada a seguinte situação: um determinado sistema possui um processo de alta prioridade p_a, um processo de baixa prioridade p_b e alguns processos de prioridade média p_m. Além disso, há um recurso R que deve ser acessado em exclusão mútua; para simplificar, somente p_a e p_b estão interessados em usar esse recurso. A seguinte sequência de eventos, ilustrada na Figura 22, é um exemplo de como pode ocorrer uma inversão de prioridades:

1. Em um dado momento, o processador está livre e é alocado a um processo de baixa prioridade p_b;

2. durante seu processamento, p_b obtém o acesso exclusivo a um recurso R e começa a usá-lo;

3. p_b perde o processador, pois um processo com prioridade maior que a dele (p_m) foi acordado devido a uma interrupção;

4. p_b volta ao final da fila de tarefas prontas, aguardando o processador; enquanto ele não voltar a executar, o recurso R permanecerá alocado a ele e ninguém poderá usá-lo;

5. Um processo de alta prioridade p_a recebe o processador e solicita acesso ao recurso R; como o recurso está alocado ao processo p_b, p_a é suspenso até que o processo de baixa prioridade p_b libere o recurso.

Neste momento, o processo de alta prioridade p_a não pode continuar sua execução, porque o recurso de que necessita está nas mãos do processo de baixa prioridade p_b. Dessa forma, p_a deve esperar que p_b execute e libere R, o que justifica o nome *inversão de prioridades*. A espera de p_a pode ser longa, pois p_b tem baixa prioridade e pode demorar a receber o processador novamente, caso existam outros processos em execução no sistema (como p_m). Como tarefas de alta prioridade são geralmente críticas para o funcionamento de um sistema, a inversão de prioridades pode ter efeitos graves.

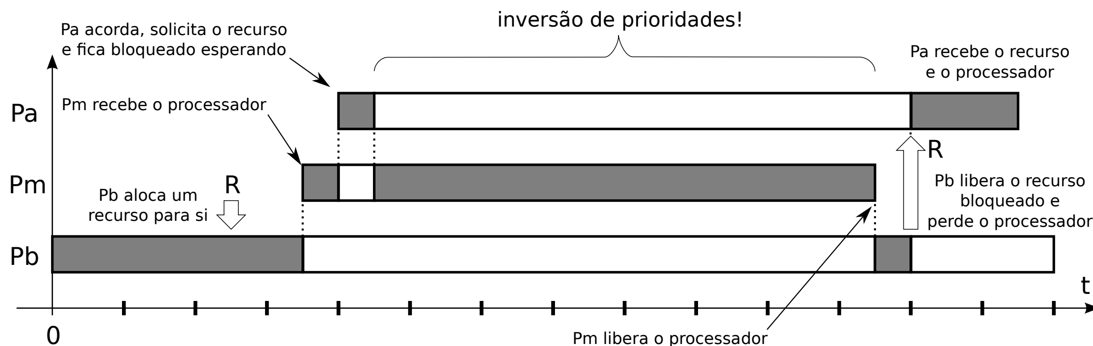

Figura 22: Cenário de uma inversão de prioridades.

Uma solução elegante para o problema da inversão de prioridades é obtida através de um *protocolo de herança de prioridade* [Sha et al., 1990]. O protocolo de herança de prioridade mais simples consiste em aumentar temporariamente a prioridade do processo p_b que detém o recurso de uso exclusivo R. Caso esse recurso seja requisitado por um processo de maior prioridade p_a, o processo p_b "herda" temporariamente a prioridade de p_a, para que possa voltar a executar e liberar o recurso R mais rapidamente. Assim que liberar o recurso, p_b retorna à sua prioridade anterior. Essa estratégia está ilustrada na Figura 23.

Provavelmente o melhor exemplo real de inversão de prioridades tenha ocorrido na sonda espacial *Mars Pathfinder*, enviada pela NASA em 1996 para explorar o solo marciano (Figura 24) [Jones, 1997]. O software da sonda executava sobre o sistema operacional de tempo real *VxWorks* e consistia de 97 *threads* com vários níveis de prioridades. Essas tarefas se comunicavam através de uma área de transferência em memória compartilhada, com acesso mutuamente exclusivo controlado por semáforos (semáforos são estruturas de sincronização discutidas na Seção **??**).

A gerência da área de transferência estava a cargo de uma tarefa t_g, rápida mas de alta prioridade, que era ativada frequentemente para mover blocos de informação para dentro e fora dessa área. A coleta de dados meteorológicos era feita por uma tarefa t_m

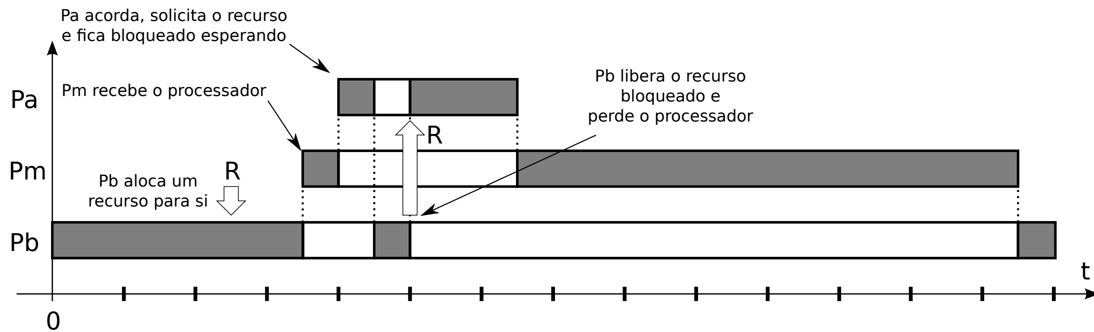

Figura 23: Um protocolo de herança de prioridade.

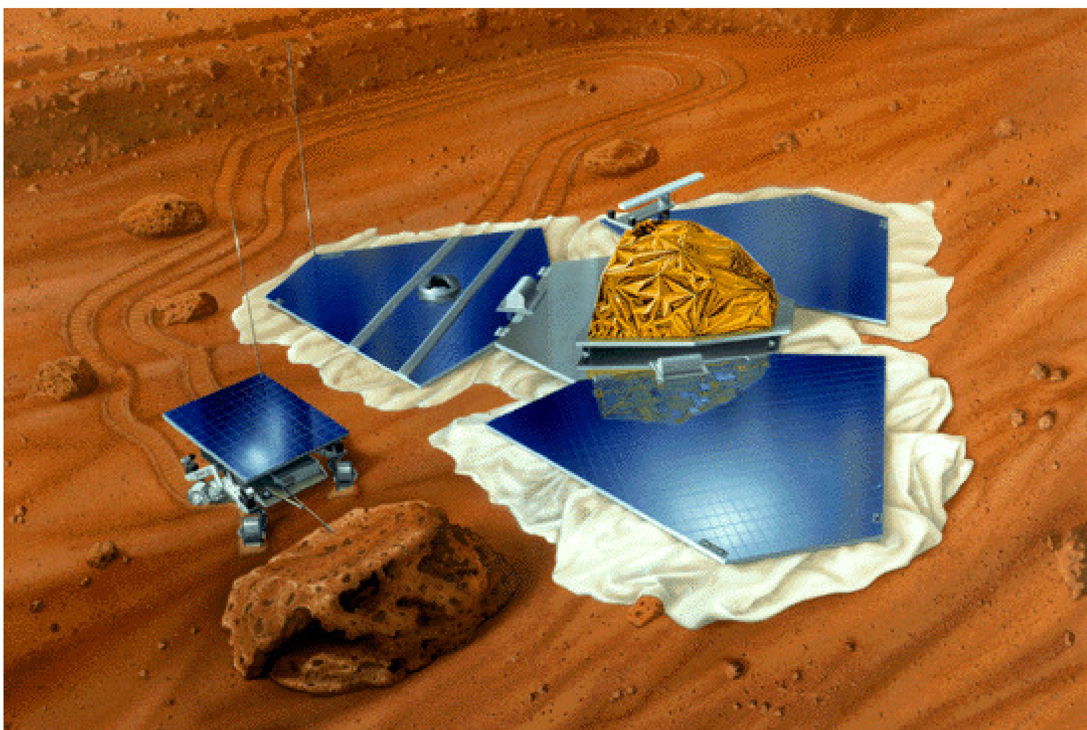

Figura 24: Sonda *Mars Pathfinder* com o robô *Sojourner* (NASA).

de baixa prioridade, que executava esporadicamente e escrevia seus dados na área de transferência, para uso por outras tarefas. Por fim, a comunicação com a Terra estava sob a responsabilidade de uma tarefa t_c de prioridade média e potencialmente demorada (Tabela 2 e Figura 25).

tarefa	**função**	**prioridade**	**duração**
t_g	gerência da área de transferência	alta	curta
t_m	coleta de dados meteorológicos	baixa	curta
t_c	comunicação com a Terra	média	longa

Tabela 2: Algumas tarefas do software da sonda *Mars Pathfinder*.

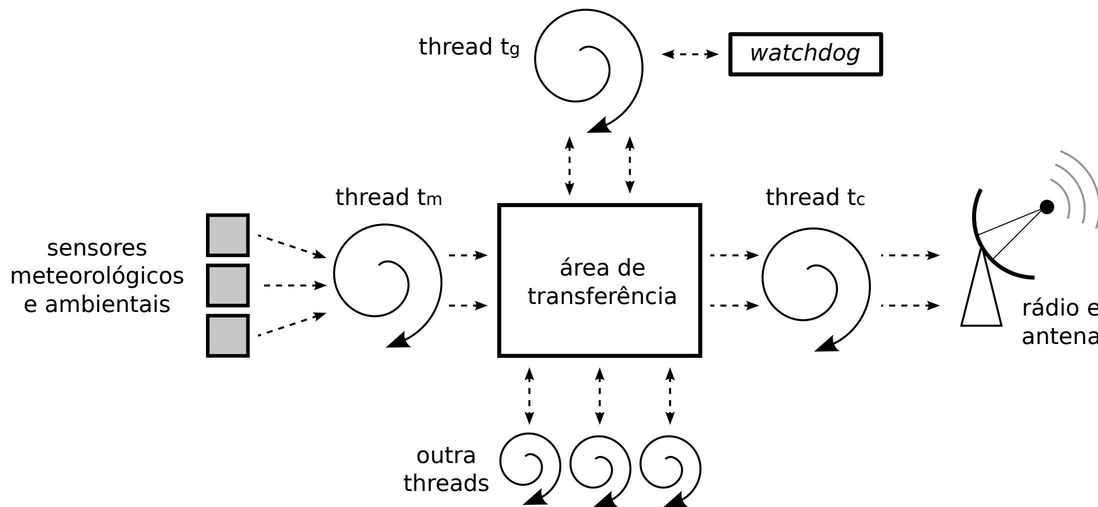

Figura 25: Principais tarefas do software embarcado da sonda *Mars Pathfinder*.

Como o sistema *VxWorks* define prioridades preemptivas, as tarefas eram atendidas conforme suas necessidades na maior parte do tempo. Todavia, a exclusão mútua no acesso à área de transferência escondia uma inversão de prioridades: caso a tarefa de coleta de dados meteorológicos t_m perdesse o processador sem liberar a área de transferência, a tarefa de gerência t_g teria de ficar esperando até que t_m voltasse a executar para liberar a área. Isso poderia demorar se, por azar, a tarefa de comunicação estivesse executando, pois ela tinha mais prioridade que t_m.

Como todos os sistemas críticos, a sonda *Mars Pathfinder* possui um sistema de proteção contra erros, ativado por um temporizador (*watchdog*). Caso a gerência da área de transferência ficasse parada por muito tempo, um procedimento de reinício geral do sistema era automaticamente ativado pelo temporizador. Dessa forma, a inversão de prioridades provocava reinícios esporádicos e imprevisíveis no software da sonda, interrompendo suas atividades e prejudicando seu funcionamento. A solução foi obtida através da herança de prioridades: caso a tarefa de gerência t_g fosse bloqueada pela tarefa de coleta de dados t_m, esta última herdava a alta prioridade de t_g para poder liberar rapidamente a área de transferência, mesmo se a tarefa de comunicação t_c estivesse em execução.

5.6 Outros algoritmos de escalonamento

Além dos algoritmos de escalonamento vistos nesta seção, diversos outros podem ser encontrados na literatura e em sistemas de mercado, como os escalonadores de tempo real [Farines et al., 2000], os escalonadores multimídia [Nieh and Lam, 1997], os escalonadores justos [Kay and Lauder, 1988, Ford and Susarla, 1996], os escalonadores multiprocessador [Black, 1990] e multicore [Boyd-Wickizer et al., 2009].

5.7 Um escalonador real

Na prática, os sistemas operacionais de mercado implementam mais de um algoritmo de escalonamento. A escolha do escalonador adequado é feita com base na *classe de escalonamento* atribuída a cada tarefa. Por exemplo, o núcleo Linux implementa dois escalonadores (Figura 26): um escalonador de tarefas de tempo real (classes `SCHED_FIFO` e `SCHED_RR`) e um escalonador de tarefas interativas (classe `SCHED_OTHER`) [Love, 2004]. Cada uma dessas classes de escalonamento está explicada a seguir:

Classe `SCHED_FIFO` : as tarefas associadas a esta classe são escalonadas usando uma política FCFS sem preempção (sem *quantum*) e usando apenas suas prioridades estáticas (não há envelhecimento). Portanto, uma tarefa desta classe executa até bloquear por recursos ou liberar explicitamente o processador (através da chamada de sistema `sched_yield()`).

Classe `SCHED_RR` : implementa uma política similar à anterior, com a inclusão da preempção por tempo. O valor do *quantum* é proporcional à prioridade atual de cada tarefa, variando de 10*ms* a 200*ms*.

Classe `SCHED_OTHER` : suporta tarefas interativas em lote, através de uma política baseada em prioridades dinâmicas com preempção por tempo com *quantum* variável. Tarefas desta classe somente são escalonadas se não houverem tarefas prontas nas classes `SCHED_FIFO` e `SCHED_RR`.

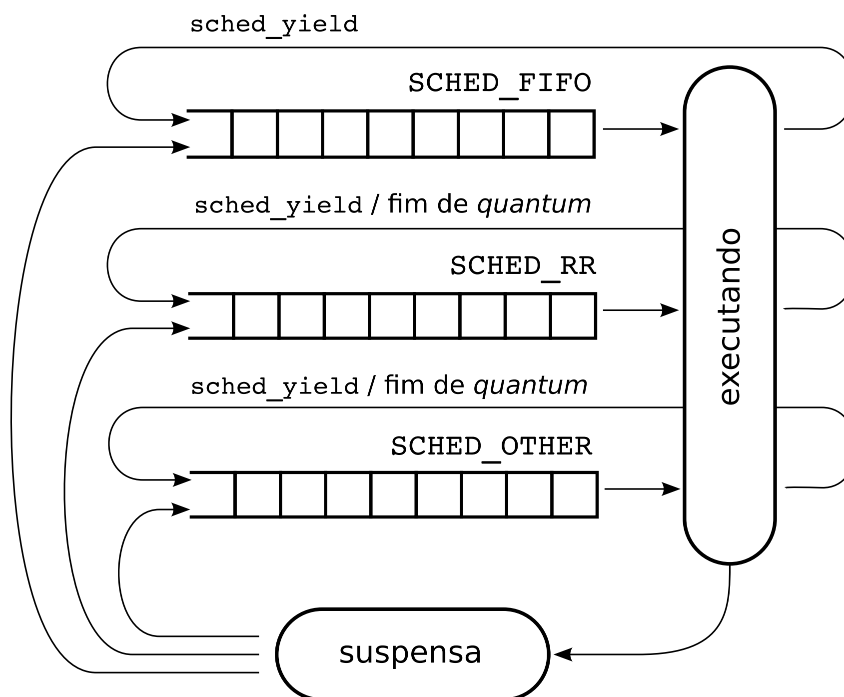

Figura 26: O escalonador multifilas do Linux.

As classes de escalonamento `SCHED_FIFO` e `SCHED_RR` são reservadas para tarefas de tempo real, que só podem ser lançadas pelo administrador do sistema. Todas as demais

tarefas, ou seja, a grande maioria das aplicações e comandos dos usuários, executa na classe de escalonamento SCHED_OTHER.

Referências

[Anderson et al., 2002] Anderson, D., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. (2002). SETI@home: An experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61.

[Anderson et al., 1992] Anderson, T., Bershad, B., Lazowska, E., and Levy, H. (1992). Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79.

[Barney, 2005] Barney, B. (2005). POSIX threads programming. http://www.llnl.gov/computing/tutorials/pthreads.

[Black, 1990] Black, D. L. (1990). Scheduling and resource management techniques for multiprocessors. Technical Report CMU-CS-90-152, Carnegie-Mellon University, Computer Science Dept.

[Boyd-Wickizer et al., 2009] Boyd-Wickizer, S., Morris, R., and Kaashoek, M. (2009). Reinventing scheduling for multicore systems. In *12th conference on Hot topics in operating systems*, page 21. USENIX Association.

[Burns and Wellings, 1997] Burns, A. and Wellings, A. (1997). *Real-Time Systems and Programming Languages, 2nd edition*. Addison-Wesley.

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

[Engeschall, 2005] Engeschall, R. (2005). The GNU Portable Threads. http://www.gnu.org/software/pth.

[Evans and Elischer, 2003] Evans, J. and Elischer, J. (2003). Kernel-scheduled entities for FreeBSD. http://www.aims.net.au/chris/kse.

[Farines et al., 2000] Farines, J.-M., da Silva Fraga, J., and de Oliveira, R. S. (2000). *Sistemas de Tempo Real – 12ª Escola de Computação da SBC*. Sociedade Brasileira de Computação.

[Ford and Susarla, 1996] Ford, B. and Susarla, S. (1996). CPU Inheritance Scheduling. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105.

[Jones, 1997] Jones, M. (1997). What really happened on Mars Rover Pathfinder. *ACM Risks-Forum Digest*, 19(49).

[Kay and Lauder, 1988] Kay, J. and Lauder, P. (1988). A fair share scheduler. *Communications of the ACM*, 31(1):44–55.

[Love, 2004] Love, R. (2004). *Linux Kernel Development*. Sams Publishing Developer's Library.

[Nichols et al., 1996] Nichols, B., Buttlar, D., and Farrell, J. (1996). *PThreads Programming*. O'Reilly Media, Inc.

[Nieh and Lam, 1997] Nieh, J. and Lam, M. (1997). The design, implementation and evaluation of SMART: a scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 184–197.

[Sha et al., 1990] Sha, L., Rajkumar, R., and Lehoczky, J. (1990). Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185.

[Silberschatz et al., 2001] Silberschatz, A., Galvin, P., and Gagne, G. (2001). *Sistemas Operacionais – Conceitos e Aplicações*. Campus.

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

A O *Task Control Block* do Linux

A estrutura em linguagem C apresentada a seguir constitui o descritor de tarefas (*Task Control Block*) do Linux (estudado na Seção 4.1). Ela foi extraída do arquivo `include/linux/sched.h` do código fonte do núcleo Linux 2.6.12 (o arquivo inteiro contém mais de 1.200 linhas de código em C).

```
1  struct task_struct {
2        volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
3        struct thread_info *thread_info;
4        atomic_t usage;
5        unsigned long flags; /* per process flags, defined below */
6        unsigned long ptrace;
7
8        int lock_depth;      /* BKL lock depth */
9
10       int prio, static_prio;
11       struct list_head run_list;
12       prio_array_t *array;
13
14       unsigned long sleep_avg;
15       unsigned long long timestamp, last_ran;
16       unsigned long long sched_time; /* sched_clock time spent running */
17       int activated;
18
19       unsigned long policy;
20       cpumask_t cpus_allowed;
21       unsigned int time_slice, first_time_slice;
22
23 #ifdef CONFIG_SCHEDSTATS
24       struct sched_info sched_info;
```

```
25 #endif
26
27        struct list_head tasks;
28        /*
29         * ptrace_list/ptrace_children forms the list of my children
30         * that were stolen by a ptracer.
31         */
32        struct list_head ptrace_children;
33        struct list_head ptrace_list;
34
35        struct mm_struct *mm, *active_mm;
36
37 /* task state */
38        struct linux_binfmt *binfmt;
39        long exit_state;
40        int exit_code, exit_signal;
41        int pdeath_signal; /* The signal sent when the parent dies */
42        /* ??? */
43        unsigned long personality;
44        unsigned did_exec:1;
45        pid_t pid;
46        pid_t tgid;
47        /*
48         * pointers to (original) parent process, youngest child, younger sibling,
49         * older sibling, respectively. (p->father can be replaced with
50         * p->parent->pid)
51         */
52        struct task_struct *real_parent; /* real parent process (when being debugged) */
53        struct task_struct *parent; /* parent process */
54        /*
55         * children/sibling forms the list of my children plus the
56         * tasks I'm ptracing.
57         */
58        struct list_head children; /* list of my children */
59        struct list_head sibling;  /* linkage in my parent's children list */
60        struct task_struct *group_leader; /* threadgroup leader */
61
62        /* PID/PID hash table linkage. */
63        struct pid pids[PIDTYPE_MAX];
64
65        struct completion *vfork_done;     /* for vfork() */
66        int __user *set_child_tid;         /* CLONE_CHILD_SETTID */
67        int __user *clear_child_tid;       /* CLONE_CHILD_CLEARTID */
68
69        unsigned long rt_priority;
70        cputime_t utime, stime;
71        unsigned long nvcsw, nivcsw; /* context switch counts */
72        struct timespec start_time;
73 /* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
74        unsigned long min_flt, maj_flt;
75
76        cputime_t it_prof_expires, it_virt_expires;
77        unsigned long long it_sched_expires;
78        struct list_head cpu_timers[3];
79
```

```
80  /* process credentials */
81          uid_t uid,euid,suid,fsuid;
82          gid_t gid,egid,sgid,fsgid;
83          struct group_info *group_info;
84          kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
85          unsigned keep_capabilities:1;
86          struct user_struct *user;
87  #ifdef CONFIG_KEYS
88          struct key *thread_keyring; /* keyring private to this thread */
89  #endif
90          int oomkilladj; /* OOM kill score adjustment (bit shift). */
91          char comm[TASK_COMM_LEN]; /* executable name excluding path
92                                       - access with [gs]et_task_comm (which lock
93                                         it with task_lock())
94                                       - initialized normally by flush_old_exec */
95  /* file system info */
96          int link_count, total_link_count;
97  /* ipc stuff */
98          struct sysv_sem sysvsem;
99  /* CPU-specific state of this task */
100         struct thread_struct thread;
101 /* filesystem information */
102         struct fs_struct *fs;
103 /* open file information */
104         struct files_struct *files;
105 /* namespace */
106         struct namespace *namespace;
107 /* signal handlers */
108         struct signal_struct *signal;
109         struct sighand_struct *sighand;
110
111         sigset_t blocked, real_blocked;
112         struct sigpending pending;
113
114         unsigned long sas_ss_sp;
115         size_t sas_ss_size;
116         int (*notifier)(void *priv);
117         void *notifier_data;
118         sigset_t *notifier_mask;
119
120         void *security;
121         struct audit_context *audit_context;
122         seccomp_t seccomp;
123
124 /* Thread group tracking */
125         u32 parent_exec_id;
126         u32 self_exec_id;
127 /* Protection of (de-)allocation: mm, files, fs, tty, keyrings */
128         spinlock_t alloc_lock;
129 /* Protection of proc_dentry: nesting proc_lock, dcache_lock, write_lock_irq(&tasklist_lock); */
130         spinlock_t proc_lock;
131 /* context-switch lock */
132         spinlock_t switch_lock;
133
134 /* journalling filesystem info */
```

```
135        void *journal_info;
136
137 /* VM state */
138        struct reclaim_state *reclaim_state;
139
140        struct dentry *proc_dentry;
141        struct backing_dev_info *backing_dev_info;
142
143        struct io_context *io_context;
144
145        unsigned long ptrace_message;
146        siginfo_t *last_siginfo; /* For ptrace use. */
147 /*
148  * current io wait handle: wait queue entry to use for io waits
149  * If this thread is processing aio, this points at the waitqueue
150  * inside the currently handled kiocb. It may be NULL (i.e. default
151  * to a stack based synchronous wait) if its doing sync IO.
152  */
153        wait_queue_t *io_wait;
154 /* i/o counters(bytes read/written, #syscalls */
155        u64 rchar, wchar, syscr, syscw;
156 #if defined(CONFIG_BSD_PROCESS_ACCT)
157        u64 acct_rss_mem1;   /* accumulated rss usage */
158        u64 acct_vm_mem1;    /* accumulated virtual memory usage */
159        clock_t acct_stimexpd; /* clock_t-converted stime since last update */
160 #endif
161 #ifdef CONFIG_NUMA
162        struct mempolicy *mempolicy;
163        short il_next;
164 #endif
165 #ifdef CONFIG_CPUSETS
166        struct cpuset *cpuset;
167        nodemask_t mems_allowed;
168        int cpuset_mems_generation;
169 #endif
170 };
```