Capítulo 6

Escalonamento de tarefas

Um dos componentes mais importantes da gerência de tarefas é o **escalonador de tarefas** (*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.

6.1 Tipos de tarefas

Antes de se definir o algoritmo usado por um escalonador de tarefas, 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, tratamentos de grandes massas de dados em lote, 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 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 ficam 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.

6.2 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 5.2. 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 execução de uma 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 e aos dispositivos).

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, em um editor de textos seria o tempo decorrido entre apertar uma tecla e o caractere correspondente aparecer na tela. 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 interativas 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 e prioridade similares devem ter durações de execução similares.

Eficiência: a eficiência $\mathcal{E}$, conforme definido na Seção 5.2, 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).

6.3 Escalonamento preemptivo e cooperativo

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

Sistemas preemptivos: nestes sistemas uma tarefa pode perder o processador caso termine seu *quantum* de tempo, caso 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 reavalia todas as tarefas da fila de prontas e decide se mantém ou substitui a tarefa atualmente em execução.

Sistemas cooperativos: a tarefa em execução permanece no processador tanto quanto possível, só liberando o mesmo caso termine de executar, solicite uma operação de entrada/saída ou libere explicitamente o processador[1], voltando à fila de tarefas prontas. 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 é preemptiva. Sistemas mais antigos, como o Windows 3.*, PalmOS 3 e MacOS 8 e 9 operavam de forma cooperativa.

Em um sistema preemptivo simples, 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.

6.4 Algoritmos de escalonamento de tarefas

Esta seção descreve os algoritmos mais simples de escalonamento de tarefas encontrados na literatura. Esses algoritmos raramente são usados *ipsis litteris*, mas

[1] Uma tarefa pode liberar explicitamente o processador e voltar à fila de prontas através de uma chamada de sistema específica, como `sched_yield()` em Linux ou `SwitchToThread()` em Windows.

servem de base conceitual para a construção dos escalonadores mais complexos que são usados em sistemas operacionais reais. Para a descrição do funcionamento de cada algoritmo será considerado um sistema monoprocessado e um conjunto hipotético de 4 tarefas ($t_1 \cdots t_5$) na fila de prontas do sistema operacional, descritas na Tabela 6.1 a seguir.

Tarefa	t_1	t_2	t_3	t_4	t_5
Ingresso	0	0	1	3	5
Duração	5	2	4	1	2
Prioridade	2	3	1	4	5

Tabela 6.1: Tarefas na fila de prontas.

Para simplificar a análise dos algoritmos, as tarefas $t_1 \cdots t_5$ são orientadas a processamento, ou seja, não param para realizar operações de entrada/saída. Cada tarefa tem uma data de ingresso (instante em que entrou no sistema), uma duração (tempo de processamento que necessita para realizar sua execução) e uma prioridade (usada nos algoritmos PRIOc e PRIOp, seção 6.4.5).

6.4.1 First-Come, First Served (FCFS)

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 ingresso na fila de tarefas prontas). Esse algoritmo é conhecido como FCFS – *First Come - First Served* – e tem como principal vantagem sua simplicidade.

O diagrama da Figura 6.1 mostra o escalonamento das tarefas de Tabela 6.1 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).

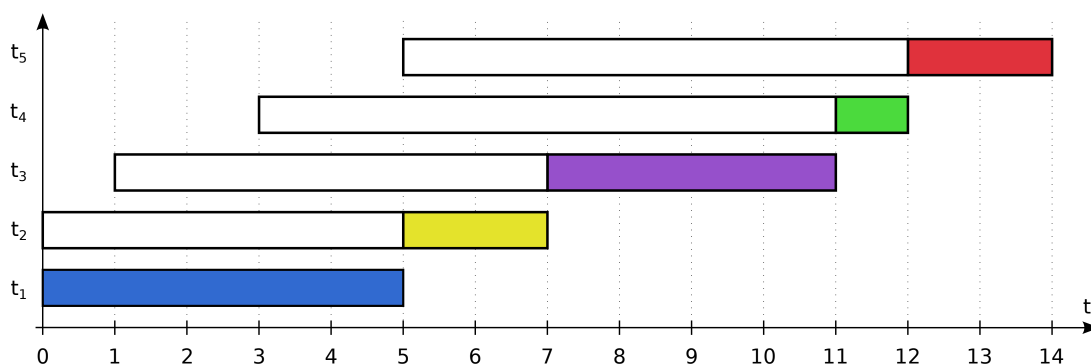

Figura 6.1: Escalonamento FCFS.

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 a execução da Figura 6.1, temos:

$$T_t = \frac{t_t(t_1) + \cdots + t_t(t_5)}{5} = \frac{(5-0) + (7-0) + (11-1) + (12-3) + (14-5)}{5}$$

$$= \frac{5+7+10+9+9}{5} = \frac{40}{5} = 8,0s$$

$$T_w = \frac{t_w(t_1) + \cdots + t_w(t_5)}{5} = \frac{(0-0)+(5-0)+(7-1)+(11-3)+(12-5)}{5}$$

$$= \frac{0+5+6+8+7}{5} = \frac{26}{5} = 5,2s$$

6.4.2 Round-Robin (RR)

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 6.2.

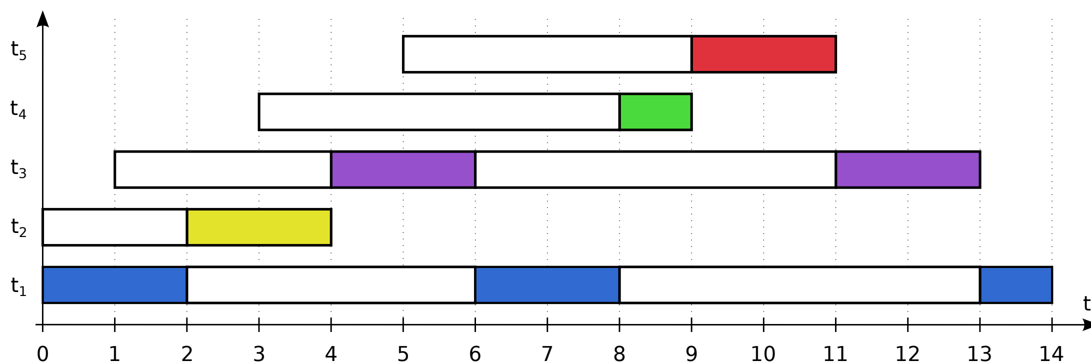

Figura 6.2: Escalonamento *Round-Robin*.

Na Figura 6.2, é importante observar que a execução das tarefas não obedece uma sequência óbvia como $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_4 \rightarrow t_5 \rightarrow t_1 \rightarrow t_2 \rightarrow \ldots$, mas uma sequência bem mais complexa: $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow t_1 \rightarrow t_4 \rightarrow t_5 \rightarrow \ldots$. 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 6.3 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 a execução da Figura 6.2, temos:

$$T_t = \frac{t_t(t_1) + \cdots + t_t(t_5)}{5} = \frac{14+4+12+6+6}{5} = \frac{42}{5} = 8,4s$$

$$T_w = \frac{t_w(t_1) + \cdots + t_w(t_5)}{5} = \frac{9+2+8+5+4}{5} = \frac{28}{5} = 5,6s$$

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 proporciona tempos de resposta bem melhores às aplicações interativas. Outro problema deste escalonador é o aumento no número de trocas de contexto, que,

Figura 6.3: Evolução da fila de tarefas prontas no escalonamento *Round-Robin*.

se for significativo, pode levar a uma degradação de desempenho, conforme discutido na Seção 5.2.

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.

6.4.3 Shortest Job First (SJF)

O algoritmo de escalonamento conhecido como *menor tarefa primeiro* (SJF - *Shortest Job First*) consiste em atribuir o processador à menor (mais curta) tarefa da fila de tarefas prontas. Esse algoritmo (e sua versão preemptiva, SRTF) proporciona os menores tempos médios de espera das tarefas. Aplicando-se este algoritmo às tarefas da Tabela 6.1, obtém-se o escalonamento apresentado na Figura 6.4.

Calculando o tempo médio de execução T_t e o tempo médio de espera T_w para a execução da Figura 6.4, temos:

$$T_t = \frac{t_t(t_1) + \cdots + t_t(t_5)}{5} = \frac{14+2+5+4+4}{5} = \frac{29}{5} = 5,8s$$

$$T_w = \frac{t_w(t_1) + \cdots + t_w(t_5)}{5} = \frac{9+0+1+3+2}{5} = \frac{15}{5} = 3,0s$$

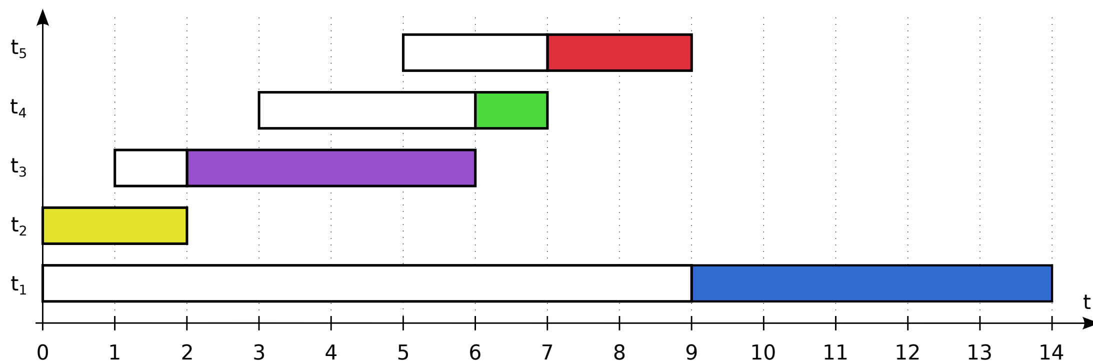

Figura 6.4: Escalonamento SJF.

A maior dificuldade no uso do algoritmo SJF consiste em estimar a priori a duração de cada tarefa, 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 ao algoritmo RR, esse algoritmo pode ser de grande valia, sobretudo para tarefas orientadas a entrada/saída.

Supondo 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 utilização de *quantum* provável 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 beneficiar 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 6.4.6.

6.4.4 Shortest Remaining Time First (SRTF)

O algoritmo SJF é cooperativo, ou seja, uma vez que uma tarefa recebe o processador, ela executa até encerrar (ou liberá-lo explicitamente). Em uma variante preemptiva, o escalonador deve comparar a duração prevista de cada nova tarefa que ingressa no sistema com o tempo de processamento restante das demais tarefas presentes, inclusive aquela que está executando no momento. Caso a nova tarefa tenha um tempo restante menor, ela recebe o processador. Essa abordagem é denominada

por alguns autores de *menor tempo restante primeiro* (SRTF – *Short Remaining Time First*) [Tanenbaum, 2003]. O algoritmo SRTF está exemplificado na Figura 6.5.

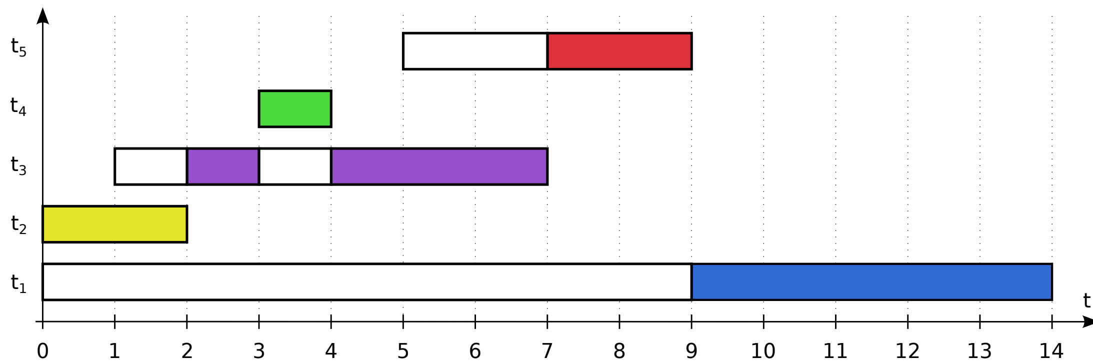

Figura 6.5: Escalonamento SRTF.

Calculando o tempo médio de execução T_t e o tempo médio de espera T_w para a execução da Figura 6.5, temos:

$$T_t = \frac{t_t(t_1) + \cdots + t_t(t_5)}{5} = \frac{14+2+6+1+4}{5} = \frac{27}{5} = 5,4s$$

$$T_w = \frac{t_w(t_1) + \cdots + t_w(t_5)}{5} = \frac{9+0+2+0+2}{5} = \frac{13}{5} = 2,6s$$

Pode-se observar que os tempos médios de execução T_t e de espera T_w são os menores observados até aqui. De fato, SRTF é o algoritmo de escalonamento que permite obter os mínimos valores de T_t e T_w. Ele torna o processamento de tarefas curtas muito eficiente, todavia com o risco de inanição das tarefas mais longas.

6.4.5 Escalonamento por prioridades fixas (PRIOc, PRIOp)

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 prioridade, a cada tarefa é associada uma prioridade, geralmente na forma de um número inteiro, que representa sua importância no sistema. Os valores de prioridade são então usados para definir a ordem de execução das tarefas. O algoritmo de escalonamento por prioridade define um modelo mais genérico de escalonamento, que permite modelar várias abordagens, entre as quais o FCFS e o SJF.

O escalonamento por prioridade pode ser cooperativo ou preemptivo. O diagrama da Figura 6.6 mostra o escalonamento das tarefas da Tabela 6.1 usando o **algoritmo por prioridade cooperativo**, ou PRIOc. Neste exemplo, os valores de prioridade são considerados em uma escala de prioridade positiva, ou seja, valores numéricos maiores indicam maior prioridade.

Calculando o tempo médio de execução T_t e o tempo médio de espera T_w para a execução da Figura 6.6, temos:

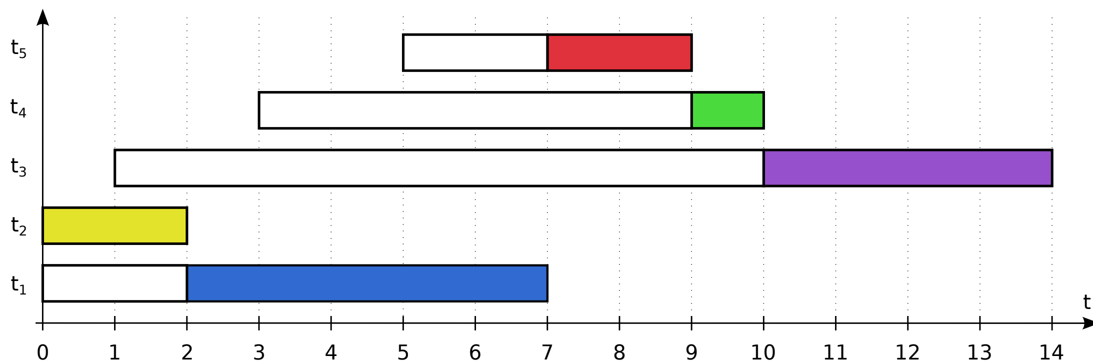

Figura 6.6: Escalonamento por prioridade cooperativo (PRIOc).

$$T_t = \frac{t_t(t_1) + \cdots + t_t(t_5)}{5} = \frac{7+2+13+7+4}{5} = \frac{33}{5} = 6,6s$$

$$T_w = \frac{t_w(t_1) + \cdots + t_w(t_5)}{5} = \frac{2+0+9+6+2}{5} = \frac{19}{5} = 3,8s$$

No **escalonamento por prioridade preemptivo** (PRIOp), quando uma tarefa de maior prioridade se torna disponível para execução, o escalonador entrega o processador a ela, trazendo a tarefa atualmente em execução de volta para a fila de prontas. Em outras palavras, a tarefa em execução pode ser "preemptada" por uma nova tarefa mais prioritária. Esse comportamento é apresentado na Figura 6.7 (observe que, quando t_4 ingressa no sistema, ela recebe o processador e t_1 volta a esperar na fila de prontas).

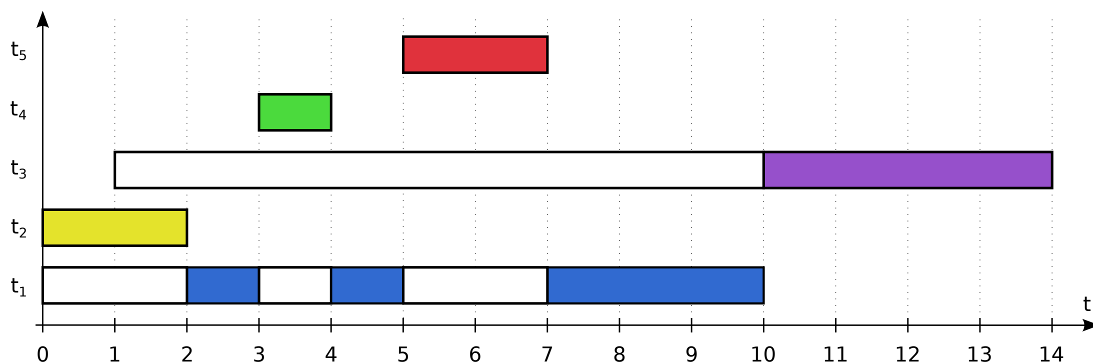

Figura 6.7: Escalonamento por prioridade preemptivo (PRIOp).

Calculando o tempo médio de execução T_t e o tempo médio de espera T_w para a execução da Figura 6.7, temos:

$$T_t = \frac{t_t(t_1) + \cdots + t_t(t_5)}{5} = \frac{10+2+13+1+2}{5} = \frac{28}{5} = 5,6s$$

$$T_w = \frac{t_w(t_1) + \cdots + t_w(t_5)}{5} = \frac{5+0+9+0+0}{5} = \frac{14}{5} = 2,8s$$

6.4.6 Escalonamento por prioridades dinâmicas (PRIOd)

No escalonamento por prioridades fixas apresentado na seção anterior, as tarefas de menor prioridade só recebem o processador na ausência de tarefas de maior prioridade. Caso existam tarefas de maior prioridade frequentemente ativas, as de menor prioridade podem "morrer de fome" por nunca conseguir chegar ao processador. Esse fenômeno se denomina **inanição** (do inglês *starvation*).

Para evitar a inanição das tarefas de menor prioridade, um fator interno denominado **envelhecimento** (*aging*) deve ser definido. O envelhecimento aumenta a prioridade da tarefa proporcionalmente ao tempo que ela está aguardando o processador. Dessa forma, o envelhecimento define um esquema de **prioridades dinâmicas**, que permite a elas executar periodicamente e assim evitar a inanição.

Uma forma de implementar o envelhecimento de tarefas está indicada no algoritmo a seguir, que associa duas prioridades a cada tarefa t_i: a prioridade estática ou fixa pe_i, definida por fatores externos, e a prioridade dinâmica pd_i, que evolui ao longo da execução (considera-se uma escala de prioridades positiva).

Definições:
- N : número de tarefas no sistema
- t_i : tarefa i, $1 \leq i \leq N$
- pe_i : prioridade estática da tarefa t_i
- pd_i : prioridade dinâmica da tarefa t_i

Quando uma tarefa nova t_{nova} ingressa no sistema:
- $pe_{nova} \leftarrow prioridade\ fixa$
- $pd_{nova} \leftarrow pe_{nova}$

Para escolher t_{prox}, a próxima tarefa a executar:
- escolher $t_{prox} \mid pd_{prox} = max_{i=1}^{N}(pd_i)$
- $\forall t_i \neq t_{prox} : pd_i \leftarrow pd_i + \alpha$
- $pd_{prox} \leftarrow pe_{prox}$

O funcionamento é simples: a cada turno o escalonador escolhe como próxima tarefa (t_{prox}) aquela que tiver a maior prioridade dinâmica (pd). A prioridade dinâmica das demais tarefas é aumentada de uma constante α, conhecida como *fator de envelhecimento* (ou seja, essas tarefas "envelhecem" um pouco e no próximo turno terão mais chances de ser escolhidas). Por sua vez, a tarefa escolhida (t_{prox}) "rejuvenesce", tendo sua prioridade dinâmica ajustada de volta para o valor de sua prioridade estática ($pd_{prox} \leftarrow pe_{prox}$).

Aplicando o escalonamento com prioridades dinâmicas com envelhecimento, a execução das tarefas da Tabela 6.1 pode ser observada[2] na Figura 6.8.

Calculando o tempo médio de execução T_t e o tempo médio de espera T_w para a execução da Figura 6.8, temos:

[2]Observe que não ocorre envelhecimento em $t = 6$, pois esse exemplo trata de um sistema com preempção por prioridades, mas não por tempo. Por isso, as prioridades das tarefas só são reavaliadas quando uma tarefa inicia ou encerra. Assim, as prioridades das tarefas são reavaliadas em $t = 5$ (entrada de t_5) e $t = 7$ (conclusão de t_5), mas não em $t = 6$.

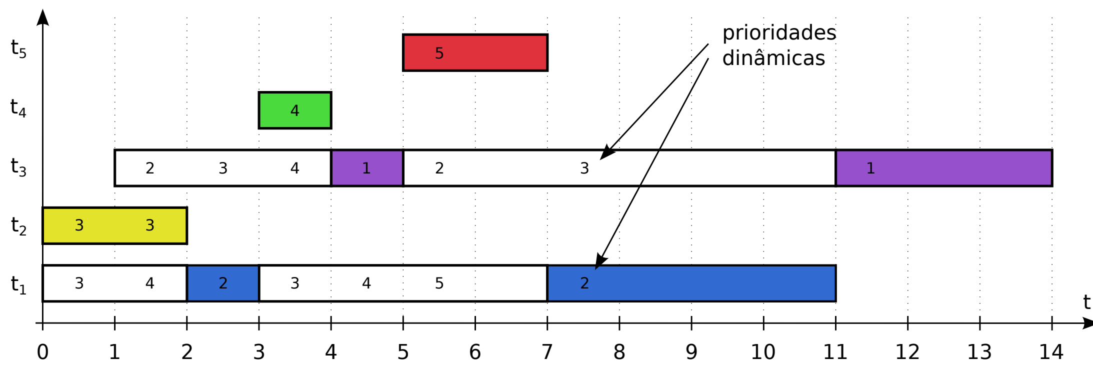

Figura 6.8: Escalonamento por prioridade preemptivo dinâmico (PRIOd).

$$T_t = \frac{t_t(t_1) + \cdots + t_t(t_5)}{5} = \frac{11 + 2 + 13 + 1 + 2}{5} = \frac{29}{5} = 5,8s$$

$$T_w = \frac{t_w(t_1) + \cdots + t_w(t_5)}{5} = \frac{6 + 0 + 9 + 0 + 0}{5} = \frac{15}{5} = 3,0s$$

As prioridades dinâmicas resolvem um problema importante em sistemas de tempo compartilhado. Nesses sistemas, as prioridades definidas pelo usuário estão intuitivamente relacionadas à **proporcionalidade** desejada na divisão do tempo de processamento. Por exemplo, caso um sistema de tempo compartilhado receba três tarefas: t_1 com prioridade 1, t_2 com prioridade 3 e t_3 com prioridade 6, espera-se que t_3 receba mais o processador que t_2, e esta mais que t_1 (assumindo uma escala de prioridades positiva).

Contudo, se associarmos apenas prioridades fixas a um escalonador *Round-Robin*, as tarefas irão executar de forma sequencial, sem a distribuição proporcional do processador. Esse resultado indesejável ocorre porque, a cada fim de *quantum*, sempre a tarefa mais prioritária é escolhida. Essa situação está ilustrada na Figura 6.9.

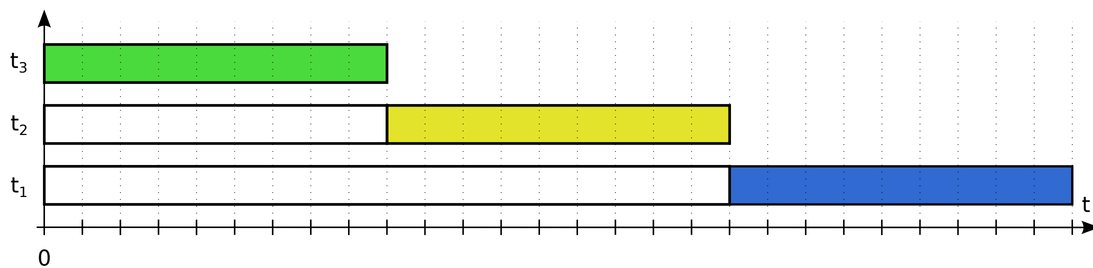

Figura 6.9: Escalonamento *Round-Robin* com prioridades fixas.

Usando o algoritmo de envelhecimento, a divisão do processador entre as tarefas se torna proporcional às suas prioridades estáticas. A Figura 6.10 ilustra a proporcionalidade obtida: 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 6.10: Escalonamento *Round-Robin* com prioridades dinâmicas.

6.4.7 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: 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: informações que podem ser obtidas ou estimadas pelo próprio escalonador, com base em dados disponíveis no sistema operacional. Os fatores internos mais utilizados são a idade da tarefa, sua duração estimada, sua interatividade, seu uso de memória ou de outros recursos, seu envelhecimento, etc. Os fatores internos mudam continuamente e devem ser recalculados periodicamente pelo escalonador.

Os fatores externos são usualmente sintetizados em um valor inteiro denominado **prioridade estática** (ou *prioridade de base*), que resume a "opinião" do usuário ou administrador sobre a importância daquela tarefa. Em geral, cada família de sistemas operacionais define sua própria escala de prioridades estáticas. Alguns exemplos de escalas comuns são:

Windows (2000 em diante): 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*:

- 24: *tempo real*
- 13: *alta*
- 10: *acima do normal*
- 8: *normal*
- 6: *abaixo do normal*
- 4: *baixa* ou *ociosa*

Além disso, geralmente a prioridade da tarefa responsável pela janela ativa recebe um incremento de prioridade: +1 ou +2, conforme a configuração do sistema (essa informação é considerada um fator interno de prioridade).

Linux (núcleo 2.4 em diante): considera duas escalas de prioridade separadas:

Tarefas de tempo real: usam uma escala de 1 a 99 positiva (valores maiores indicam **maior** prioridade). Somente o núcleo ou o administrador (*root*) podem lançar tarefas de tempo real.

Demais tarefas: usam uma escala que vai de −20 a +19, negativa (valores maiores indicam **menor** prioridade). Esta escala, denominada *nice level*, é padronizada em todos os sistemas UNIX-*like*. A prioridade das tarefas de usuário pode ser ajustada através dos comandos `nice` e `renice`.

6.4.8 Comparação entre os algoritmos apresentados

A Tabela 6.2 traz uma comparação sucinta entres os algoritmos apresentados nesta seção. Pode-se observar que os algoritmos preemptivos (RR, SRTF e PRIOp) possuem um número de trocas de contexto maior que seus correspondentes cooperativos, o que era de se esperar. Também pode-se constatar que o algoritmo SRTF proporciona os melhores tempos médios de execução T_t e de espera T_w, enquanto os piores tempos são providos pelo algoritmo RR (que, no entanto, oferece um melhor tempo de resposta a aplicações interativas).

Observa-se também que o tempo total de processamento é constante, pois ele só depende da carga de processamento de cada tarefa e não da ordem em que são executadas. Contudo, esse tempo pode ser influenciado pelo número de trocas de contexto, caso seja muito elevado.

Algoritmo de escalonamento	FCFS	RR	SJF	SRTF	PRIOc	PRIOp	PRIOd
Tempo médio de execução T_t	8,0	8,4	5,8	5,4	6,6	5,6	5,8
Tempo médio de espera T_w	5,2	5,6	3,0	2,6	3,8	2,8	3,0
Número de trocas de contexto	4	7	4	5	4	6	6
Tempo total de processamento	14	14	14	14	14	14	14

Tabela 6.2: Comparação entre os algoritmos apresentados.

6.4.9 Outros algoritmos de escalonamento

Além dos algoritmos vistos neste capítulo, diversos outros algoritmos foram propostos na literatura, alguns dos quais servem de base conceituais para escalonadores usados em sistemas operacionais reais. Dentre eles, podem ser citados os escalonadores de múltiplas filas, com ou sem *feedback* [Arpaci-Dusseau and Arpaci-Dusseau, 2014], os escalonadores justos [Kay and Lauder, 1988; Ford and Susarla, 1996], os escalonadores multiprocessador [Black, 1990] e multicore [Boyd-Wickizer et al., 2009], os escalonadores de tempo real [Farines et al., 2000] e os escalonadores multimídia [Nieh and Lam, 1997].

6.5 Escalonadores reais

Sistemas operacionais de mercado como Windows, Linux e MacOS são feitos para executar tarefas de diversos tipos, como jogos, editores de texto, conversores de vídeo, backups, etc. Para lidar com essa grande diversidade de tarefas, os escalonadores desses sistemas implementam algoritmos complexos, combinando mais de uma política de escalonamento.

No Linux, as tarefas são divididas em diversas classes de escalonamento, de acordo com suas demandas de processamento. Cada classe possui sua própria fila de tarefas, em um esquema conhecido como *Multiple Feedback Queues* [Arpaci-Dusseau and Arpaci-Dusseau, 2014]. As classes definidas no escalonador atual (núcleo 4.16, em 2018) são:

Classe `SCHED_DEADLINE`: classe específica para tarefas de tempo real que devem executar periodicamente e respeitar prazos (*deadlines*) predefinidos. A fila de tarefas desta classe é organizada por um algoritmo chamado *Earliest Deadline First* (EDF), usual em sistemas de tempo real [Farines et al., 2000].

Classe `SCHED_FIFO`: tarefas nesta classe são escalonadas usando uma política com prioridade fixa preemptiva, sem envelhecimento nem *quantum*. Portanto, uma tarefa nesta classe executa até bloquear por recursos, liberar explicitamente o processador (através da chamada de sistema `sched_yield()`) ou ser interrompida por outra tarefa de maior prioridade nesta mesma classe.

Classe `SCHED_RR`: esta classe implementa uma política similar à `SCHED_FIFO`, com a inclusão da preempção por tempo (*Round-Robin*). O valor default do *quantum* nesta classe é de 100 *ms*.

Classe `SCHED_NORMAL` ou `SCHED_OTHER`: é a classe padrão, que suporta as tarefas interativas dos usuários e a maioria dos serviços do SO. As tarefas são escalonadas por uma política baseada em prioridades dinâmicas (com envelhecimento) e *Round-Robin* com *quantum* variável, normalmente entre 0,75 *ms* e 6 *ms*.

Classe `SCHED_BATCH`: é similar à classe `SCHED_NORMAL`, mas pressupõe que as tarefas são orientadas a processamento (*CPU-bound*) e portanto são ativadas menos frequentemente que as tarefas em `SCHED_NORMAL`. Em compensação, recebem um *quantum* maior, de 1,5 segundos.

Classe `SCHED_IDLE`: classe com a menor prioridade de escalonamento; tarefas nesta classe só recebem processador caso não exista nenhuma outra tarefa em execução no sistema.

As tarefas são ativadas de acordo com sua classe, sendo `SCHED_DEADLINE` > `SCHED_FIFO/RR` > `SCHED_NORMAL` > `SCHED_BATCH` > `SCHED_IDLE`. Assim, tarefas nas chamadas classes interativas (`SCHED_NORMAL/BATCH/IDLE`) só executam se não houverem tarefas ativas nas classes denominadas de tempo real (`SCHED_DEADLINE/FIFO/RR`).

O núcleo Linux usa escalas de prioridade e algoritmos distintos para as várias classes de escalonamento. Para as tarefas interativas (`SCHED_NORMAL/BATCH/IDLE`) é usado o algoritmo *Completely Fair Scheduler* (CFS), baseado no conceito de escalonadores justos [Kay and Lauder, 1988], com prioridades entre -20 e +19. O CFS mantém a lista

de tarefas prontas em uma árvore rubro-negra ordenada por tempos de processamento realizado, o que confere uma boa escalabilidade (os custos de busca, inserção e remoção nessa árvore são da ordem de $O(\log n)$). Por outro lado, as tarefas nas classes de tempo real têm prioridades entre 1 e 99 e são escalonadas por algoritmos específicos, como EDF (*Earliest Deadline First*) para `SCHED_DEADLINE`, o PRIOp para `SCHED_FIFO` e o PRIOp+RR para `SCHED_RR`.

A Tabela 6.3 sintetiza as principais características dos escalonadores disponíveis no núcleo Linux:

Classe	`DEADLINE`	`FIFO`	`RR`	`NORMAL`	`BATCH`	`IDLE`
Precedência	1	2	2	3	4	5
Algoritmo	EDF	PRIOp	PRIOp + Round Robin	CFS	CFS	CFS
Tempo real	✓	✓	✓			
Preempção por prioridade	✓	✓	✓	✓	✓	✓
Preempção por tempo			✓	✓	✓	✓
Quantum			100 *ms*	0,75 a 6 *ms*	1500 *ms*	1500 *ms*
Prioridade fixa	0	1 a 99	1 a 99	0	0	0
Níveis "nice"				-20 a +19	-20 a +19	> +19
Envelhecimento				✓	✓	

Tabela 6.3: Classes de escalonamento no núcleo Linux.

Exercícios

1. Explique o que é escalonamento *round-robin*, dando um exemplo.

2. Considere um sistema de tempo compartilhado com valor de quantum t_q e duração da troca de contexto t_{tc}. Considere tarefas de entrada/saída que usam em média p% de seu quantum de tempo cada vez que recebem o processador. Defina a eficiência $\mathcal{E}$ do sistema como uma função dos parâmetros t_q, t_{tc} e p.

3. Explique o que é, para que serve e como funciona a técnica de *aging*.

4. No algoritmo de envelhecimento definido na Seção 6.4.6, o que seria necessário modificar para suportar uma escala de prioridades negativa?

5. A tabela a seguir representa um conjunto de tarefas prontas para utilizar um processador:

Tarefa	t_1	t_2	t_3	t_4	t_5
ingresso	0	0	3	5	7
duração	5	4	5	6	4
prioridade	2	3	5	9	6

Represente graficamente a sequência de execução das tarefas e calcule os tempos médios de vida (*tournaround time*) e de espera (*waiting time*), para as políticas de escalonamento a seguir:

(a) FCFS cooperativa
(b) SJF cooperativa
(c) SJF preemptiva (SRTF)
(d) PRIO cooperativa
(e) PRIO preemptiva
(f) RR com $t_q = 2$, sem envelhecimento

Considerações: todas as tarefas são orientadas a processamento; as trocas de contexto têm duração nula; em eventuais empates (idade, prioridade, duração, etc), a tarefa t_i com menor i prevalece; valores maiores de prioridade indicam maior prioridade.

6. Idem, para as tarefas da tabela a seguir:

Tarefa	t_1	t_2	t_3	t_4	t_5
ingresso	0	0	1	7	11
duração	5	6	2	6	4
prioridade	2	3	4	7	9

Referências

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

R. Arpaci-Dusseau and A. Arpaci-Dusseau. *Multi-level Feedback Queue*, chapter 8. Arpaci-Dusseau Books, 2014. http://pages.cs.wisc.edu/~remzi/OSTEP/cpu-sched-mlfq.pdf.

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

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

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

J.-M. Farines, J. da Silva Fraga, and R. S. de Oliveira. *Sistemas de Tempo Real – 12ª Escola de Computação da SBC*. Sociedade Brasileira de Computação, 2000. http://www.das.ufsc.br/gtr/livro.

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

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

J. Nieh and M. Lam. 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, 1997.

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