Sistemas Operacionais
II - Gerência de tarefas *

Prof. Carlos Alberto Maziero
PPGIA CCET PUCPR
http://www.ppgia.pucpr.br/~maziero

9 de maio de 2008

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.

Sumário

1 Objetivos

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

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

No entanto, um processador convencional somente trata um fluxo de instruções de cada vez. Até mesmo computadores com vários processadores (máquinas *Dual Pentium* ou processadores com tecnologia *hyper-threading*, por exemplo) têm mais atividades a executar que o número de processadores disponíveis. Como fazer para atender simultaneamente as múltiplas necessidades de processamento dos usuários? Uma solução ingênua seria equipar o sistema com um processador para cada tarefa, mas essa solução ainda é inviável econômica e tecnicamente. Outra solução seria *multiplexar o processador* entre as várias tarefas que requerem processamento. Por multiplexar entendemos compartilhar o uso do processador entre as várias tarefas, de forma a atendê-las da melhor maneira possível.

Os principais conceitos abordados neste capítulo compreendem:

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

2 O conceito de tarefa

Uma tarefa é definida como sendo a execução de um fluxo seqüencial 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 seqüência 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 seqüências 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 seqüências 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.

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

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

Dessa forma, o processador tem então de se multiplexar entre todas as tarefas definidas nessa "receita de torta" (e outras tarefas que possam estar sendo executadas pelos usuários) para atingir os objetivos esperados. Às vezes podemos ter mais de uma cozinheira trabalhando juntas na mesma cozinha, o que irá agilizar o trabalho, mas também tornará mais complexa a gerência das tarefas.

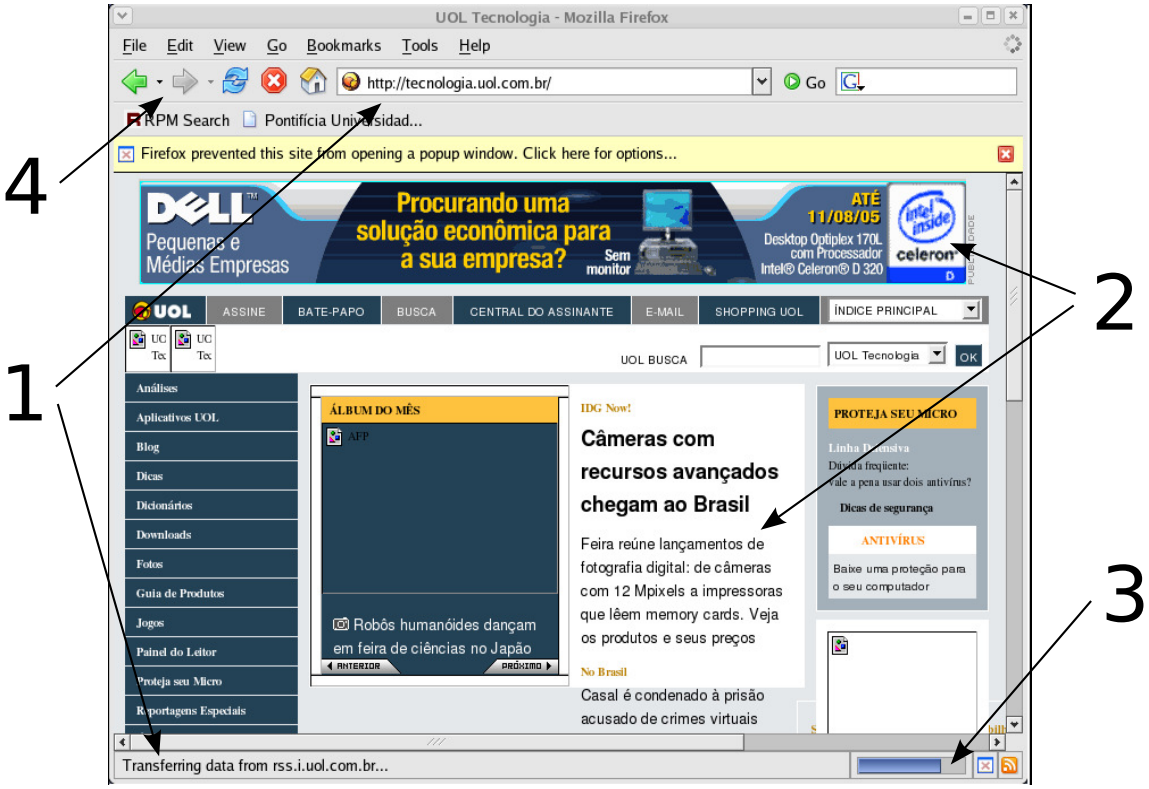

Figura 1: Tarefas de um navegador Internet

3 A gerência de tarefas

Os primeiros sistemas de computação, nos anos 40, executavam apenas uma tarefa de cada vez. Nestes sistemas, cada programa binário era carregado do disco para a memória e executado até sua conclusão. Os dados de entrada da tarefa eram carregados na memória juntamente com a mesma e os resultados obtidos no processamento eram descarregados de volta no disco após a conclusão da tarefa. Todas as operações de transferência de código e dados entre o disco e a memória eram coordenados por um operador humano. Esses sistemas primitivos eram usados sobretudo para aplicações de cálculo numérico, muitas vezes com fins militares (problemas de trigonometria, balística, mecânica dos fluidos, etc). A figura 2 a seguir ilustra um sistema desse tipo.

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.

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

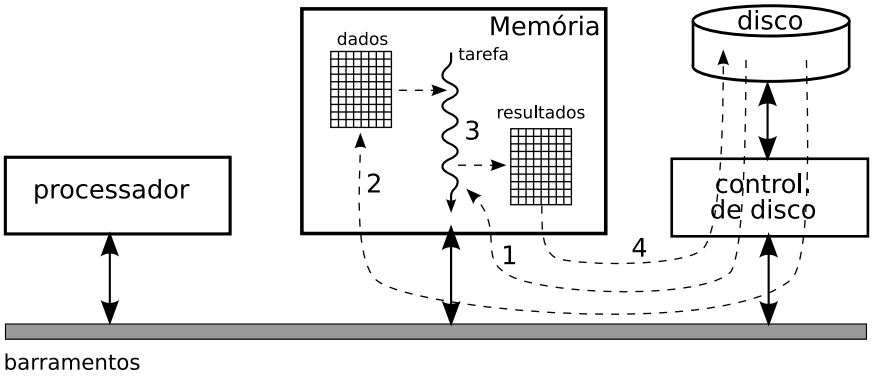

Figura 2: Sistema mono-tarefa: 1) carga do código na memória, 2) carga dos dados na memória, 3) processamento, consumindo dados e produzindo resultados, 4) ao término da execução, a descarga dos resultados no disco.

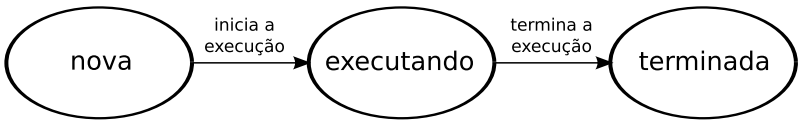

Figura 3: Diagrama de estados de uma tarefa em um sistema mono-tarefa.

demais programas. O programa monitor executava basicamente os seguintes passos sobre uma fila de programas a executar, armazenada no disco:

repetir
carregar um programa do disco para a memória
carregar os dados de entrada do disco para a memória
transferir a execução para o programa recém carregado
aguardar o término da execução do programa
escrever os resultados gerados pelo programa no disco
até processar todos os programas da fila

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.

O uso do programa monitor agilizou o uso do processador, mas outros problemas persistiam. Como a velocidade de processamento era muito maior que a velocidade

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

A solução encontrada para resolver esse problema foi permitir ao processador suspender a execução da tarefa que espera dados externos e passar a executar outra tarefa. Mais tarde, quando os dados de que necessita estiverem disponíveis, a tarefa suspensa pode ser retomada no ponto onde parou. Para tal, é necessário ter mais memória (para poder carregar mais de um programa ao mesmo tempo) e definir procedimentos para suspender uma tarefa e retomá-la mais tarde. O ato de retirar um recurso de uma tarefa (neste caso o recurso é o processador) é denominado *preempção*. Sistemas que implementam esse conceito são chamados *sistemas preemptivos*.

A adoção da preempção levou a sistemas mais produtivos (e complexos), nos quais várias tarefas podiam estar em andamento simultaneamente: uma estava ativa e as demais suspensas, esperando dados externos ou outras condições. Sistemas que suportavam essa funcionalidade foram denominados *monitores multi-tarefas*. O diagrama de estados da figura 4 ilustra o comportamento de uma tarefa em um sistema desse tipo:

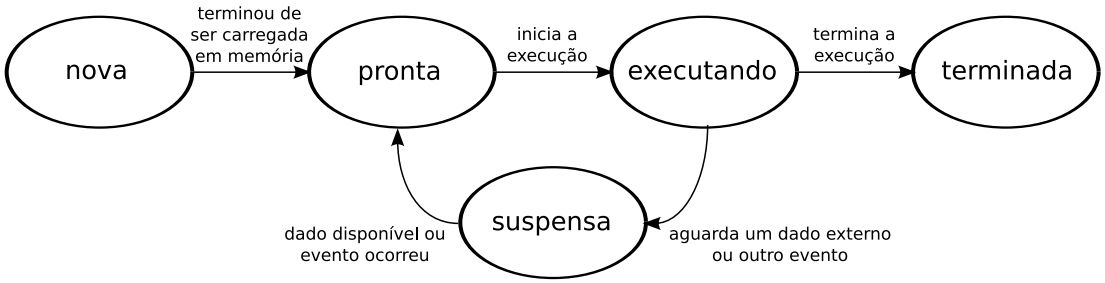

Figura 4: Diagrama de estados de uma tarefa em um sistema multi-tarefas.

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 ()
  {
    int i = 0, soma = 0 ;
```

[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!

```
  while (i < 1000)
    soma += i ; // erro: o contador i não foi incrementado

  printf ("A soma vale %d\n", soma);
}
```

Esse tipo de programa pode inviabilizar o 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* [Cor63]. 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. Essa "preempção por tempo de processamento" é implementada com a ajuda de temporizadores externos programáveis (implementados em hardware) que geram interrupções ao disparar.

O diagrama de estados das tarefas deve ser reformulado para incluir a preempção por tempo que implementa a estratégia de tempo compartilhado. A figura 5 apresenta esse novo diagrama, que é conhecido na literatura da área como *diagrama de ciclo de vida das tarefas*.

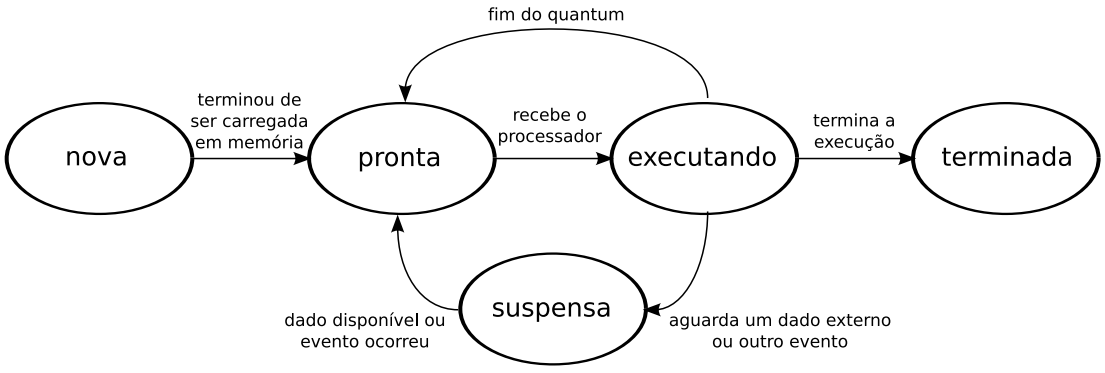

Figura 5: Diagrama de estados de uma tarefa em um sistema de tempo compartilhado.

Os estados e transições do ciclo de vida apresentado na figura 5 têm o seguinte significado:

[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 [Lov04].

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

Pronta : A tarefa está em memória, pronta para executar (ou para continuar sua execução), apenas aguardando a disponibilidade do processador. Todas as tarefas prontas são organizadas em uma fila cuja ordem é determinada por algoritmos de escalonamento, que serão estudados na seção 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 5 são as *transições* entre esses estados, que são explicadas a seguir:

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

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

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

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

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

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

Executando → 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.

4 Implementação de tarefas

Nesta seção são descritos os problemas relacionados à implementação do conceito de tarefa em um sistema operacional multi-tarefas. 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 a situação atual da tarefa: a instrução que ela está executando, os valores de suas variáveis, os arquivos que ela utiliza, por exemplo. Esse estado se modifica conforme a execução da tarefa avança. O estado de uma tarefa em um determinado instante é caracterizado pelas seguintes informações:

- Registradores do processador:
  - Contador de programa (PC – *Program Counter*), que indica a posição corrente da execução no código binário da tarefa.
  - Apontador de pilha (SP – *Stack Pointer*), que aponta para o topo da pilha de execução (estrutura que armazena os parâmetros e endereços de retorno das funções, entre outros dados).
  - Flags indicando vários aspectos do comportamento do processador naquele momento (nível usuário ou nível núcleo, status da última operação realizada, etc).
  - Demais registradores (acumulador, de uso geral, de mapeamento de memória, etc).
- Áreas de memória usadas pela tarefa.
- Recursos usados pela tarefa (arquivos abertos, conexões de rede e semáforos, entre outros).

As informações que permitem definir completamente o estado de uma tarefa são coletivamente denominadas *contexto da tarefa*. Cada tarefa ativa no sistema possui uma

estrutura de dados associada a ela, onde são armazenadas as informações relativas ao seu contexto e outros dados necessários à sua gerência. Essa estrutura de dados é geralmente chamada de TCB (do inglês *Task Control Block*). Cada TCB funciona como um "descritor de tarefa" e tipicamente contém as seguintes informações:

- Identificador da tarefa (geralmente um número inteiro).
- Estado da tarefa (nova, pronta, executando, suspensa ou terminada).
- Valores dos registradores do processador quando o contexto foi salvo pela última vez.
- Lista das áreas de memória usadas pela tarefa (exclusivas ou compartilhadas com outras tarefas).
- Listas de arquivos abertos, conexões de rede e outros recursos usados pela tarefa (exclusivos ou compartilhados com outras tarefas).
- Informações de contabilização (data de início, tempo de processamento, volume de dados lidos/escritos, etc.).
- Outras informações (prioridade, proprietário, etc.).

Os TCBs das tarefas são organizados em listas ou vetores (lista de tarefas prontas, lista de tarefas aguardando um pacote de rede, 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 um 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 (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, tempos de vida e 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 6 apresenta um diagrama temporal com os principais passos envolvidos em 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 em um periférico (também através de uma interrupção) ou pela execução de uma chamada de sistema pela tarefa corrente (ou seja, por uma interrupção de software).

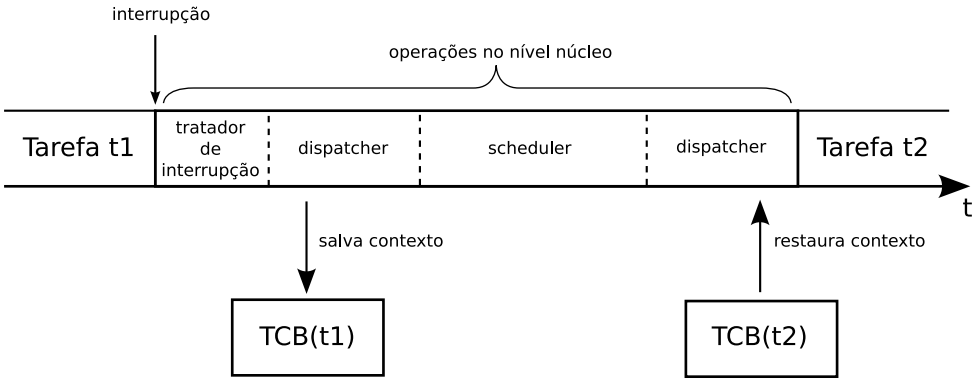

Figura 6: Passos de uma troca de contexto.

A realização de uma troca de contexto completa, envolvendo a interrupção de uma tarefa, armazenamento do contexto, escalonamento e reativação da tarefa escolhida, é uma operação relativamente rápida (de dezenas a centenas de micro-segundos, dependendo do hardware e do sistema operacional). Quanto menor o tempo despendido nas trocas de contexto, maior será a **eficiência** da gerência de tarefas, pois menos tempo será gasto nessas atividades e sobrará mais tempo de processador para as tarefas. Assim, é possível definir uma medida de eficiência $\mathcal{E}$ do uso do processador por um sistema de tempo compartilhado, que é função da relação entre as durações médias do *quantum* de tempo t_q e da troca de contexto t_{tc}:

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

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 do *quantum*, diminuindo o valor médio de t_q).

4.3 Processos

Além de seu próprio código, 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 [SGG01, Tan03] 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 seqüencial (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, 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 7 ilustra o conceito de processo, visto como um contêiner de recursos.

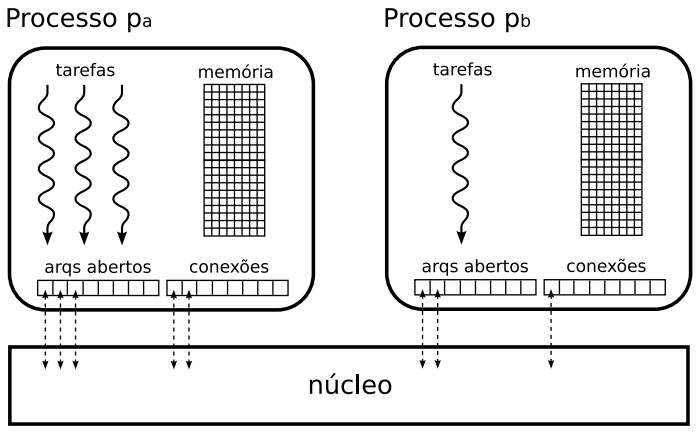

Figura 7: O processo visto como um contêiner de recursos.

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 identificado 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.

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 8 ilustra o funcionamento dessa chamada.

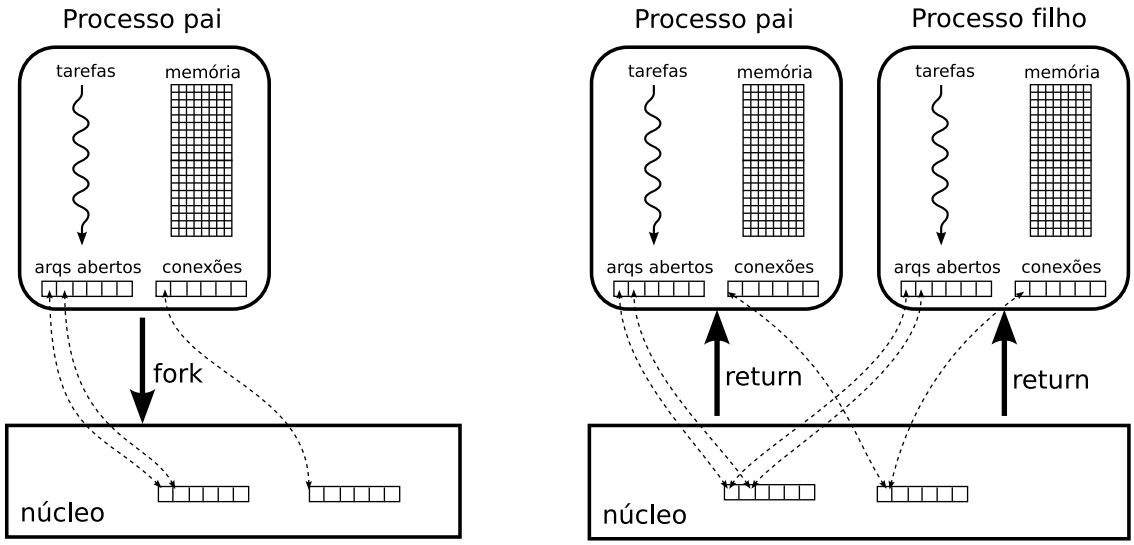

Figura 8: A chamada de sistema `fork`: antes (esq) e depois (dir) de sua execução pelo núcleo do sistema UNIX.

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:

```
1  #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 */

11   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 */
21   }
     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.

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

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

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*). Grosso modo, cada thread de usuário corresponde a uma tarefa a ser executada. Por sua vez, os fluxos de execução reconhecidos e gerenciados pelo núcleo do sistema operacional são são chamados de **threads de núcleo** (*kernel-level threads* ou *kernel threads*).

Sem poder contar com o suporte do sistema operacional para a criação de múltiplos threads, os desenvolvedores contornaram o problema construindo bibliotecas para gerenciar threads dentro de cada processo, sem o envolvimento do núcleo. 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 naquele 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 9 ilustra esse modelo.

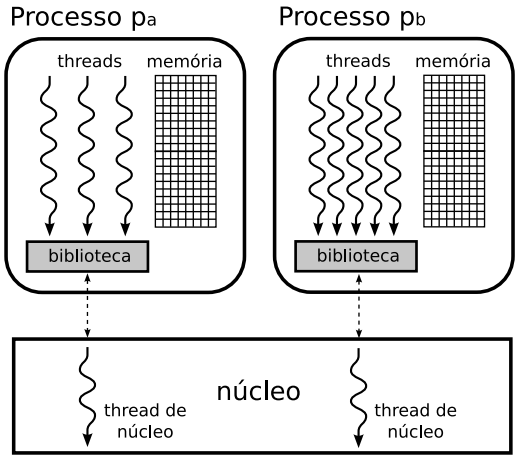

Figura 9: O modelo de threads N:1.

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 este modelo torna 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* [Eng05].

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

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

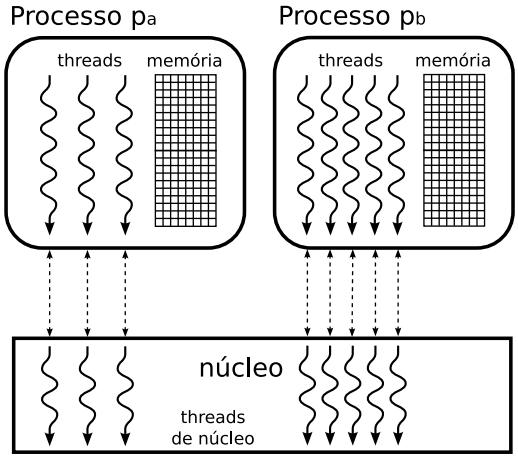

Figura 10: O modelo de threads 1:1.

minada **Modelo de Threads N:M**, onde N threads de usuário são mapeados em $M \leq N$ threads de núcleo. A figura 11 apresenta esse modelo.

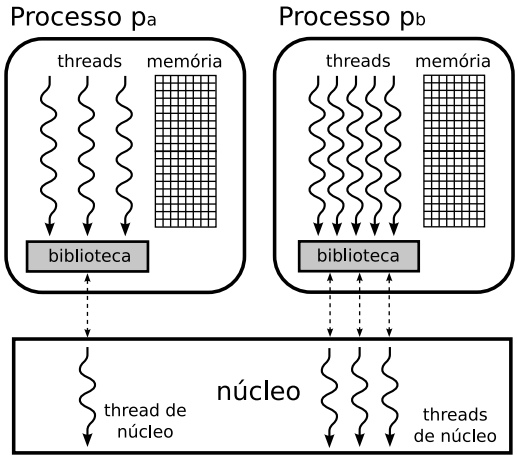

Figura 11: O modelo de threads N:M.

O modelo N:M é implementado pelo Solaris e também pelo projeto KSE (*Kernel-Scheduled Entities*) do FreeBSD [EE03] baseado nas idéias apresentadas em [ABLL92]. 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 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* [NBF96], que

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	depende
Divisão de recursos entre tarefas	injusta	justa	depende da situação
Exemplos	GNU Portable Threads	Windows XP, Linux	Solaris, FreeBSD KSE

Tabela 1: Quadro comparativo dos modelos de threads.

busca definir uma interface padronizada para a criação e manipulação de threads. Esse padrão é amplamente difundido e suportado hoje em dia. A listagem a seguir, extraída de [Bar05], exemplifica o uso do padrão *PThreads*.

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

#define NUM_THREADS 5

/* each thread will run this */
void *PrintHello(void *threadid)
{
   printf("%d: Hello World!\n", (int) threadid);
   pthread_exit(NULL);
}

/* main thread (create the other threads) */
int main (int argc, char *argv[])
{
   pthread_t thread[NUM_THREADS];
   int status, i;

   /* creating threads */
   for(i=0; i<NUM_THREADS; i++)
   {
      printf("Creating thread %d\n", i);
      status = pthread_create(&thread[i], NULL, PrintHello, (void *) i);
```

```
    if (status)
    {
      perror ("pthread_create");
      exit(-1);
    }
  }

  /* waiting for threads termination */
  for(i=0; i<NUM_THREADS; i++)
  {
    printf("Waiting for thread %d\n", i);
    status = pthread_join(thread[i], NULL);
    if (status)
    {
      perror ("pthread_join");
      exit(-1);
    }
  }

  pthread_exit(NULL);
}
```

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 das tarefas:

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 [BW97, FdSFdO00].

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 anti-vírus, 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* [oC05]) 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 freqüentemente utilizados são:

Tempo de execução ou de retorno (*turnaround time*, t_t): diz respeito ao tempo total de "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.

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 não-preemptivo

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

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 não-preemptivos : 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 também conhecidos como *cooperativos*, pois exigem a cooperação das tarefas para que todas possam executar.

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 seqüência, 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 12 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).

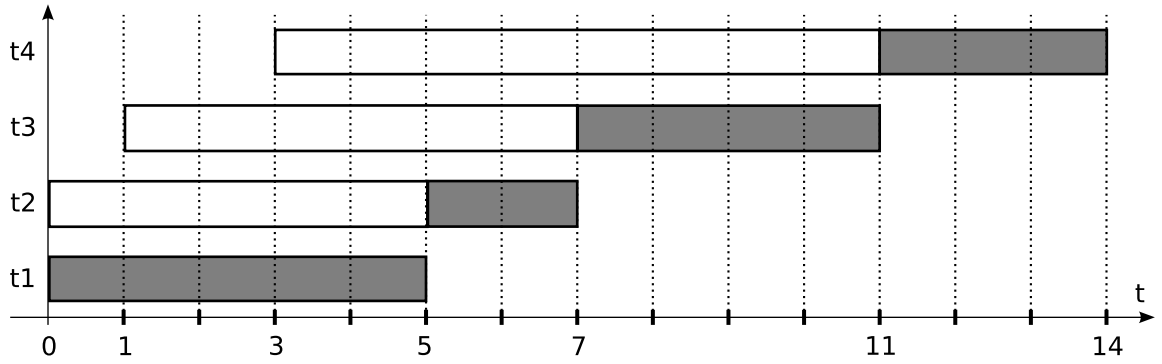

Figura 12: Escalonamento FCFS.

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:

$$T_t = \frac{t_t(t_1) + t_t(t_2) + t_t(t_3) + t_t(t_4)}{4} = \frac{(5-0) + (7-0) + (11-1) + (14-3)}{4}$$

$$= \frac{5+7+10+11}{4} = \frac{33}{4} = 8.25s$$

$$T_w = \frac{t_w(t_1)+t_w(t_2)+t_w(t_3)+t_w(t_4)}{4} = \frac{(0-0)+(5-0)+(7-1)+(11-3)}{4}$$

$$= \frac{0+5+6+8}{4} = \frac{19}{4} = 4.75s$$

O escalonamento FCFS não leva em conta a importância das tarefas nem seu comportamento em relação aos recursos. Por exemplo, com esse algoritmo as tarefas orientadas a entrada/saída irão receber menos tempo de processador que as tarefas orientadas a processamento (pois geralmente não usam integralmente seus *quanta* de tempo), o que pode ser prejudicial para aplicações interativas.

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 seqüência de escalonamento apresentada na figura 13.

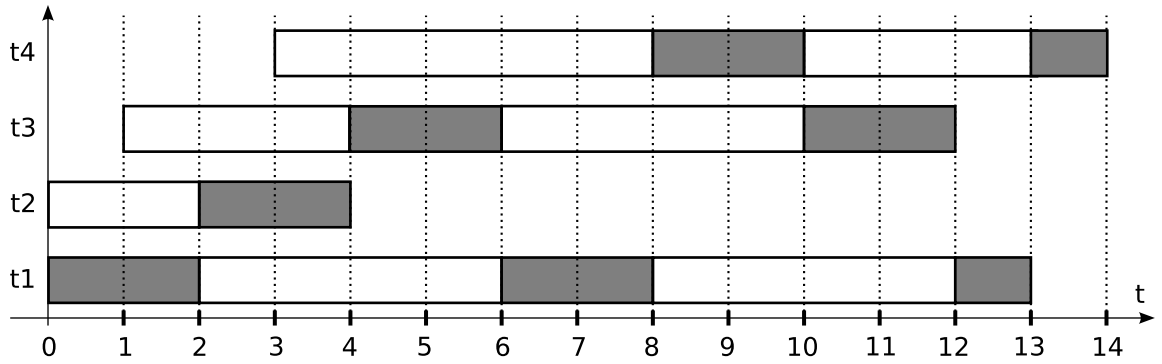

Figura 13: Escalonamento *Round-Robin*.

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

$$T_t = \frac{t_t(t_1)+t_t(t_2)+t_t(t_3)+t_t(t_4)}{4} = \frac{(13-0)+(4-0)+(12-1)+(14-3)}{4}$$

$$= \frac{13+4+11+11}{4} = \frac{39}{4} = 9.75s$$

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

Observa-se o aumento nos tempos T_t e T_w e também mais trocas de contexto que no algoritmo FCFS, 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 melhores tempos de resposta às 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 14.

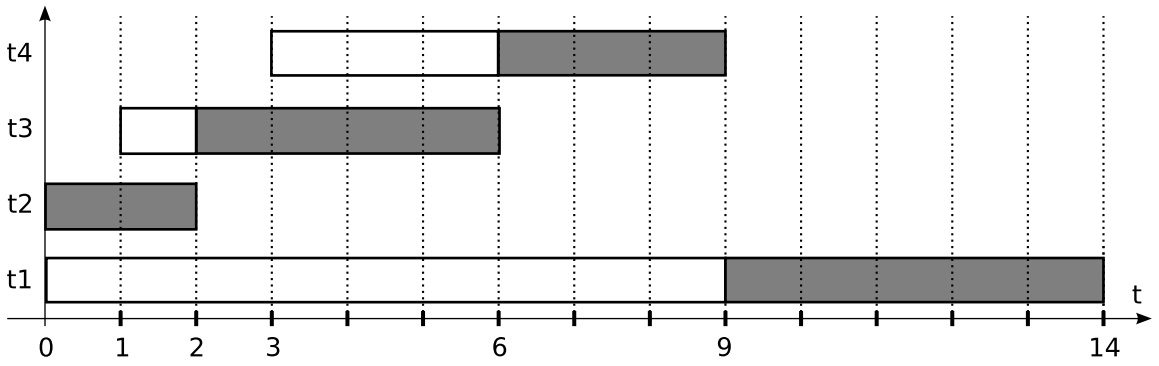

Figura 14: Escalonamento SJF.

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

$$\begin{aligned} T_t &= \frac{t_t(t_1) + t_t(t_2) + t_t(t_3) + t_t(t_4)}{4} = \frac{(14-0) + (2-0) + (6-1) + (9-3)}{4} \\ &= \frac{14+2+5+6}{4} = \frac{27}{4} = 6.75s \\ T_w &= \frac{t_w(t_1) + t_w(t_2) + t_w(t_3) + t_w(t_4)}{4} = \frac{(9-0) + (0-0) + (2-1) + (6-3)}{4} \\ &= \frac{9+0+1+3}{4} = \frac{13}{4} = 3.25s \end{aligned}$$

Deve-se observar que o comportamento expresso na figura 14 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*) [Tan03].

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).

5.5 Escalonamento baseado em 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.

O algoritmo de escalonamento baseado em prioridades define um modelo genérico de escalonamento, que permite modelar várias abordagens, entre as quais o FCFS e o SJF. 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. Em geral, cada família de sistemas operacionais define sua própria escala de prioridades. 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*
- 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.

Para exemplificar o escalonamento baseado em 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 15 mostra o escalonamento do processador usando o algoritmo baseado em prioridades em modo cooperativo (ou seja, sem *quantum* ou outras interrupções).

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

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

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

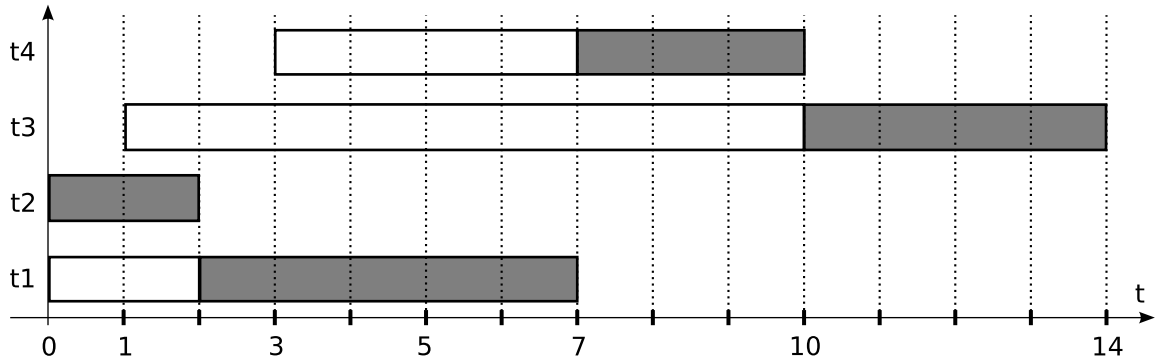

Figura 15: Escalonamento baseado em prioridades (cooperativo).

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 baseado em prioridades *preemptivo*, cujo comportamento é apresentado na figura 16 (observe que, quando t_4 ingressa no sistema, ela recebe o processador e t_1 volta a esperar na fila de prontas).

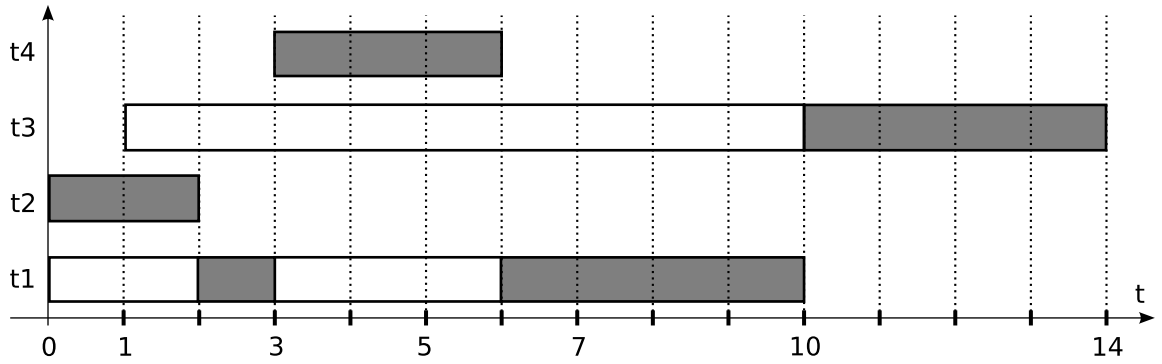

Figura 16: Escalonamento baseado em prioridades (preemptivo).

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

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

$$T_w = \frac{t_w(t_1) + t_w(t_2) + t_w(t_3) + t_w(t_4)}{4} = \frac{5 + 0 + 9 + 0}{4} = \frac{14}{4} = 3.5s$$

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

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.

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).

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 17 resume esse procedimento.

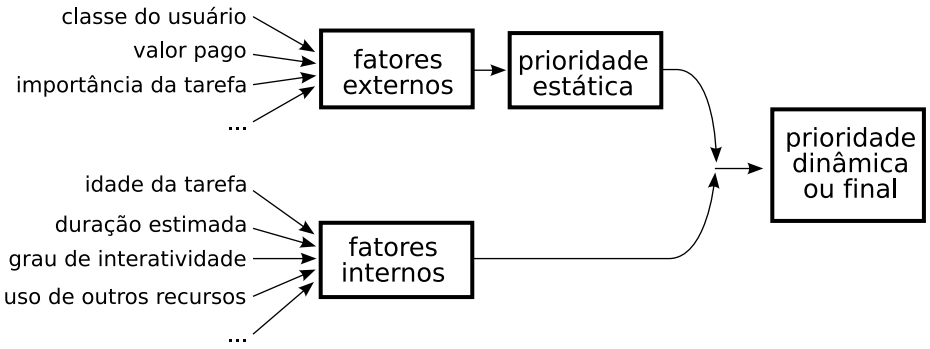

Figura 17: Composição da prioridade dinâmica.

Em um sistema com escalonador baseado em prioridades e preempção por tempo (*quantum*), as tarefas orientadas a processamento mais prioritárias tendem a monopolizar o processador, não permitindo que as tarefas de menor prioridade executem.

Em sistemas interativos, a intuição associada às prioridade estáticas (definidas pelo usuário) é a de *proporcionalidade* na divisão do tempo de processamento. Por exemplo, se um sistema recebe simultaneamente duas tarefas orientadas a processamento com a mesma prioridade, espera-se que cada uma receba 50% do processador e que ambas terminem juntas. Caso o sistema receba duas tarefas t_1 e t_2 com prioridade 2 e uma tarefa t_3 com prioridade 1, espera-se que t_1 e t_2 recebam cada uma 40% do processador, enquanto t_3 recebe apenas 20% (assumindo uma escala de prioridades positiva). Entretanto, se aplicarmos o algoritmo de prioridades básico a tarefa t_3 só irá executar após o término de t_1 e t_2, sem respeitar a proporcionalidade. Essa situação está ilustrada na figura 18.

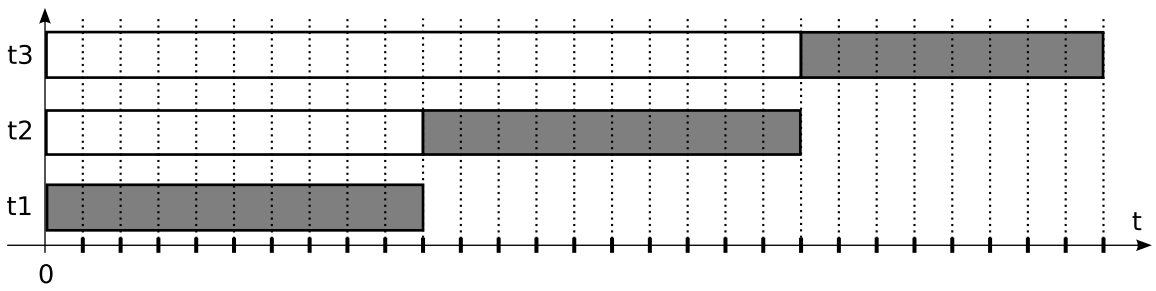

Figura 18: Violação da proporcionalidade.

Para garantir a proporcionalidade no uso do processador 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. 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 nova tarefa 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 ela recebe o processador. A prioridade dinâmica das demais tarefas é aumentada de α, ou seja, elas "envelhecem" e no próximo turno terão mais chance de ser escolhidas. A constante α é conhecida como *fator de envelhecimento*. Usando esse algoritmo, a divisão do processador entre as tarefas se torna proporcional às suas prioridades. A figura 19 ilustra essa proporcionalidade na execução de 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_1 recebe proporcionalmente mais tempo de processador que t_2, e que t_2 recebe mais que t_3.

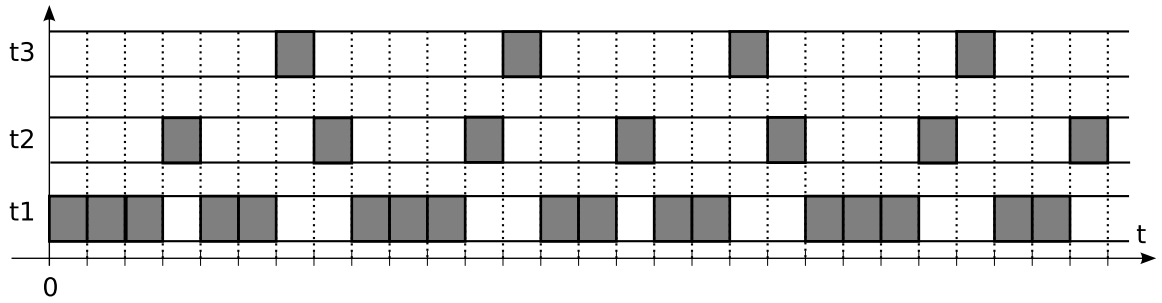

Figura 19: Proporcionalidade garantida através do envelhecimento.

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 20): um escalonador de tarefas de tempo-real (classes `SCHED_FIFO` e `SCHED_RR`) e um escalonador de tarefas interativas (classe `SCHED_OTHER`) [Lov04]. 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 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`.

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`.

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 [FdSFdO00], os escalonadores multimídia [NL97], os escalonadores justos [KL88, FS96] e os escalonadores multi-processador [Bla90].

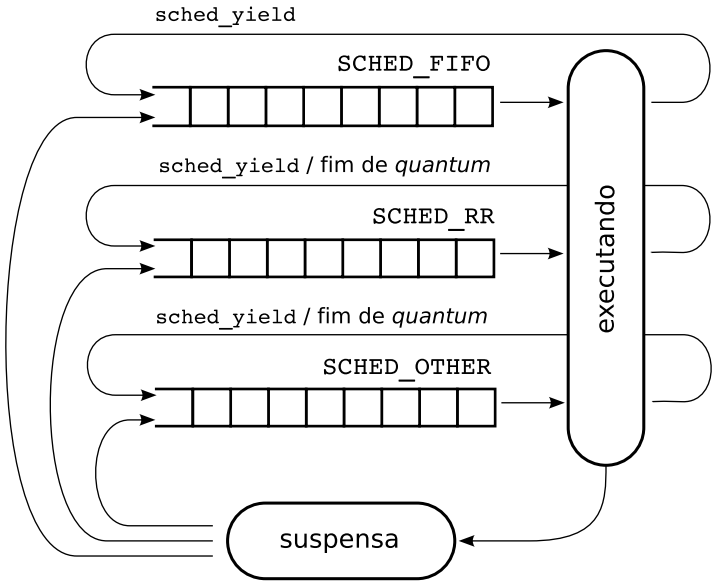

Figura 20: O escalonador multi-filas do Linux.

Questões

1. Explique o que é, para que serve e o que contém um PCB - *Process Control Block*.
2. O que são *threads* e para que servem?
3. Quais as principais vantagens e desvantagens de *threads* em relação a processos?
4. Forneça dois exemplos de problemas cuja implementação *multi-thread* não tem desempenho melhor que a respectiva implementação seqüencial.
5. O que significa *time sharing* e qual a sua importância em um sistema operacional?
6. Como e com base em que critérios é escolhida a duração de um *quantum* de processamento?
7. Explique o que é escalonamento *round-robin*, dando um exemplo.
8. Explique o que é, para que serve e como funciona a técnica de *aging*.
9. Explique os conceitos de *inversão* e *herança de prioridade*.

Exercícios

1. Considerando o diagrama de estados dos processos apresentado na figura 21, complete o diagrama com a transição de estado que está faltando e apresente o significado de cada um dos estados e transições.

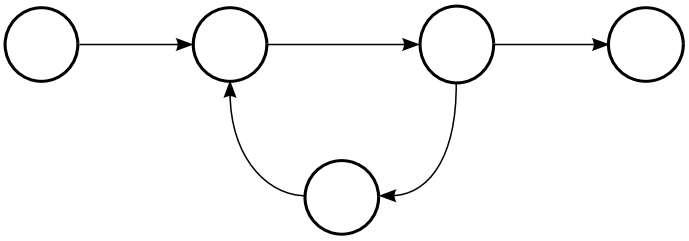

Figura 21: Diagrama de estados de processos.

2. Indique se cada uma das transições de estado de tarefas a seguir definidas é possível ou não. Se for possível, dê um exemplo de situação na qual essa transição ocorre.

   (a) executando → pronta
   (b) executando → suspensa
   (c) suspensa → executando
   (d) suspensa → terminada
   (e) executando → terminada
   (f) pronta → suspensa

3. Relacione as afirmações abaixo aos respectivos estados no ciclo de vida das tarefas (Nova, Pronta, Executando, Suspensa, Terminada):

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

4. Desenhe o diagrama de tempo da execução do código a seguir e informe qual a saída do programa na tela e a duração aproximada de sua execução.

```
void main()
{
   int x = 0 ;

   fork () ;
   x++ ;
   sleep (5) ;
   wait (0) ;

   fork () ;
   wait (0) ;
   sleep (5) ;
   x++ ;

   print ("Valor de x: %d\n", x) ;
}
```

5. Considerando as implementações de threads N:1 e 1:1 para o trecho de código a seguir, a) desenhe os diagramas de execução, b) informe as durações aproximadas de execução e c) indique a saída do programa na tela. Considere a operação `sleep()` como uma chamada de sistema (*syscall*).

   Significado das operações:

   - `thread_create`: cria uma nova thread, que pode executar (ser escalonada) imediatamente.
   - `thread_join`: espera o encerramento da thread informada como parâmetro.
   - `thread_exit`: encerra a thread.

```
int y = 0 ;

void threadBody
{
   int x = 0 ;
   sleep (10) ;
   printf ("x: %d, y:%d\n", ++x, ++y) ;
   thread_exit();
}

main ()
{
   thread_create (&tA, threadBody, ...) ;
   thread_create (&tB, threadBody, ...) ;
   sleep (1) ;
   thread_join (&tA) ;
   thread_join (&tB) ;
   sleep (1) ;
   thread_create (&tC, threadBody, ...) ;
```

```
  thread_join (&tC) ;
}
```

6. A tabela a seguir representa um conjunto de tarefas prontas para utilizar o processador. Para as políticas FCFS, SJF e PRIO cooperativas, indique a seqüência de execução das tarefas, o tempo médio de execução (*tournaround time*) e o tempo médio de espera (*waiting time*).

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

   Considerações: as trocas de contexto têm duração nula; para tarefas de mesma prioridade, use FCFS como critério de desempate; para tarefas de mesma idade, a tarefa t_i com menor i prevalece; todas as tarefas são orientadas a processamento; valores maiores de prioridade indicam maior prioridade.

7. Repita o exercício anterior para as políticas SJF e PRIO preemptivas.

8. Idem, para a política RR (*Round-Robin*) com $t_q = 2$.

9. Associe as afirmações a seguir aos seguintes modelos de threads: a) *many-to-one* (N:1); b) *one-to-one* (1:1); c) *many-to-many* (N:M):

   (a) Tem a implementação mais simples, leve e eficiente.
   (b) Multiplexa os threads de usuário em um pool de threads de kernel.
   (c) Pode impor uma carga muito pesada ao kernel.
   (d) Não permite explorar a presença de várias CPUs.
   (e) Permite uma maior concorrência sem impor muita carga ao kernel.
   (f) Geralmente implementado por bibliotecas.
   (g) É o modelo implementado no Windows NT e seus sucessores.
   (h) Se um thread bloquear, todos os demais têm de esperar por ele.
   (i) Cada thread no nível do usuário tem sua correspondente dentro do kernel.
   (j) É o modelo com implementação mais complexa.

10. 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.

Projetos

Referências

[ABLL92] Thomas Anderson, Brian Bershad, Edward Lazowska, and Henry Levy. Scheduler activations: effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[Bar05] Blaise Barney. POSIX threads programming. http://www.llnl.gov/computing/tutorials/pthreads, 2005.

[Bla90] D. L. Black. Scheduling and resource management techniques for multiprocessors. Technical Report CMU-CS-90-152, Carnegie-Mellon University, Computer Science Dept, 1990.

[BW97] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages, 2nd edition*. Addison-Wesley, 1997.

[Cor63] Fernando Corbató. *The Compatible Time-Sharing System: A Programmer's Guide*. MIT Press, 1963.

[EE03] Jason Evans and Julian Elischer. Kernel-scheduled entities for FreeBSD. http://www.aims.net.au/chris/kse, 2003.

[Eng05] Ralf Engeschall. The GNU Portable Threads. http://www.gnu.org/software/pth, 2005.

[FdSFdO00] Jean-Marie Farines, Joni da Silva Fraga, and Rômulo Silva de Oliveira. *Sistemas de Tempo Real – 12ª Escola de Computação da SBC*. Sociedade Brasileira de Computação, 2000.

[FS96] Bryan Ford and Sai Susarla. CPU Inheritance Scheduling. In *Usenix Association Second Symposium on Operating Systems Design and Implementation (OSDI)*, pages 91–105, 1996.

[KL88] J. Kay and P. Lauder. A fair share scheduler. *Communications of the ACM*, 31(1):44–55, January 1988.

[Lov04] Robert Love. *Linux Kernel Development*. Sams Publishing Developer's Library, 2004.

[NBF96] Bradford Nichols, Dick Buttlar, and Jacqueline Farrell. *PThreads Programming*. O'Reilly, 1996.

[NL97] 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.

[oC05] University of California. The SETI@home project. http://setiathome.ssl.berkeley.edu, 2005.

[SGG01] Abraham Silberschatz, Peter Galvin, and Greg Gane. *Sistemas Operacionais – Conceitos e Aplicações*. Campus, 2001.

[Tan03] Andrew Tanenbaum. *Sistemas Operacionais Modernos, 2ª edição*. Pearson – Prentice-Hall, 2003.

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. 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).

```
struct task_struct {
        volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
        struct thread_info *thread_info;
        atomic_t usage;
        unsigned long flags; /* per process flags, defined below */
        unsigned long ptrace;

        int lock_depth;      /* BKL lock depth */

        int prio, static_prio;
        struct list_head run_list;
        prio_array_t *array;

        unsigned long sleep_avg;
        unsigned long long timestamp, last_ran;
        unsigned long long sched_time; /* sched_clock time spent running */
        int activated;

        unsigned long policy;
        cpumask_t cpus_allowed;
        unsigned int time_slice, first_time_slice;

#ifdef CONFIG_SCHEDSTATS
        struct sched_info sched_info;
#endif

        struct list_head tasks;
        /*
         * ptrace_list/ptrace_children forms the list of my children
         * that were stolen by a ptracer.
         */
        struct list_head ptrace_children;
        struct list_head ptrace_list;

        struct mm_struct *mm, *active_mm;

/* task state */
        struct linux_binfmt *binfmt;
        long exit_state;
        int exit_code, exit_signal;
        int pdeath_signal; /* The signal sent when the parent dies */
        /* ??? */
        unsigned long personality;
        unsigned did_exec:1;
```

```
        pid_t pid;
        pid_t tgid;
        /*
         * pointers to (original) parent process, youngest child, younger sibling,
         * older sibling, respectively. (p->father can be replaced with
         * p->parent->pid)
         */
        struct task_struct *real_parent; /* real parent process (when being debugged) */
        struct task_struct *parent; /* parent process */
        /*
         * children/sibling forms the list of my children plus the
         * tasks I'm ptracing.
         */
        struct list_head children;  /* list of my children */
        struct list_head sibling;   /* linkage in my parent's children list */
        struct task_struct *group_leader; /* threadgroup leader */

        /* PID/PID hash table linkage. */
        struct pid pids[PIDTYPE_MAX];

        struct completion *vfork_done;     /* for vfork() */
        int __user *set_child_tid;         /* CLONE_CHILD_SETTID */
        int __user *clear_child_tid;       /* CLONE_CHILD_CLEARTID */

        unsigned long rt_priority;
        cputime_t utime, stime;
        unsigned long nvcsw, nivcsw; /* context switch counts */
        struct timespec start_time;
/* mm fault and swap info: this can arguably be seen as either mm-specific or thread-specific */
        unsigned long min_flt, maj_flt;

        cputime_t it_prof_expires, it_virt_expires;
        unsigned long long it_sched_expires;
        struct list_head cpu_timers[3];

/* process credentials */
        uid_t uid,euid,suid,fsuid;
        gid_t gid,egid,sgid,fsgid;
        struct group_info *group_info;
        kernel_cap_t cap_effective, cap_inheritable, cap_permitted;
        unsigned keep_capabilities:1;
        struct user_struct *user;
#ifdef CONFIG_KEYS
        struct key *thread_keyring; /* keyring private to this thread */
#endif
        int oomkilladj; /* OOM kill score adjustment (bit shift). */
        char comm[TASK_COMM_LEN]; /* executable name excluding path
                                  - access with [gs]et_task_comm (which lock
                                    it with task_lock())
                                  - initialized normally by flush_old_exec */
/* file system info */
```

```
        int link_count, total_link_count;
/* ipc stuff */
        struct sysv_sem sysvsem;
/* CPU-specific state of this task */
        struct thread_struct thread;
/* filesystem information */
        struct fs_struct *fs;
/* open file information */
        struct files_struct *files;
/* namespace */
        struct namespace *namespace;
/* signal handlers */
        struct signal_struct *signal;
        struct sighand_struct *sighand;

        sigset_t blocked, real_blocked;
        struct sigpending pending;

        unsigned long sas_ss_sp;
        size_t sas_ss_size;
        int (*notifier)(void *priv);
        void *notifier_data;
        sigset_t *notifier_mask;

        void *security;
        struct audit_context *audit_context;
        seccomp_t seccomp;

/* Thread group tracking */
        u32 parent_exec_id;
        u32 self_exec_id;
/* Protection of (de-)allocation: mm, files, fs, tty, keyrings */
        spinlock_t alloc_lock;
/* Protection of proc_dentry: nesting proc_lock, dcache_lock, write_lock_irq(&tasklist_lock); */
        spinlock_t proc_lock;
/* context-switch lock */
        spinlock_t switch_lock;

/* journalling filesystem info */
        void *journal_info;

/* VM state */
        struct reclaim_state *reclaim_state;

        struct dentry *proc_dentry;
        struct backing_dev_info *backing_dev_info;

        struct io_context *io_context;

        unsigned long ptrace_message;
        siginfo_t *last_siginfo; /* For ptrace use. */
```

```
/*
 * current io wait handle: wait queue entry to use for io waits
 * If this thread is processing aio, this points at the waitqueue
 * inside the currently handled kiocb. It may be NULL (i.e. default
 * to a stack based synchronous wait) if its doing sync IO.
 */
       wait_queue_t *io_wait;
/* i/o counters(bytes read/written, #syscalls */
       u64 rchar, wchar, syscr, syscw;
#if defined(CONFIG_BSD_PROCESS_ACCT)
       u64 acct_rss_mem1;   /* accumulated rss usage */
       u64 acct_vm_mem1;    /* accumulated virtual memory usage */
       clock_t acct_stimexpd; /* clock_t-converted stime since last update */
#endif
#ifdef CONFIG_NUMA
       struct mempolicy *mempolicy;
       short il_next;
#endif
#ifdef CONFIG_CPUSETS
       struct cpuset *cpuset;
       nodemask_t mems_allowed;
       int cpuset_mems_generation;
#endif
};
```