Capítulo

8

Organização de Sistemas Operacionais Convencionais e de Tempo Real

Rômulo S. de Oliveira, Alexandre da S. Carissimi e Simão S. Toscani

Abstract

The objective of this text is to present the many design solutions for operating system kernels, both conventional and real-time. It defines a taxonomy for the organization of operating systems and shows the consequences of each type of organization with respect to general system properties, such as performance, reliability, etc. Finally, it presents Win2000, Linux and variations of Linux for real-time as case studies. After reading this text, the reader should be able to infer the fundamental properties of an operating system that are a direct consequence of its internal organization.

Resumo

O objetivo deste texto é apresentar as diferentes soluções de projeto (design) para o núcleo de sistemas operacionais, tanto convencionais como de tempo real. A partir do estabelecimento de uma taxonomia para a organização interna dos sistemas operacionais, mostrar as conseqüências de cada tipo de organização com respeito a propriedades gerais do sistema, tais como desempenho, confiabilidade, etc. Finalmente, Win2000, Linux e variações do Linux para tempo real são usados como estudo de caso. Ao final, espera-se que o leitor seja capaz de inferir as propriedades fundamentais de um sistema operacional que são uma conseqüência direta de sua organização interna.

8.1. Introdução

O **sistema operacional** é uma camada de software colocada entre o hardware e os programas que executam tarefas para os usuários. Essa visão de um sistema computacional é ilustrada na figura 8.1. O sistema operacional procura tornar a utilização do computador, ao mesmo tempo, mais eficiente e mais conveniente. Maior eficiência significa mais trabalho obtido do mesmo hardware. Uma utilização mais conveniente vai diminuir o tempo necessário para a construção dos programas, o que implica a redução no custo do software, e vai tornar o usuário final mais produtivo.

Figura 8.1. Sistema computacional.

Uma utilização mais eficiente do computador é obtida através da distribuição de seus recursos entre os programas. Neste contexto, são considerados recursos quaisquer componentes do hardware disputados pelos programas. Por exemplo, espaço na memória principal, tempo de processador, impressora, espaço e acesso a disco, etc.

Uma utilização mais conveniente do computador é obtida, escondendo-se do programador detalhes do hardware, em especial dos periféricos. Ao esconder os detalhes dos periféricos, muitas vezes são criados recursos de mais alto nível. Por exemplo, os programas utilizam o espaço em disco através do conceito de arquivo. Arquivos não existem no hardware. Eles formam um recurso criado a partir do que o hardware oferece. Para o programador, é muito mais confortável trabalhar com arquivos do que receber uma área de espaço em disco que ele próprio teria que organizar. O usuário final também é beneficiado quando o sistema operacional esconde os detalhes necessários para a administração do computador e para a execução dos programas.

Por ser ele próprio um programa de computador, o sistema operacional possui uma especificação e um projeto. A sua especificação corresponde à lista de serviços que deve oferecer e de chamadas de sistema que deve suportar. Por outro lado, o seu **projeto** ou ***design*** diz respeito à sua estrutura interna, como as diferentes partes necessárias a sua implementação são organizadas internamente. O objetivo deste texto é apresentar as diferentes soluções de projeto (*design*) para o núcleo de sistemas operacionais, tanto convencionais como de tempo real. A partir do estabelecimento de uma taxonomia para a organização interna dos sistemas operacionais, o texto aponta as conseqüências de cada tipo de organização com respeito a propriedades gerais do sistema, tais como: desempenho, confiabilidade, extensibilidade, etc. Finalmente, usar alguns sistemas operacionais como estudo de caso. Ao final, espera-se que o leitor seja capaz de inferir as propriedades fundamentais de um sistema operacional que surgem como conseqüência direta de sua organização interna.

A seção 8.2 contém uma revisão de programação concorrente, necessária para o entendimento do restante do texto. A seção 8.3 apresenta uma taxonomia para a organização interna dos sistemas operacionais, a qual procura sintetizar as principais opções escolhidas pelos projetistas. A seção 8.4 descreve a problemática da construção de sistemas operacionais de tempo real. Na seção 8.5 os aspectos específicos da construção de sistemas operacionais para máquinas paralelas são apresentados. As seções 8.6, 8.7 e 8.8 apresentam respectivamente os sistemas Linux, Windows 2000 e variações de Linux para tempo real como estudos de caso. Finalmente, a seção 8.9 contém os comentários finais.

O texto contido neste capítulo foi construído a partir da experiência coletiva dos autores na área de sistemas operacionais. Versões preliminares de algumas seções foram publicadas antes. A revisão de programação concorrente da seção 8.2 está baseada no livro “Sistemas Operacionais”, dos mesmos autores [Oliveira et al. 2001], assim como os estudos de caso sobre Linux e Windows 2000. Uma versão preliminar da taxonomia apresentada na seção 8.3 foi publicada nos anais da 2ª Escola Regional de Alto Desempenho [Oliveira et al. 2002]. Finalmente, a seção 8.8 sobre adaptações de Linux para tempo real está baseada no livro “Sistemas de Tempo Real” [Farines et al. 2000], onde um dos autores deste capítulo aparece como co-autor.

8.2. Revisão de Programação Concorrente

Um programa que é executado por apenas um processo é chamado de **programa seqüencial**. A grande maioria dos programas escritos são programas seqüenciais. Nesse caso, existe somente um fluxo de controle durante a execução. Isso permite, por exemplo, que o programador realize uma "execução imaginária" de seu programa apontando com o dedo, a cada instante, a linha que está sendo executada no momento.

Um **programa concorrente** é executado simultaneamente por diversos processos que cooperam entre si, isto é, trocam informações. Para o programador realizar agora uma "execução imaginária", ele vai necessitar de vários dedos, um para cada processo que faz parte do programa. Nesse contexto, trocar informações significa trocar dados ou realizar algum tipo de sincronização. É necessária a existência de interação entre processos para que o programa seja considerado concorrente. Embora a interação entre processos possa ocorrer através do acesso a arquivos comuns, esse tipo de concorrência é tratada na disciplina de Banco de Dados. A programação concorrente tratada neste texto utiliza mecanismos rápidos para interação entre processos: variáveis compartilhadas e troca de mensagens.

O termo "programação concorrente" vem do inglês *concurrent programming*, onde *concurrent* significa "acontecendo ao mesmo tempo". Uma tradução mais exata seria programação concomitante. Entretanto, o termo programação concorrente já está estabelecido no Brasil, sendo algumas vezes usado o termo **programação paralela**.

O verbo "concorrer" admite em português vários sentidos. Pode ser usado no sentido de cooperar, como em "tudo concorria para o bom êxito da operação". Também pode ser usado com o significado de disputa ou competição, como em "ele concorreu a uma vaga na universidade". Em uma forma menos comum ele significa também existir simultaneamente. De certa forma, todos os sentidos são aplicáveis aqui na programação concorrente. Em geral, processos concorrem (disputam) pelos mesmos recursos do hardware e do sistema operacional. Por exemplo, processador, memória, periféricos, estruturas de dados, etc. Ao mesmo tempo, pela própria definição de programa concorrente, eles concorrem (cooperam) para o êxito do programa como um todo. Certamente, vários processos concorrem (existem simultaneamente) em um programa concorrente. Logo, programação concorrente é um bom nome para o que vamos tratar nesta seção. Maiores informações podem ser encontradas em [Oliveira et al. 2001].

8.2.1. Problema da Seção Crítica

Uma forma de implementar a passagem de dados são variáveis compartilhadas pelos processos envolvidos na comunicação. A passagem de dados acontece quando um

processo escreve em uma variável que será lida por outro processo. A quantidade exata de memória compartilhada entre os processos pode variar conforme o programa. Processos podem compartilhar todo o seu espaço de endereçamento, apenas um segmento de memória ou algumas variáveis.

No entanto, o compartilhamento de uma mesma região de memória por 2 ou mais processos pode causar problemas. A solução está em controlar o acesso dos processos a essas variáveis compartilhadas de modo a garantir que um processo não acesse uma estrutura de dados enquanto essa estiver sendo atualizada por outro processo. Os problemas desse tipo podem acontecer de maneira muito sutil.

Vamos chamar de **seção crítica** aquela parte do código de um processo que acessa uma estrutura de dados compartilhada. O problema da seção crítica está em garantir que, quando um processo está executando sua seção crítica, nenhum outro processo entre na sua respectiva seção crítica. Por exemplo, isso significa que, enquanto um processo estiver inserindo nomes em uma fila, outro processo não poderá retirar nomes da fila, e vice-versa.

Uma solução para o problema da seção crítica estará correta quando apresentar as seguintes quatro propriedades:

- A solução não depende das velocidades relativas dos processos;
- Quando um processo P deseja entrar na seção crítica e nenhum outro processo está executando a sua seção crítica, o processo P não é impedido de entrar;
- Nenhum processo pode ter seu ingresso na seção crítica postergado indefinidamente, ou seja, ficar esperando para sempre;
- Existe exclusividade mútua entre os processos com referência a execução das respectivas seções críticas.

Soluções erradas para o problema da seção crítica normalmente apresentam a possibilidade de postergação indefinida ou a possibilidade de *deadlock*. Ocorre **postergação indefinida** quando um processo está preso tentando entrar na seção crítica e nunca consegue por ser sempre preterido em benefício de outros processos. Ocorre ***deadlock*** quando dois ou mais processos estão à espera de um evento que nunca vai acontecer. Isto pode ocorrer porque cada um detém um recurso que o outro precisa, e solicita o recurso que o outro detém. Como nenhum dos dois vai liberar o recurso que possui antes de obter o outro, ambos ficarão bloqueados indefinidamente.

Uma solução simples para o problema da seção crítica é desabilitar interrupções. Toda vez que um processo vai acessar variáveis compartilhadas ele antes desabilita as interrupções. Dessa forma, ele pode acessar as variáveis com a certeza de que nenhum outro processo vai ganhar o processador. No final da seção crítica, ele torna a habilitar as interrupções. Esse esquema é efetivamente usado por alguns sistemas operacionais, porém apresenta, como veremos mais tarde, problemas em multiprocessadores.

Uma solução possível para o problema da seção crítica é o chamado ***Spin-Lock*** ou ***Test-and-Set.*** Essa solução é baseada em uma instrução de máquina chamada "*Test-and-Set*", embora uma instrução do tipo "*Swap*" ou "*Compare on Store*" possa ser usada também. Considere uma instrução de máquina que troca (*swap*) o valor contido em uma posição de memória com o valor contido em um registrador. A posição de memória e o registrador a serem utilizados são especificados como operandos da instrução. Em resumo, temos a Instrução SWAP(reg, mem):

```
[mem] → aux "copia conteúdo da posição [mem] para um reg. auxiliar"
reg → [mem]      "copia conteúdo de reg para a posição [mem]"
aux → reg  "copia conteúdo do reg. auxiliar para o registrador reg"
```

É essencial que o processador execute toda a instrução de máquina *SWAP* sem interrupções e sem perder o acesso exclusivo ao barramento do computador. Isso é normal em máquinas com apenas um processador, mas exige alguns cuidados especiais no hardware de máquinas com vários processadores acessando a mesma memória.

A seção crítica será protegida por uma variável que ocupará a posição [mem] da memória. Essa variável é normalmente chamada de *lock* (fechadura). Quando *lock* contém "0", a seção crítica está livre. Quando *lock* contém "1", ela está ocupada. A variável é inicializada com "0":

```
int lock = 0;
```

Antes de entrar na seção crítica, um processo precisa "fechar a porta", colocando "1" em *lock*. Entretanto, ele só pode fazer isso se "a porta estiver aberta". Logo, antes de entrar na seção crítica, o processo executa um procedimento equivalente ao seguinte código:

```
do {        reg = 1;
            swap( reg, lock);
            } while( reg == 1);
código-da-seção-crítica
```

Observe que o processo fica repetidamente colocando "1" em *lock* e lendo o valor que estava lá antes. Se esse valor era "1", a seção estava (e está) ocupada, e o processo repete a operação. Quando o valor lido de *lock* for "0", então a seção crítica está livre, o processo sai do do-while e prossegue sua execução dentro da seção crítica. Ao sair da seção crítica, basta colocar "0" na variável *lock*. Se houver algum processo esperando para entrar, a sua instrução *swap* pegará o valor "0", e ele entrará. Isso é denominado de *spin-lock.*

A vantagem do *Spin-Lock* é sua simplicidade, aliada ao fato de que não é necessário desabilitar interrupções. A instrução de máquina necessária está presente em praticamente todos os processadores atuais. Essa mesma solução funciona em máquinas com vários processadores, a partir de alguns cuidados na construção do hardware. Entretanto, *Spin-Lock* tem como desvantagem o *busy-waiting*. O processo que está no laço de espera executando "*swap*" ocupa o processador enquanto espera. Além disso, existe a possibilidade de postergação indefinida, quando vários processos estão esperando simultaneamente para ingressar na seção crítica e um processo "muito azarado" sempre perde na disputa de quem "pega antes" o valor "0" colocado na variável *lock*. Na prática o *spin-lock* é muito usado em situações nas quais a seção crítica é pequena (algumas poucas instruções). Nesse caso, a probabilidade de um processo encontrar a seção crítica ocupada é baixíssima, o que torna o *busy-waiting* e a postergação indefinida situações teoricamente possíveis, mas altamente improváveis.

8.2.2 Semáforos

Um mecanismo de sincronização entre processos muito empregado é o **semáforo**. Ele foi criado pelo matemático holandês E. W. Dijkstra em 1965. O semáforo é um tipo abstrato de dado composto por um valor inteiro e uma fila de processos. Somente duas

operações são permitidas sobre o semáforo. Elas são conhecidas como **P** (do holandês *proberen*, testar) e **V** (do holandês *verhogen*, incrementar).

Quando um processo executa a operação P sobre um semáforo, o seu valor inteiro é decrementado. Caso o novo valor do semáforo seja negativo, o processo é bloqueado e inserido no fim da fila desse semáforo. Quando um processo executa a operação V sobre um semáforo, o seu valor inteiro é incrementado. Caso exista algum processo bloqueado na fila desse semáforo, o primeiro processo da fila é liberado. Podemos sintetizar o funcionamento das operações P e V sobre o semáforo S da seguinte forma:

```
P(S):
      S.valor = S.valor - 1;
      Se  S.valor < 0
            Então bloqueia o processo, insere em S.fila
V(S):
      S.valor = S.valor + 1;
      Se S.valor <= 0
            Então retira processo P de S.fila, acorda P
```

Para que semáforos funcionem corretamente, é essencial que as operações P e V sejam atômicas. Isso é, uma operação P ou V não pode ser interrompida no meio e outra operação sobre o mesmo semáforo iniciada.

Semáforos tornam a proteção da seção crítica muito simples. Para cada estrutura de dados compartilhada, deve ser criado um semáforo S inicializado com o valor 1. Todo processo, antes de acessar essa estrutura, deve executar um P(S), ou seja, a operação P sobre o semáforo S associado com a estrutura de dados em questão. Ao sair da seção crítica, o processo executa V(S).

Uma variação muito comum de semáforos são as construções ***mutex*** ou **semáforo binário.** Nesse caso, temos um semáforo capaz de assumir apenas os valores 0 e 1. Ele pode ser visto como uma variável tipo mutex, a qual assume apenas os valores **livre** e **ocupado**. Nesse caso, as operações P e V são normalmente chamadas de ***lock*** e ***unlock***, respectivamente. Assim como P e V, elas devem ser atômicas. Sua operação é:

```
lock(x):
      Se x está livre
            Então       marca x como ocupado
            Senão       insere processo no fim da fila "x"
unlock(x):
      Se fila "x" está vazia
            Então       marca x como livre
            Senão       libera processo do início da fila "x"
```

O problema da seção crítica pode ser facilmente resolvido com o mutex:

```
      mutex x = LIVRE;
      ...
      lock(x);          /* entrada da seção crítica */
      Seção Crítica;
      unlock(x);        /* saída da seção crítica */
```

8.2.3. Threads

Um processo é uma abstração que reúne uma série de atributos como espaço de endereçamento, descritores de arquivos abertos, permissões de acesso, quotas, etc. Um processo possui ainda áreas de código, dados e pilha de execução. Também é associado ao processo um fluxo de execução. Por sua vez, uma ***thread*** nada mais é que um fluxo de execução. Na maior parte das vezes, cada processo é formado por um conjunto de recursos mais uma única *thread*.

A idéia de ***multithreading*** é associar vários fluxos de execução (várias *threads*) a um único processo. Em determinadas aplicações, é conveniente disparar várias *threads* dentro do mesmo processo (programação concorrente). É importante notar que as *threads* existem no interior de um processo, compartilhando entre elas os recursos do processo, como o espaço de endereçamento (código e dados). Devido a essa característica, a gerência de *threads* (criação, destruição, troca de contexto, sincronização) é "mais leve" quando comparada com processos. Por exemplo, criar um processo implica alocar e inicializar estruturas de dados no sistema operacional para representá-lo. Por outro lado, criar uma *thread* implica apenas definir uma pilha e um novo contexto de execução dentro de um processo já existente. O chaveamento entre duas *threads* de um mesmo processo é muito mais rápido que o chaveamento entre dois processos. Por exemplo, como todas as *threads* de uma mesmo processo compartilham o mesmo espaço de endereçamento, a MMU (*memory management unit*) não é afetada pelo chaveamento entre elas. Em função do exposto acima, *threads* são muitas vezes chamadas de **processos leves**.

Duas maneiras básicas podem ser utilizadas para implementar o conceito de *threads* em um sistema. Na primeira, o sistema operacional suporta apenas processos convencionais, isto é, processos com uma única *thread*. O conceito de *thread* é então implementado pelo próprio processo a partir de uma biblioteca ligada ao programa do usuário. Devido a essa característica, *threads* implementadas dessa forma são denominadas de ***threads* do nível do usuário** (*user-level threads*). No segundo caso, o sistema operacional suporta diretamente o conceito de *thread*. A gerência de fluxos de execução não é mais orientada a processos mas sim a *threads*. As *threads* que seguem esse modelo são ditas ***threads* do nível do sistema** (*kernel threads*).

O primeiro método é denominado N:1 (*many-to-one*). A principal vantagem é o fato de as *threads* serem implementadas em espaço de usuário, não exigindo assim nenhuma interação com o sistema operacional. Esse tipo de *thread* oferece um chaveamento de contexto mais rápido e menor custo para criação e destruição. A biblioteca de *threads* é responsável pelo compartilhamento, entre elas, do tempo alocado ao processo. O sistema operacional preocupa-se apenas em dividir o tempo do processador entre os diferentes processos. A grande desvantagem desse método é que as *threads* são efetivamente simuladas a partir de um único fluxo de execução pertencente a um processo convencional. Como conseqüência, qualquer paralelismo real disponível no computador não pode ser aproveitado pelo programa, embora permita vários processos diferentes executarem ao mesmo tempo. Outra conseqüência é que uma *thread* efetuando uma operação de entrada ou saída bloqueante provoca o bloqueio de todas as *threads* do seu processo. Existem técnicas de programação para evitar isso, mas são relativamente complexas.

O segundo método é dito 1:1 (*one-to-one*). Ele resolve os dois problemas mencionados acima: aproveitamento do paralelismo real dentro de um único programa e processamento junto com E/S. Para que isso seja possível, o sistema operacional deve ser projetado de forma a considerar a existência de *threads* dividindo o espaço de endereçamento do processo hospedeiro. A desvantagem desse método é que as operações relacionadas com as *threads* passam necessariamente por chamadas ao sistema operacional, o que torna *threads* tipo 1:1 "menos leves" que *threads* tipo N:1.

Existe um método misto que tenta combinar as duas abordagens, chamado M:N (*many-to-many*). Esse método possui escalonamento nos dois níveis. Uma biblioteca no espaço do usuário seleciona *threads* (M) do programa para serem executadas em uma ou mais *threads* do sistema (N). Embora flexível, esse método exige que o programador decida qual o valor de N e como será a execução das M *threads* do programa pelas N *threads* do sistema.

8.3. Organização de Sistemas Operacionais

Um sistema operacional também é um programa de computador e, como tal, possui uma especificação e um projeto. A especificação do mesmo corresponde à lista de serviços que deve fornecer e as chamadas de sistema que deve suportar. Por outro lado, o seu **projeto** ou ***design*** diz respeito à sua estrutura interna, como as diferentes rotinas necessárias na implementação dos serviços são organizadas internamente. O tamanho de um sistema operacional pode variar desde alguns milhares de linhas no caso de um pequeno núcleo para aplicações embutidas (*embedded*) até vários milhões de linhas, como na versão 2.4 do Linux, chegando a 30 milhões de linhas no caso do Windows 2000 [Stallings 2001]. Embora princípios básicos como baixo acoplamento e alta coesão [Pressman 2001] sejam sempre desejáveis, existem algumas formas de **organização interna** para sistemas operacionais que tornaram-se clássicas. Também ao longo do tempo a terminologia sofreu variações. A forma como os termos são apresentados neste texto procura criar uma taxonomia coerente e didática, mesmo que alguns autores, em alguns momentos, possam ter usados os termos com um sentido ligeiramente diferente.

Ao longo dos últimos 40 anos, sistemas operacionais têm crescido de tamanho e de complexidade. Conforme o histórico que aparece em [Stallings 2001], o CTSS, criado pelo MIT em 1963, ocupava aproximadamente 32 Kbytes de memória. O sistema OS/360, introduzido em 1964 pela IBM, incluia mais de 1 milhão de instruções de máquina. Ainda no início dos anos 70, o Multics, desenvolvido principalmente pelo MIT e pela Bell Labs, já possuia mais de 20 milhões de instruções de máquina. Atualmente mesmo microcomputadores pessoais executam sistemas operacionais com elevada complexidade. O Windows NT 4.0 possui 16 milhões de linhas de código e o Windows 2000 mais de 30 milhões de linhas de código. Desde o início, sistemas operacionais foram considerados um dos tipos de software mais difíceis de serem construídos. Por exemplo, em [Boehm 1981] o autor classifica os sistemas em 3 tipos com o propósito de estimar seus custos, e os sistemas operacionais são incluídos na classe mais complexa.

Ao longo dos anos, muito esforço foi colocado no sentido de criar as estruturas de software mais apropriadas para organizar internamente os sistemas operacionais. Existem grandes diferenças entre os serviços oferecidos por um pequeno sistema

operacional de propósito específico, embutido em um equipamento qualquer, e um sistema operacional de propósito geral como o Linux ou o Windows. Entretanto, as estruturas de software empregadas não variam tanto assim, mesmo entre sistemas cujo tamanho difere por 3 ordens de grandeza (milhares de linhas versus milhões de linhas). A taxonomia apresentada neste texto classifica a organização interna dos sistemas operacionais conforme a árvore mostrada na figura 8.2. No restante desta seção cada uma das organizações que aparecem na árvore serão caracterizadas e discutidas.

As recomendações da engenharia de software valem também nesta área. O desenvolvimento de um sistema operacional não deixa de ser um projeto de software e sofre dos mesmos problemas que aplicações em geral. Por exemplo, a criação do OS/360 nos anos 60 envolveu cerca de 5000 programadores, o que obviamente não tolera práticas ad-hoc de desenvolvimento. Propriedades como modularidade, interfaces claras e a mais simples possível entre os módulos, alta coesão e baixo acoplamento são muito bem vindas.

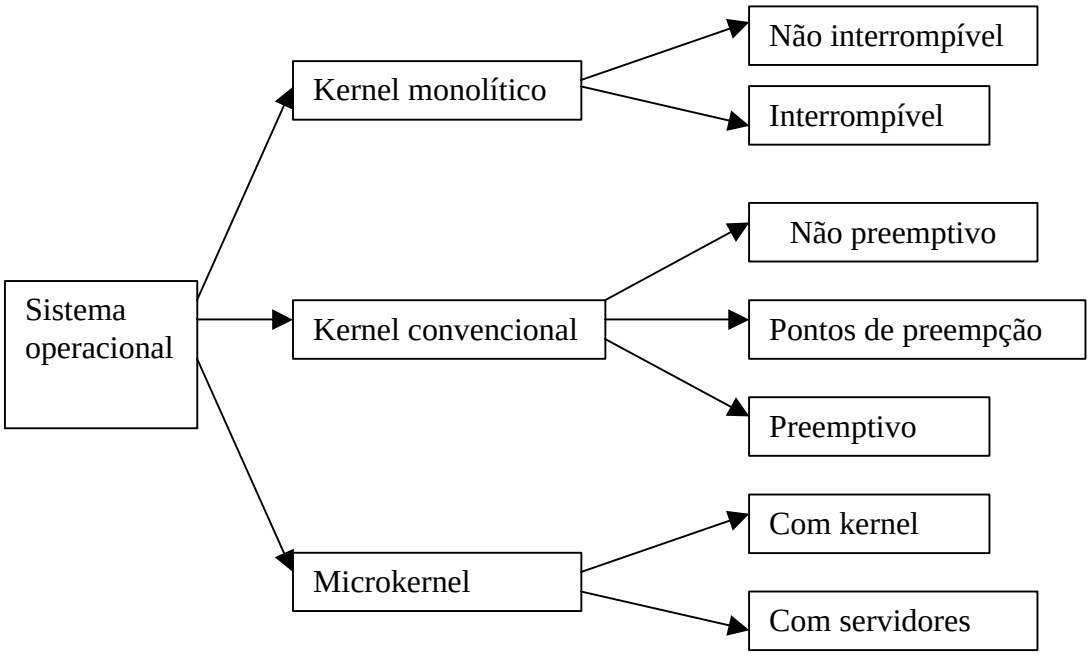


Figura 8.2. Taxonomia das organizações de sistemas operacionais.

8.3.1 Kernel Monolítico

A forma mais simples de organizar um sistema operacional é colocar toda a sua funcionalidade dentro de um único programa chamado kernel ou núcleo. O **kernel** inclui o código necessário para prover toda a funcionalidade do sistema operacional (escalonamento, gerência de memória, sistema de arquivos, protocolos de rede, *device-drivers*, etc). O kernel executa em modo supervisor e suporta o conjunto de chamadas de sistemas. Ele é carregado na inicialização do computador e permanece sempre na memória principal.

Internamente, o código do kernel é dividido em procedimentos os quais podem ser agrupados em módulos. De qualquer forma, tudo é ligado (*linked*) junto e qualquer

rotina pode, a princípio, chamar qualquer outra rotina. Todas as rotinas e estruturas de dados fazem parte de um único espaço lógico de endereçamento. O conceito de processo existe fora do kernel, mas não dentro dele. Apenas um fluxo de execução existe dentro do kernel. Vamos chamar este projeto de **kernel monolítico** (figura 8.3). Uma interrupção de hardware (periférico) ou de software (chamada de sistema) gera um chaveamento de contexto do processo que estava executando para o fluxo interno do kernel, o qual possui seu próprio espaço de endereçamento e uma pilha para as chamadas de subrotinas internas.

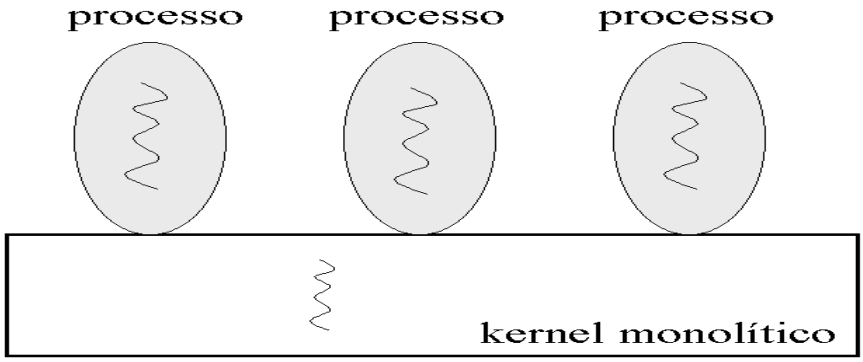


Figura 8.3. Kernel monolítico.

Duas melhorias podem ser feitas sem alterar a essência do kernel monolítico. Uma delas é a introdução de **programas de sistema**. Os programas de sistema são programas que executam fora do kernel (em modo usuário), fazem chamadas de sistema como programas normais, mas implementam funcionalidades típicas de sistemas operacionais, como listar os arquivos de um diretório ou gerenciar o compartilhamento de uma impressora (*spooler* de impressão). Uma vantagem de transferir parte da funcionalidade do kernel para os programas de sistema é a economia de memória. O kernel está sempre residente, enquanto programas de sistema são carregados para a memória somente quando necessário. Além disso, programas de sistema podem ser atualizados com facilidade, enquanto uma atualização do kernel exige uma reinstalação e reinicialização do sistema. Outra vantagem é com relação à questão da confiabilidade, pois uma falha em um programa de sistema dificilmente compromete os demais programas e o próprio kernel. Finalmente, programas de sistema geram uma modularização de qualidade que facilita o desenvolvimento e a manutenção do sistema.

Outra melhoria são os **módulos dinamicamente carregáveis**, isto é, permitir que algumas partes do kernel possam ser carregadas e removidas dinamicamente, sem interromper a execução do sistema. Tipicamente isto é feito para permitir que novos *device-drivers* sejam instalados sem a reinicialização do computador. Memória é alocada para o kernel, o código do *device-driver* é carregado para esta memória, o endereço de suas rotinas é incluído em tabelas do kernel, e uma ligação dinâmica entre o módulo carregado e o resto do kernel é realizada. Mais recentemente este tipo de mecanismo foi ampliado para que outras funcionalidades possam também ser carregadas dinamicamente, como novos sistemas de arquivos. Entretanto, a maior parte do kernel e toda a sua funcionalidade básica possui carga estática.

Existem duas variações de kernel monolítico no que diz respeito às interrupções de hardware. Na forma mais simples, o kernel executa com interrupções desabilitadas. O **kernel monolítico não-interrompível** possui código mais simples, pois enquanto o código do kernel está executando nada mais acontece, nem mesmo interrupções de periféricos como temporizadores e controladores de disco. Programas de usuário executam com interrupções habilitadas todo o tempo. O preço a ser pago por esta simplificação é uma redução no desempenho. Por exemplo, se o controlador do disco termina o acesso em andamento e gera uma interrupção enquanto o kernel está executando, o disco ficará parado até que o kernel termine sua execução, retorne a executar código de usuário e habilite as interrupções. Periféricos serão sub-utilizados, além de haver uma troca de contexto desnecessária, pois o processo de usuário ativado será imediatamente interrompido.

Uma forma mais eficiente é executar o código do kernel com interrupções habilitadas. O **kernel monolítico interrompível** possui desempenho melhor pois os eventos associados com periféricos e temporizadores ganham imediata atenção, mesmo quando o código do kernel está executando. Entretanto, é preciso notar que tratadores de interrupções podem acessar estruturas de dados do kernel, as quais podem estar inconsistentes enquanto uma chamada de sistema é atendida. Nessa situação, a execução do tratador de interrupção poderia corromper todo o sistema. Na construção de um kernel monolítico interrompível é necessário identificar todas as estruturas de dados acessadas por tratadores de interrupção e, quando o código normal do kernel acessa essas estruturas de dados, interrupções devem ser desabilitadas (pelo menos aquelas cujos tratadores acessem a estrutura em questão). As estruturas de dados acessadas por tratadores de interrupção formam uma **seção crítica** que deve ser protegida, e o mecanismo usado para isto é simplesmente desabilitar as interrupções enquanto estas estruturas de dados estiverem sendo acessadas. É preciso cuidado quando as seções com interrupções desabilitadas aparecem aninhadas, pois neste caso somente ao sair da seção mais externa é que as interrupções podem ser habilitadas novamente.

Imagine agora a situação onde o código de um kernel monolítico interrompível está executando em função da chamada de sistema de um processo de baixa prioridade. Neste momento ocorre uma interrupção de tempo (*timer tick*) e um processo de alta prioridade é liberado. No kernel monolítico interrompível o tratador da interrupção executa, mas ao seu término o código do kernel volta a executar em nome de um processo de baixa prioridade, mesmo que um processo de alta prioridade aguarde na fila do processador. Temos então que este tipo de kernel é intrinsicamente **não-preemptivo**, isto é, nenhum processo recebe o processador enquanto a linha de execução dentro do kernel não for concluída ou ficar bloqueada por alguma razão.

8.3.2 Kernel Convencional

Vamos chamar de **kernel convencional** (figura 8.4) aquele que, além de interrompível, permite uma troca de contexto mesmo quando código do kernel estiver executando. A passagem de um kernel monolítico interrompível para um kernel convencional possui várias implicações. Entre elas podemos destacar que agora o conceito de processo existe também dentro do kernel, uma vez que o processo pode ser suspenso e liberado mais tarde enquanto executa código do kernel. No kernel monolítico apenas uma pilha basta, pois a cada momento apenas um fluxo de execução existe dentro dele. Interrupções podem acontecer, mas ainda assim uma única pilha é suficiente. No kernel convencional

um processo pode passar o processador para outro processo que também vai executar código do kernel. Logo, é necessária uma pilha interna ao kernel para cada processo, além das pilhas em modo usuário.

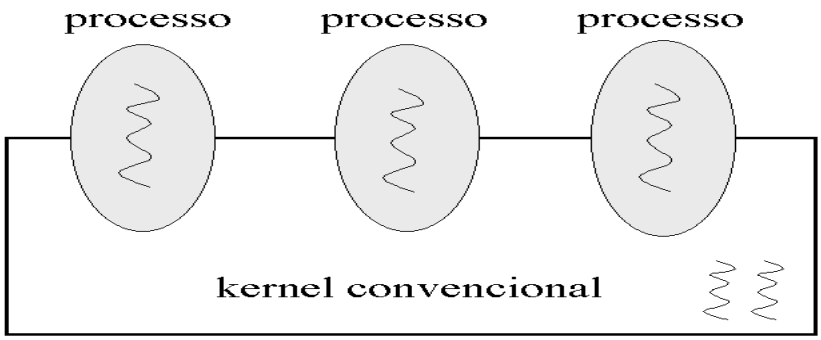


Figura 8.4. Kernel convencional.

Quando o processo executando código do usuário faz uma chamada de sistema, ocorre um chaveamento no modo de execução e no espaço de endereçamento, mas conceitualmente o mesmo processo continua executando. Apenas agora ele executa código do kernel. Este processo executa até que a chamada de sistema seja concluída. Nesse caso simplesmente é efetuado um retorno para o contexto do usuário. Também é possível que esse processo tenha que esperar por algum evento como, por exemplo, um acesso a disco. Nesse caso, ele fica bloqueado e um outro processo passa a ser executado. Como vários processos podem estar bloqueados simultaneamente dentro do kernel, deve existir no espaço de endereçamento do kernel uma pilha para cada processo. Observe que, neste tipo de kernel, o processo executando código do kernel não perde o processador para outro processo, a não ser que fique bloqueado, caracterizando assim um **kernel convencional não-preemptivo**.

No kernel convencional as interrupções de software e de exceção estão fortemente associadas com o processo em execução, representando uma chamada de sistema ou uma violação de algum tipo. Assim, as ativações dos tratadores desses dois tipos de interrupções são vistas apenas como uma nova fase na vida do processo. O mesmo não ocorre com as interrupções de hardware, as quais estão associadas, na maioria das vezes, com solicitações de entrada e saída feitas por outros processos.

No kernel monolítico, a chamada de sistema implica em chavear o modo de execução, o espaço de endereçamento e o próprio processo. No kernel convencional, a chamada de sistema implica tão somente no chaveamento do modo de execução e uma adaptação no espaço de endereçamento. Se acontecer um retorno imediato da chamada de sistema (por exemplo com gettime() ou read() de dados bufferizados), os outros dois tipos de chaveamento sequer acontecerão. O resultado final é a melhoria no desempenho do sistema, embora o código do kernel fique um pouco mais complexo. Em alguns sistemas esta solução é dita empregar um processo envelope, pois os processos "envelopam" (executam) tanto o programa usuário como o código do sistema operacional [Holt 1983]. É possível a existência de processos que jamais executam código fora do kernel, isto é, processos que executam tarefas auxiliares dentro do kernel e não estão associados com nenhum programa de usuário em particular. Este tipo de

processo pode ser usado para, por exemplo, escrever blocos de arquivos alterados da *cache* do sistemas de arquivos para o disco.

O código do kernel convencional pode ser visto como um conjunto de rotinas, possivelmente agrupadas em módulos, que estão à disposição dos processos. Obviamente os usuários estão sujeitos ao controle de acesso embutido no kernel, o qual é preservado pela MMU, que impede o acesso direto à memória do kernel por parte de programas de usuário. Muitas vezes o espaço de endereçamento de um processo inclui tanto os segmentos com código e dados do programa de usuário quanto o código e dados do kernel. Mecanismos de proteção associados com o modo de execução impedem que o código executado em modo usuário acesse diretamente os bytes pertencentes aos segmentos do sistema.

Quando o processo dentro do kernel deve ser bloqueado, ocorre um chaveamento entre processos. Algumas rotinas do kernel são responsáveis por este efeito e, quando elas executam, conceitualmente nenhum processo está executando. Temos a transição:

- Processo P1 executando código do kernel.
- Processo P1 chama rotina que executa chaveamento entre processos.
- Rotina de chaveamento salva todo o contexto do processo P1.
- Processo P1 transformou-se em um fluxo de execução interno ao kernel, não vinculado a nenhum processo, cujo único propósito é carregar o próximo processo a executar.
- Rotina de chaveamento de processos carrega todo o contexto do processo P2, o fluxo de execução transforma-se no processo P2.
- Processo P2 retoma a execução do ponto onde havia sido bloqueado.

A distinção entre kernel monolítico interrompível e kernel convencional não-preemptivo pode ser resumida pela execução do código do kernel estar associada com processos ou com um fluxo desvinculado do conceito de processo. Na prática isto pode ser determinado pelo número de pilhas internas ao kernel. Se existe somente uma pilha dentro do kernel, temos um kernel monolítico interrompível. Já o kernel convencional, mesmo não-preemptivo, exige uma pilha dentro do kernel para cada processo. Ao mesmo tempo, as rotinas do kernel convencional devem ser **re-entrantes**, isto é, devem ser programadas de tal forma que um processo possa iniciar a execução, ser suspenso por algum motivo, e a mesma rotina ser então executada por outro processo. O kernel agora corresponde realmente a um programa concorrente, algo que não acontecia com o kernel monolítico.

Como o kernel convencional é um programa concorrente, as suas estruturas de dados devem ser protegidas, pois formam seções críticas dentro do programa. A sincronização entre processos dentro do kernel é feita de várias formas. Primeiramente, no kernel convencional não-preemptivo o processo só libera o processador voluntariamente. Desta forma, a maioria das estruturas de dados não precisa ser protegida pois é garantido, pela disciplina de programação, que o processo fazendo o acesso deixará a estrutura em um estado consistente antes de liberar o processador. Quando, em função dos algoritmos usados, o processo necessita bloquear-se ainda com uma estrutura de dados em estado inconsistente, alguma primitiva de sincronização deve ser usada, como semáforos ou algo semelhante. Se outro processo chamar o kernel e tentar acessar esta estrutura de dados que está inconsistente, então ele ficará

bloqueado no semáforo. Finalmente, estruturas de dados também acessadas pelos tratadores de interrupção de hardware somente podem ser acessadas por um processo com as interrupções desabilitadas, como já acontecia antes no kernel monolítico.

Outra questão relevante, relacionada com as estruturas de dados internas ao kernel, leva ao surgimento de mais dois tipos de kernel convencionais. O **kernel convencional com pontos de preempção** somente suspende um processo que executa código do kernel em pontos previamente definidos do código, nos quais é sabido que nenhuma estrutura de dados está inconsistente. O desempenho deste tipo de kernel é superior ao do kernel monolítico, mas o processo de mais alta prioridade ainda é obrigado a esperar até que a execução do processo de mais baixa prioridade atinja um ponto de preempção. O sistema SVR4.2/MP funciona assim.

Por outro lado, o **kernel convencional preemptivo** realiza o chaveamento de contexto tão logo o processo de mais alta prioridade seja liberado. Para isto, todas as estruturas de dados do kernel que são compartilhadas entre processos devem ser protegidas por algum mecanismo de sincronização, como semáforos, mutexes, etc. Esta solução, usada no Solaris 2.x, resulta em melhor resposta do sistema aos eventos externos, e um comportamento mais coerente com as prioridades dos processos.

Uma vez que existe no kernel convencional uma estrutura disponível para a convivência de vários fluxos de execução simultâneos internos ao kernel, pode-se utilizar construções denominadas de *threads* do kernel. As ***threads* do kernel** são fluxos de execução que executam o tempo todo código do kernel e não possuem nenhum código de usuário associado. Elas executam tarefas auxiliares, como a manutenção dos níveis de memória disponível através da escrita de *buffers* para o disco ou implementam protocolos de comunicação em rede.

Alguns sistemas como o Solaris [Vahalia 1996] vão além, e transformam os tratadores de interrupção em *threads* de kernel. Dessa forma, a única coisa que o tratador de interrupção propriamente dito faz é liberar a *thread* de kernel correspondente, inserindo-a na fila do processador. O código executado pela *thread* é que realiza efetivamente o tratamento da interrupção. A sincronização com respeito às estruturas de dados do kernel é feita normalmente, pois tanto processos normais como *threads* de kernel podem ser bloqueados se isto for necessário. O código fica mais simples e robusto, pois agora não é necessário determinar exatamente qual estrutura de dados é acessada por qual tratador de interrupção, para desabilitar as interrupções corretamente durante o acesso. Simplesmente cada estrutura de dados compartilhada é protegida por um semáforo ou mecanismo semelhante e o problema está resolvido. A experiência com o Solaris também mostra que não existe redução de desempenho quando este esquema é utilizado, pois um conjunto de *threads* pode ser previamente criado, para evitar o custo da criação a cada interrupção. Permanece apenas o custo do chaveamento de contexto.

8.3.3. Microkernel

Recentemente uma organização tem sido proposta na qual a funcionalidade típica de kernel é dividida em duas grandes camadas. Esta solução baseia-se na existência de um **microkernel**, o qual suporta os serviços mais elementares de um sistema operacional: gerência de processador e uma gerência de memória simples. A idéia de microkernel foi popularizada pelo sistema operacional Mach [Tanenbaum 1995].

Na sua forma mais simples, o microkernel corresponde à funcionalidade de mais baixo nível do kernel convencional preemptivo, aquela responsável por realizar o chaveamento dos processos envelopes. O microkernel suporta um kernel semelhante ao convencional preemptivo, separado da funcionalidade inserida no microkernel. Esta solução é ilustrada pela figura 8.5 e será chamada de **microkernel com kernel**.

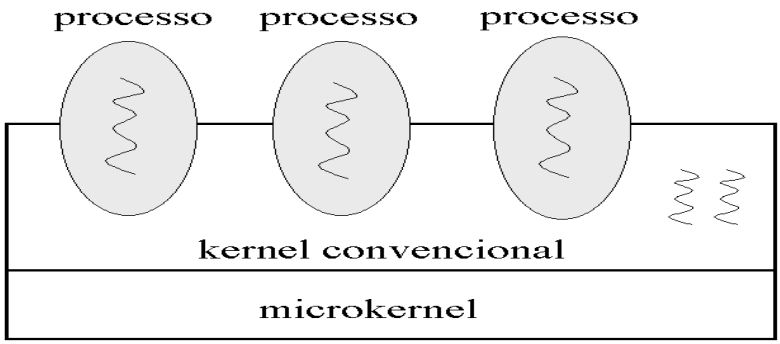


Figura 8.5. Microkernel com kernel.

Um passo adiante na idéia de microkernel é criar um conjunto de processos servidores autônomos, tal que as chamadas de serviço dos processos de usuário sejam feitas diretamente para eles, sem passar por uma camada que uniformize a interface do kernel. Nesta solução temos um microkernel que suporta um conjunto de processos com espaços de endereçamento independentes. Alguns processos executam programas de usuário enquanto outros implementam serviços do sistema operacional, tais como o sistema de arquivos e o gerenciador de memória virtual. Quando o processo do usuário necessita ler um arquivo ele envia uma mensagem para o processo "sistema de arquivos", o qual executa o pedido e retorna outra mensagem com o resultado. A participação do microkernel resume-se a dividir o tempo do processador entre os diversos processos e prover o mecanismo de comunicação entre eles.

Internamente, o microkernel tem a forma de um pequeno kernel monolítico. As desvantagens do kernel monolítico não são tão sérias neste caso pois o código do microkernel é pequeno. Por sua vez, cada processo servidor constitui um espaço de endereçamento independente e pode ser composto por várias *threads* para melhorar o seu desempenho. Vamos chamar esta organização de **microkernel com processos servidores** (figura 8.6).

Os processos servidores que necessitam acesso especial podem executar em modo supervisor. O modo de execução passa a ser uma característica do processo, definida pelo administrador do sistema. Modos de execução intermediários podem ser utilizados se estiverem disponíveis. O microkernel sempre executa em modo supervisor e pode ser visto como um componente monolítico. Um sistema operacional bastante conhecido que trabalha dessa forma é o Minix [Tanenbaum e Woodhull 2000].

A existência de processos servidores representa uma decomposição funcional que facilita o desenvolvimento do sistema operacional nos aspectos de codificação, teste e manutenção. As vantagens da organização baseada em microkernel com processos servidores são semelhantes àquelas dos programas de sistema: facilidade para atualizações, isolamento entre os diferentes componentes, depuração e manutenção mais

fácil, modularização superior, facilidade para distribuição e tolerância a falhas. Também agora temos o microkernel e os processos servidores executando em espaço de endereçamento diferentes, o que é ótimo do ponto de vista da engenharia de software, embora represente uma penalidade no tempo de execução. Isto resulta em facilidade para desenvolver e testar novos processos servidores. Também o sistema como um todo responde melhor às definições de prioridades feitas pela administração, pois a preempção ocorre mais facilmente.

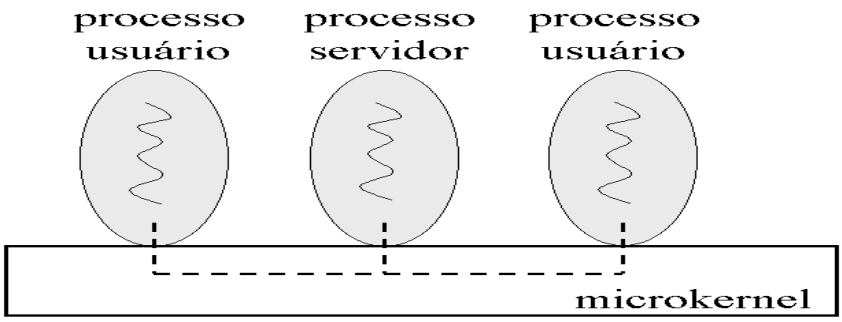


Figura 8.6. Microkernel com processos servidores.

A grande motivação para empregar uma organização como esta é a modularidade e flexibilidade resultantes. É possível configurar cada instalação com apenas os processos servidores desejados, depois atualizá-los sem a necessidade de reinicializar o sistema. É especialmente atraente em sistemas distribuídos, pois os mecanismos utilizados para implementar serviços distribuídos são complexos e, se colocados todos em um kernel convencional, o tornariam intratável. Nesta organização pode-se com mais facilidade distribuir os processos por diferentes computadores, já que não é suposta a existência de memória compartilhada entre os processos servidores e os processos usuários, e ainda assim obter elevado nível de transparência. Apenas o microkernel deve executar em todos os computadores.

Uma das mais notáveis capacidades dos sistemas baseados em microkernel é suportar simultaneamente várias API (*Application Program Interface*) de diferentes sistemas operacionais. Por exemplo, é possível colocar sobre um mesmo microkernel processos servidores que suportem as chamadas da API Win32 (Microsoft) e outros processos servidores que suportem a API Posix (IEEE). Embora todo este código de sistema operacional represente uma carga considerável para o computador, em determinados ambientes esta possibilidade é atrativa. Esta capacidade pode também ser implementada sobre outros tipos de organização, mas não com a mesma facilidade.

A maior desvantagem dessa solução é o custo muito mais alto associado com uma chamada de sistema, que agora implica em vários chaveamentos sucessivos. Além disso, o programa de usuário deve montar e desmontar mensagens ao solicitar serviços para outros processos, pois mensagens formam a base da comunicação neste tipo de organização. Cada passagem de nível (processo de usuário para microkernel para processo de sistema e tudo de volta) implica no gasto de tempo do processador. Por exemplo, a comunicação entre sistema de arquivos e gerência de memória era feita antes com uma chamada de subrotina, mas agora exige uma chamada ao microkernel.

Quanto menor a funcionalidade do microkernel, maior a penalidade no desempenho. Assim, uma tendência atual tem sido a re-incorporação de servidores importantes ao microkernel. Ele passa a ser um pouco “menos micro”, perdendo em parte a flexibilidade, mas o desempenho melhora. Isto é conseguido através da redução do número de chaveamentos de modo de execução e de espaços de endereçamento no atendimento das chamadas de sistema. Os serviços que são os maiores candidatos a serem incluídos no microkernel são a gerência de memória e os *device-drivers*, pelo menos aqueles associados com os principais periféricos do sistema.

Esta questão do desempenho resultou em variações na literatura a respeito de onde está exatamente a fronteira entre um kernel e um microkernel. Por exemplo, quais os serviços que devem ser implementados ou qual o tamanho máximo que um microkernel pode ter. Entretanto, é mais ou menos consenso que, minimamente, o microkernel deve oferecer processos (*threads*), comunicação entre processos, facilidades para gerência de E/S e gerência de memória física (de baixo nível).

O mecanismo tipicamente usado entre processos acima do microkernel é a troca de mensagens. Uma das motivações básicas dessa organização é a independência entre os processos servidores, e a definição de variáveis compartilhadas prejudicaria esse propósito. O microkernel suporta primitivas elementares de envio e recepção de mensagens, as quais são usadas pelos processos de usuário para solicitar serviços aos processos servidores e por estes para enviar as respectivas respostas. As variações típicas dos mecanismos de mensagens, como endereçamento direto ou indireto, mensagens tipadas ou não, grau de bufferização, semântica das primitivas, também são encontradas aqui. Para que o microkernel cumpra seu papel de isolar os diferentes processos, ele deve ser capaz de controlar os direitos de acesso e a integridade das mensagens. Para evitar que o envio de uma mensagem implique na cópia memória-memória, operação tipicamente lenta, mecanismos como **remapeamento de páginas** podem ser usados. Neste caso, a mensagem não é realmente copiada, mas a página física que ela ocupa é mapeada na tabela de páginas do processo destinatário.

Uma questão importante é como o microkernel deve tratar a ocorrência de uma interrupção. Algumas interrupções, como as dos temporizadores do hardware (*timers*) e aquelas associadas com erros de execução (acesso ilegal a memória, instrução ilegal, etc) possuem um tratamento óbvio. A questão refere-se às interrupções geradas por periféricos cuja gerência é realizada por processos servidores que executam fora do microkernel. Uma solução simples é permitir que processos servidores instalem suas rotinas de tratamento de interrupções. Uma chamada de microkernel permite ao processo servidor informar que, na ocorrência de interrupção do tipo “x”, a rotina “y” do processo servidor deve ser chamada. Este esquema pode ser usado em sistemas menores, mas possui várias desvantagens. Por exemplo, se memória virtual for empregada é possível que a rotina em questão sequer esteja residente na memória principal. Além disso, a programação do processo servidor fica mais complexa, pois além das mensagens dos processos clientes ele deve se preocupar agora com rotinas executadas automaticamente. Entretanto, este tipo de programação é comum no mundo Unix, quando o mecanismo de sinais (*signals*) é utilizado.

Uma solução mais harmoniosa com a organização geral do sistema é fazer o microkernel transformar as interrupções em mensagens. Inicialmente, o processo servidor interessado nas interrupções do tipo “x” informa isto ao microkernel. Sempre

que ocorrer uma interrupção deste tipo, o microkernel gera automaticamente uma mensagem para o processo servidor em questão, informando que uma interrupção daquele tipo aconteceu. A programação do processo servidor é simplificada, pois ele agora sempre recebe mensagens, seja dos demais processos, seja do dispositivo de hardware que ele controla. Obviamente, existe uma penalidade em termos de desempenho na montagem e envio dessa mensagem, se comparada com a solução onde uma rotina do processo servidor é chamada diretamente.

O aspecto mais complexo no projeto de um microkernel está exatamente na gerência da memória. Para que os processos executando acima do microkernel possuam espaços de endereçamento independentes, o microkernel deve criar e manter as respectivas tabelas de páginas e de segmentos, conforme a arquitetura do processador. Ele também deve gerenciar quais páginas físicas estão livres na memória. Entretanto, o mecanismo de memória virtual envolve interações com o disco e deveria, a princípio, ficar fora do microkernel. Quando um processo executa um acesso ilegal a memória o microkernel pode recorrer a um processo servidor para realizar o tratamento da falta de página, com possível acesso a disco e escolha de página vítima. Entretanto, a API do microkernel para este processo servidor exige um projeto cuidadoso, ou sérias penalidades no desempenho acontecerão.

Sistemas baseados em microkernel podem variar com respeito aos modos de execução. A maioria dos processadores oferecem pelo menos dois modos de execução: supervisor e usuário. No caso de mais de dois vamos denominar os demais como modos de execução intermediários, com mais direitos que o modo usuário mas nem todos os direitos do modo supervisor. Por exemplo, o Intel Pentium oferece 4 níveis de proteção: o nível 0 corresponde ao modo supervisor, nível 3 é o modo usuário típico e os níveis 1 e 2 oferecem direitos intermediários.

Se o sistema está baseado em microkernel, naturalmente o código de usuário executa em modo usuário e o código do microkernel executa em modo supervisor. Entretanto, o código dos servidores permite várias possibilidades. Buscando segurança é possível executar o código dos servidores em modo usuário, obrigando a realização de uma chamada ao microkernel toda vez que uma operação restrita for necessária. Esta solução gera vários chaveamentos de modo de operação e a execução de código extra pelo microkernel para testar a validade das operações solicitadas. Com o propósito de tornar o sistema mais rápido, é possível executar o código dos servidores em modo supervisor. O sistema fica mais simples e rápido, porém agora qualquer servidor pode corromper todo o sistema e a proteção que havia antes, mesmo entre servidores, é perdida. Idealmente o código do sistema operacional não deveria conter erros, mas a nossa experiência diária mostra que isto não é sempre verdade. Por fim, modos de execução intermediários podem ser usados para os servidores, buscando uma solução de compromisso. Por exemplo, permitir que processos servidores acessem alguns endereços de entrada e saída previamente estabelecidos, o que não é possível para processos que executam em modo usuário.

Sistemas baseados em microkernel também podem variar com respeito aos espaços de endereçamento. Logicamente, o código de usuário executa em espaço de endereçamento próprio, o mesmo acontecendo com o código do microkernel. Entretanto, o código de cada servidor pode executar em espaço de endereçamento independente ou ser ligado ao código do microkernel, ocupando portanto o mesmo

espaço de endereçamento. Novamente temos uma questão de isolamento (proteção) versus desempenho. O chaveamento entre espaços de endereçamento exige comandos para a MMU (*memory management unit*) e também esvaziamento de sua *cache* interna, a TLB (*translation lookaside buffer*). Ao juntar microkernel e servidores no mesmo espaço de endereçamento, não é necessário chavear o contexto da MMU quando o servidor faz uma solicitação de serviço ao microkernel. Entretanto, o isolamento já não é imposto pela MMU, o que diminui o isolamento entre os diferentes componentes do sistema operacional. Também fica mais complexa a carga dinâmica de servidores.

Concluindo, sistemas do tipo microkernel com kernel podem ser considerados uma simplificação da idéia geral de microkernel. Eles têm uma organização modular, com interfaces claras entre os seus componentes. Dados essenciais são encapsulados e somente podem ser acessados através das rotinas apropriadas. Entretando, vários servidores importantes executam em modo supervisor no mesmo espaço de endereçamento, formando o kernel sobre o microkernel. Esta organização busca melhorar o desempenho do sistema ao reduzir a necessidade de chaveamentos de contextos e modos de execução. Existe aqui portanto uma variação da idéia de microkernel, onde todos os processos servidores são agrupados em um componente de software, o qual adiciona uma interface para as chamadas de sistema. Infelizmente, a terminologia, que já varia no mundo acadêmico, varia ainda mais no mundo comercial. Como os departamentos de marketing das empresas determinaram que “microkernel é bom”, agora todos os desenvolvedores de sistemas operacionais tentam mostrar que, de alguma forma, o seu sistema também é baseado em microkernel.

8.3.4. Comparação Entre as Organizações

As técnicas apresentadas nas seções anteriores são as fundamentais, mas nada impede que um sistema operacional seja construído misturando vários aspectos aqui descritos. Por exemplo, o conceito de processo servidor pode ser usado sobre um kernel monolítico para suportar determinados serviços com tempo de resposta mais dilatado. O microkernel em si pode ser visto como um kernel monolítico com pouquíssimos serviços. Cada sistema operacional existente utiliza uma mistura dessas técnicas, em geral aproximando-se mais de uma delas. As estruturas de software apresentadas neste texto devem ser entendidas como os modelos abstratos (as receitas de bolo) utilizados na construção dos sistemas operacionais, embora cada sistema (cada bolo) tenha sempre as suas peculiaridades e características próprias. Como regra geral, sistemas acadêmicos buscam a elegância e a qualidade no projeto, e portanto tendem a usar microkernel. Por outro lado, sistemas comerciais buscam principalmente desempenho, usando por isto soluções baseadas em kernel convencional. De qualquer forma, todos os sistemas apresentam uma mistura das formas básicas, repletas de detalhes específicos, que vão além dos objetivos deste texto.

Como em qualquer tipo de software, a chave para entender a organização interna de um sistema operacional está na compreensão de como ele foi decomposto em diversas partes. A decomposição pode ser analisada segundo diferentes perspectivas, mas quatro delas destacam-se, para os propósitos deste texto.

Primeiramente temos a decomposição do código em **rotinas e módulos**. Isto é feito para encapsular variáveis locais, tanto a nível de rotina como de módulo, e também rotinas internas ao seu módulo, as quais não podem ser chamadas de outros módulos.

Orientação a objetos vai nessa mesma direção, embora ainda não seja empregada completamente na construção de kernel. Este particionamento é efetuado pelo programador e imposto pelo compilador e pelo ligador, durante a tradução do código fonte do sistema operacional. A divisão rígida do sistema em camadas não é normalmente suportada por compiladores e ligadores, dependendo da sua adoção como disciplina de projeto e de programação.

Uma decomposição óbvia está relacionada com os direitos de acesso de cada parte, através dos **modos de execução do processador**. Vamos aqui considerar a existência de apenas dois modos de execução (supervisor e usuário), embora outros intermediários também possam existir. Este particionamento é decidido em projeto e implementado pela unidade de controle do processador.

O sistema operacional também pode ser decomposto em vários **espaços de endereçamento**. Neste caso, a visibilidade das rotinas e estruturas de dados do sistema operacional pelas outras partes do mesmo pode ser limitada através da criação de vários espaços de endereçamento independentes. Este particionamento é efetuado pela MMU.

Finalmente, temos a decomposição da execução do sistema operacional em vários **fluxos de execução** separados, os quais provavelmente colaboram entre si, tornando o sistema operacional um programa concorrente. O termo fluxo de execução é usado para designar *threads*, processos, ou quaisquer construções semelhantes. Este particionamento é efetuado por uma decisão de projeto e disciplina de programação.

Uma vez definidas as quatro perspectivas da decomposição de um sistema operacional que serão analisadas aqui, podemos investigar como cada tipo de organização decompõe o sistema operacional nessas perspectivas. Embora o texto tenha indicado a existência de variações a partir das organizações básicas, a análise considera apenas as 3 principais formas de organização: kernel monolítico interrompível, kernel convencional preemptivo e microkernel com processos servidores.

A **decomposição em rotinas e módulos** pode ser aplicada a qualquer forma de organização. A modularização é um dos pilares da computação e pode ser aplicada a qualquer tipo de software. Até mesmo no microkernel, embora pequeno, a modularização vai melhorar a legibilidade e robustez, facilitando testes e manutenção.

A **decomposição em modos de execução** é bem característica em cada forma de organização:

- No kernel monolítico interrompível os processos de usuário executam em modo usuário, enquanto o kernel executa em modo supervisor.
- No kernel convencional preemptivo o processo executa código de usuário em modo usuário e, após uma chamada de sistema, executa código do kernel em modo supervisor. Os tratadores de interrupção fazem parte do kernel e executam em modo supervisor. Durante o chaveamento entre processos são executadas instruções fora do contexto de qualquer processo. Este pequeno fluxo de execução interno ao kernel também executa em modo supervisor.
- No microkernel com processos servidores os processos de usuário executam código de usuário em modo usuário e o microkernel executa em modo supervisor. Os processos servidores podem executar em modo usuário, modo

supervisor ou algum modo intermediário, dependendo do tipo de serviço prestado e do projeto do sistema.

A **decomposição em espaços de endereçamento** também caracteriza com força as diferentes organizações:

- No kernel monolítico interrompível cada processo de usuário possui um espaço de endereçamento próprio, enquanto o kernel executa em seu próprio espaço de endereçamento, o qual permite acesso a toda a memória.
- No kernel convencional preemptivo existem algumas possibilidades. Em essência cada processo possui um espaço de endereçamento quando executa código de usuário e passa para outro espaço de endereçamento quando executa código do kernel. Entretanto, uma única tabela de páginas e/ou segmentos pode ser usada para os dois espaços de endereçamento, sendo a distinção obtida através do modo de execução, ou seja, as partes da memória associadas com o kernel somente podem ser acessadas quando o processo executa em modo supervisor, isto é, executa código do kernel. Neste caso, uma chamada de sistema tem o efeito de mudar o processador para modo supervisor, liberar as partes do kernel nas tabelas de memória e desviar a execução para o código do kernel. Tratadores de interrupção também executam neste espaço de endereçamento ampliado.
- No microkernel com processos servidores cada processo, seja de usuário seja servidor, executa em um espaço de endereçamento próprio. O microkernel também possui seu próprio espaço de endereçamento, permitindo acesso a toda a memória.

A **decomposição em fluxos de execução** também caracteriza muito bem as diferentes soluções de organização usadas em sistemas operacionais:

- No kernel monolítico interrompível cada processo de usuário corresponde a um fluxo de execução. Se *threads* são suportadas, cada processo de usuário pode conter vários fluxos de execução. O kernel, por sua vez, é composto por um único fluxo de execução principal, associado com uma chamada de sistema.
- No kernel convencional preemptivo cada processo está associado com um fluxo de execução. Se *threads* são suportadas então cada processo estará associado com vários fluxos de execução. Estes fluxos executam tanto código de usuário quanto o código do kernel. Durante o chaveamento de contexto entre processos existe um momento no qual a execução está desvinculada de qualquer processo e pode ser pensada como um fluxo do kernel que aparece somente neste momento. Finalmente, tratadores de interrupção representam fluxos disparados por interrupções do hardware e encerrados ao término da rotina de tratamento.
- O microkernel com processos servidores tipicamente suporta *threads*. Logo, cada processo, seja de usuário ou servidor, pode estar associado com vários fluxos de execução. O microkernel pode ser implementado de várias formas, mas na mais simples ele seria apenas um núcleo monolítico não interrompível, caracterizado portanto por apenas um fluxo de execução. Na verdade esta descrição pode tornar-se recursiva, na medida que aplicarmos também ao microkernel as várias soluções possíveis descritas para o kernel principal.

Para completar a comparação entre as 3 soluções básicas, vamos agora considerar o que acontece quando um programa de usuário faz uma chamada de sistema read() para ler dados de um arquivo previamente aberto. Na maioria das linguagens de programação o programador chama uma rotina da biblioteca da linguagem que, por sua vez, chama o sistema operacional. Em função do propósito deste texto, vamos ignorar a biblioteca da linguagem e iniciar a análise a partir da chamada ao sistema operacional propriamente dita. Também será suposto que este read() não pode ser satisfeito com dados na memória principal e um acesso ao disco será necessário.

No kernel monolítico interrompível o processo deve colocar os parâmetros nos locais apropriados e executar a interrupção de software associada com chamadas de sistema. Em função da interrupção de software a execução desvia para o código do kernel, executando em modo supervisor e com um novo espaço de endereçamento. Inicialmente o contexto de execução do processo chamador é salvo. É feito o processamento da chamada até o ponto no qual o pedido de acesso a disco foi enfileirado e nada mais resta a fazer. Neste momento um outro processo é selecionado para execução. Mais tarde uma interrupção do controlador do disco vai sinalizar a conclusão da leitura. O tratador desta interrupção vai liberar o processo chamador, que volta a disputar o processador. É sem dúvida a organização mais simples. Sua limitação está em permitir que um processo de baixa prioridade monopolize o processador enquanto executa código do kernel.

No kernel convencional preemptivo o processo também deve colocar os parâmetros nos locais apropriados e executar a interrupção de software associada com chamadas de sistema. Em função da interrupção de software a execução desvia para o código do kernel, executando em modo supervisor e com um espaço de endereçamento ampliado. Entretanto, conceitualmente, o mesmo processo está executando. É feito o processamento da chamada até o ponto no qual o pedido de acesso a disco foi enfileirado e nada mais resta a fazer. Neste momento o processo declara-se bloqueado e solicita um chaveamento de processo. Seu contexto é salvo e o contexto de outro processo da fila de aptos é carregado. Observe que durante a transição a execução prossegue sem estar conceitualmente associada com nenhum processo em particular. Esta transição é crítica e acontece com interrupções desabilitadas. Mais tarde, a interrupção do controlador do disco sinaliza a conclusão da operação e o seu tratador vai liberar o processo chamador, colocando-o novamente na fila de aptos. Depois de algum tempo ele é eleito para execução, a qual é retomada dentro do kernel, no ponto imediatamente posterior àquele onde ele solicitou o seu bloqueio. O processamento da chamada de sistema é concluído e este processo finalmente retorna para o código de usuário, restaurando modo de execução e espaço de endereçamento.

Nesta organização, caso uma interrupção de hardware libere um processo de mais alta prioridade, o mesmo retoma sua execução, mesmo que outro processo de mais baixa prioridade esteja executando código do kernel. Esta solução melhora o comportamento do sistema, se comparada com o kernel monolítico, pois ela respeita mais as diferentes prioridades dos processos. Entretanto, sua programação é mais complexa, pois agora todo o kernel tornou-se um enorme programa concorrente.

No microkernel com processos servidores o processo de usuário monta uma mensagem com a requisição de um `read()` e seus parâmetros. Ele então solicita ao microkernel que envie esta mensagem para o processo servidor "sistema de arquivos"

através do serviço `send()`. Isto implica em uma interrupção de software e a correspondente execução do código do microkernel. O microkernel verifica a integridade e validade da mensagem e a insere na fila de mensagens endereçadas para o processo servidor. Provavelmente o processo servidor "sistema de arquivos" possui uma prioridade maior que processos de usuário e, caso ele esteja bloqueado por uma chamada `receive()` anterior, ele será liberado. Neste caso o `send()` do processo usuário acaba resultando em sua preempção por processo de mais alta prioridade. Seja como for a mensagem será transferida para o processo servidor onde provavelmente uma de suas *threads* atenderá a requisição. Possivelmente o *device-driver* do disco em questão é implementado por outro processo servidor, e a mesma seqüência de eventos acontecerá novamente, agora com o servidor "sistema de arquivos" no papel de cliente do servidor "*device-driver* do disco". As respostas para as requisições implicam novamente em mensagens, dessa vez fazendo o caminho inverso. No final temos 3 processos envolvidos, 2 envios de mensagem com requisição e 2 envios de mensagem com resposta, o que implica em 8 chamadas ao microkernel (4 *send* e 4 *receive*). Este tipo de organização exige cuidado na implementação para que o desempenho não fique muito abaixo daquele obtido com um kernel convencional.

8.3.5. Outras Formas de Organização

De maneira ortogonal às organizações fundamentais apresentadas na seção anterior, existem técnicas que podem ser aplicadas no sentido de minimizar a complexidade do projeto ou permitir a adaptação das organizações básicas a contextos diferentes. Nesse sentido, esta seção discute a organização em camadas e o emprego de máquinas virtuais. A adaptação do kernel para máquinas multiprocessadoras será discutida na seção 8.5.

Uma das primeiras técnicas, além da modularidade, usadas para lidar com a complexidade dos sistemas operacionais foi a **organização hierárquica**. Nesta estrutura o sistema é dividido em camadas. Cada **camada** utiliza os serviços da camada inferior e cria novos serviços para a camada superior. Idealmente, a implementação de uma camada pode ser livremente substituída por outra, desde que a nova implementação suporte a mesma interface e ofereça os mesmos serviços. Esta idéia apareceu pela primeira vez no sistema operacional THE (Technische Hogeschool Eindhoven), criado por Dijkstra no final dos anos 60 [Dijkstra 1968]. Mais tarde esta mesma solução foi usada pela ISO (International Organization for Standardization) na famosa organização em 7 camadas do seu modelo de referência para redes de computadores, o OSI - *Open Systems Interconnection*.

O sistema operacional THE foi organizado em 5 camadas. Na camada 0 ficava o hardware. Na camada 1 ficava o escalonamento do processador e a criação do conceito de processo. A camada 2 correspondia à gerência de memória e era responsável pela implementação de memória virtual. A camada 3 continha o *device-driver* para o console do operador. A camada 4 implementava o sistema de *buffers* para os demais dispositivos de entrada e saída e, por estar na camada 4, podia usar a memória virtual implementada pela camada 3 e enviar mensagens para o operador pela camada 2. Finalmente, programas de usuário executavam na camada 5.

Em geral não é fácil compor uma pilha de camadas de tal sorte que cada camada utilize somente as camadas inferiores. Veja, por exemplo, o relacionamento entre sistema de arquivos e gerência de memória. O sistema de arquivos faz chamadas para a

gerência de memória solicitando páginas físicas para implementar um esquema de *cache* para os arquivos. Por outro lado, a gerência de memória pode solicitar uma leitura de arquivo quando ocorrer uma falta de página. A estrutura em camadas que o sistema deve ter para satisfazer a todas as necessidades não é óbvia.

Paralelamente à organização em camadas, o conceito de **máquina virtual** é por vezes usado na construção de sistemas operacionais. No início da década de 70 a IBM oferecia um produto chamado CP (*Control Program*) que criava, sobre o hardware do computador IBM 370, a ilusão de várias máquinas virtuais [Silberschatz e Galvin 1999]. Sobre cada máquina virtual era possível executar um sistema operacional completo. Embora a execução de vários sistemas operacionais simultaneamente no mesmo computador represente uma carga considerável para o hardware, existem situações onde essa solução é atraente para o usuário. Por exemplo, um *call-center* onde atendentes devem resolver problemas apontados por usuários que estão executando a aplicação em questão em diferentes versões de sistema operacional.

8.4. Sistemas Operacionais de Tempo Real

Sistemas computacionais de tempo real são definidos como aqueles submetidos a requisitos de natureza temporal [Farines et al. 2000]. Nesses sistemas, os resultados devem estar corretos não somente do ponto de vista lógico, mas também devem ser gerados no momento correto. Os aspectos temporais não estão limitados a uma questão de maior ou menor desempenho, mas estão diretamente associados com a funcionalidade do sistema.

Na medida que o uso de sistemas computacionais prolifera em nossa sociedade, aplicações com requisitos de tempo real tornam-se cada vez mais comuns. Essas aplicações variam muito com relação ao tamanho, complexidade e criticalidade. Entre os sistemas mais simples estão os controladores embutidos em utilidades domésticas, tais como lavadoras de roupa e videocassetes. Na outra extremidade deste espectro estão os sistemas militares de defesa e o controle de tráfego aéreo. Exemplos de aplicações críticas são os sistemas responsáveis pela monitorização de pacientes em hospitais e os sistemas embarcados em veículos, de automóveis até aviões e sondas espaciais. Entre aplicações não críticas estão os videogames e as aplicações multimídia em geral. No contexto da automação industrial, são muitas as possibilidades (ou necessidades) de empregar sistemas com requisitos de tempo real [Rembold et al. 1993]. Exemplos são os sistemas de controle embutidos em equipamentos industriais e os sistemas de supervisão e controle de células de manufatura.

Nos sistemas tempo real críticos (*hard real-time*) o não atendimento de um requisito temporal pode resultar em conseqüências catastróficas tanto no sentido econômico quanto em vidas humanas. Para sistemas deste tipo é necessária uma análise de escalonabilidade em tempo de projeto (*off-line*). Esta análise procura determinar se o sistema vai ou não atender os requisitos temporais mesmo em um cenário de pior caso, quando as demandas por recursos computacionais são maiores. Quando os requisitos temporais não são críticos (*soft real-time*) eles descrevem o comportamento desejado. O não atendimento de tais requisitos reduz a utilidade da aplicação mas não resulta em conseqüências catastróficas.

Uma das crenças mais comuns é que o problema de tempo real se resolve pelo aumento da velocidade computacional. A rapidez de cálculo visa melhorar o

desempenho de um sistema computacional, minimizando o tempo de resposta médio de um conjunto de tarefas. O objetivo de um cálculo em tempo real é o atendimento dos requisitos temporais de cada uma das atividades de processamento caracterizadas nesses sistemas [Stankovic 1988]. Ter um tempo de resposta curto, não dá nenhuma garantia que os requisitos temporais de cada processamento no sistema serão atendidos. Mais do que a rapidez de cálculo, para os sistemas de tempo real, importa o conceito de **previsibilidade**.

Um sistema de tempo real é dito ser **previsível** (*predictable*) no domínio lógico e no domínio temporal quando, independentemente de variações ocorrendo à nível de hardware, da carga e de falhas, o comportamento do sistema pode ser antecipado. A noção de previsibilidade pode ser associada a uma antecipação determinista do comportamento temporal do sistema. Ou seja, o sistema é previsível quando podemos antecipar que todos os prazos colocados a partir das interações com o seu ambiente serão atendidos. Na literatura, o conceito de previsibilidade também aparece associado com uma antecipação probabilista do comportamento do sistema, baseada em estimativas ou simulações que estipulam as probabilidades dos prazos serem atendidos.

Sistemas operacionais de propósito geral (SOPG) encontram dificuldades em atender as demandas específicas das aplicações de tempo real. Fundamentalmente, SOPG são construídos com o objetivo de apresentar um bom comportamento médio, ao mesmo tempo que distribuem os recursos do sistema de forma eqüitativa entre os processos e os usuários. Existe pouca preocupação com previsibilidade temporal. Mecanismos como *cache* de disco, memória virtual, fatias de tempo do processador, etc, melhoram o desempenho médio do sistema mas tornam mais difícil fazer afirmações sobre o comportamento de um processo em particular frente às restrições temporais. Aplicações com restrições de tempo real estão menos interessadas em uma distribuição uniforme dos recursos e mais interessadas em atender requisitos tais como períodos de ativação e deadlines.

Neste contexto é importante notar a diferença entre plataforma alvo (*target system*) e plataforma de desenvolvimento (*host system*). A plataforma alvo inclui o hardware e o sistema operacional de tempo real (SOTR) onde a aplicação vai executar quando concluída. Por exemplo, pode ser o computador embutido (*embedded*) em um telefone celular. A plataforma de desenvolvimento inclui o hardware e o SO onde o desenvolvimento é feito, isto é, onde as ferramentas de desenvolvimento executam. Normalmente trata-se de um computador pessoal executando um sistema operacional de propósito geral (SOPG). Um SOPG neste caso permite um melhor e mais completo ambiente de desenvolvimento, pois tipicamente possui mais recursos do que a plataforma alvo (que tal desenvolver software no próprio telefone celular ?). Entretanto, a depuração exige o SOTR e as características da plataforma alvo.

Como qualquer sistema operacional, um sistema operacional de tempo real (SOTR) procura tornar a utilização do computador mais eficiente e mais conveniente. Alguns serviços são fundamentais: processos, mecanismos para a comunicação e sincronização, instalação de tratadores de dispositivos e a disponibilidade de temporizadores. A maioria das aplicações tempo real possui uma parte (talvez a maior parte) de suas funções sem restrições temporais. Logo, é preciso considerar que um SOTR deveria, além de satisfazer as necessidades dos processos de tempo real, fornecer funcionalidade apropriada para os processos convencionais, tais como sistema de

arquivos, interface gráfica de usuário e protocolos de comunicação para a Internet. Em [Timmeman et al. 1998] é apresentado um programa de avaliação de SOTR independente de fornecedor que define vários requisitos.

Aspectos temporais estão relacionados com a capacidade do SOTR fornecer os mecanismos e as propriedades necessários para o atendimento dos requisitos temporais da aplicação tempo real. Uma vez que tanto a aplicação como o SOTR compartilham os mesmos recursos do hardware, o comportamento temporal do SOTR afeta o comportamento temporal da aplicação. Por exemplo, considere a rotina do sistema operacional que trata as interrupções do temporizador em hardware (*timer*). O projetista da aplicação pode ignorar completamente a função desta rotina, mas talvez não possa ignorar o seu efeito temporal, isto é, a interferência que ela causa no tempo de execução da aplicação.

O fator mais importante a vincular aplicação e sistema operacional são os serviços que este último presta. A simples operação de solicitar um serviço ao sistema operacional através de uma chamada de sistema significa que: (1) o processador será ocupado pelo código do sistema operacional durante a execução da chamada de sistema e, portanto, não poderá executar código da aplicação; (2) a capacidade da aplicação atender aos seus deadlines passa a depender da capacidade do sistema operacional em fornecer o serviço solicitado em um tempo que não inviabilize aqueles deadlines.

Com respeito ao comportamento temporal do sistema, qualquer análise deve considerar conjuntamente aplicação e sistema operacional, pois os requisitos temporais que um SOTR deve atender estão completamente atrelados aos requisitos temporais da aplicação tempo real que ele deverá suportar. Uma vez que existe um amplo espectro de aplicações de tempo real, também existirão diversas soluções possíveis para a construção de SOTR, cada uma mais apropriada para um determinado contexto. Por exemplo, o comportamento temporal exigido de um SOTR capaz de suportar o controle de vôo em um avião (*fly-by-wire*) é muito diferente daquele esperado de um SOTR usado para videoconferência.

Ainda existe uma distância entre a teoria de escalonamento e a prática no desenvolvimento de sistemas de tempo real [Farines et al. 2000]. De um lado a teoria buscando a previsibilidade, de outro a prática fazendo "o que é possível" nos ambientes computacionais existentes, muitas vezes confundindo tempo real com alto desempenho. No meio disto temos os sistemas operacionais de tempo real, lentamente evoluindo do conceito "desempenho" para o conceito "previsibilidade".

8.4.1 Influência do Kernel no Tempo de Resposta das Tarefas

Diversos mecanismos populares em sistemas operacionais de propósito geral são problemáticos quando as aplicações possuem requisitos temporais. Por exemplo, o mecanismo de memória virtual é capaz de gerar grandes atrasos (envolve acesso a disco) durante a execução de um processo. Mecanismos tradicionais como ordenar a fila do disco para diminuir o tempo médio de acesso, fazem com que o tempo para acessar um arquivo possa variar muito. Em geral, aplicações de tempo real procuram minimizar o efeito negativo de tais mecanismos de duas formas: desativando o mecanismo sempre que possível (ex: não usar memória virtual) ou usando o mecanismo apenas em

processos sem requisitos temporais rigorosos (ex: acesso a disco feito por processos auxiliares).

Todos os sistemas operacionais desenvolvidos ou adaptados para tempo real mostram grande preocupação com a divisão do tempo do processador entre as tarefas. Entretanto, o processador é apenas um recurso do sistema. Memória, periféricos, controladores também deveriam ser escalonados visando atender os requisitos temporais da aplicação. Entretanto, muitos sistemas ignoram isto e tratam os demais recursos da mesma maneira empregada por um sistema operacional de propósito geral, isto é, tarefas são atendidas pela ordem de chegada.

Fornecedores de SOTR costumam divulgar métricas para mostrar como o sistema em questão é mais ou menos apropriado para suportar aplicações de tempo real. Uma métrica muito utilizada é o **tempo para chaveamento entre dois processos**. Outra métrica freqüentemente utilizada é a **latência até o inicio de um tratador de interrupção** do hardware. Imagina-se que eventos urgentes no sistema serão sinalizados por interrupções de hardware e, desta forma, é importante iniciar rapidamente o tratamento destas interrupções. Este tempo aumenta quando interrupções permanecem desabilitadas por muito tempo. A figura 8.7 ilustra este conceito.

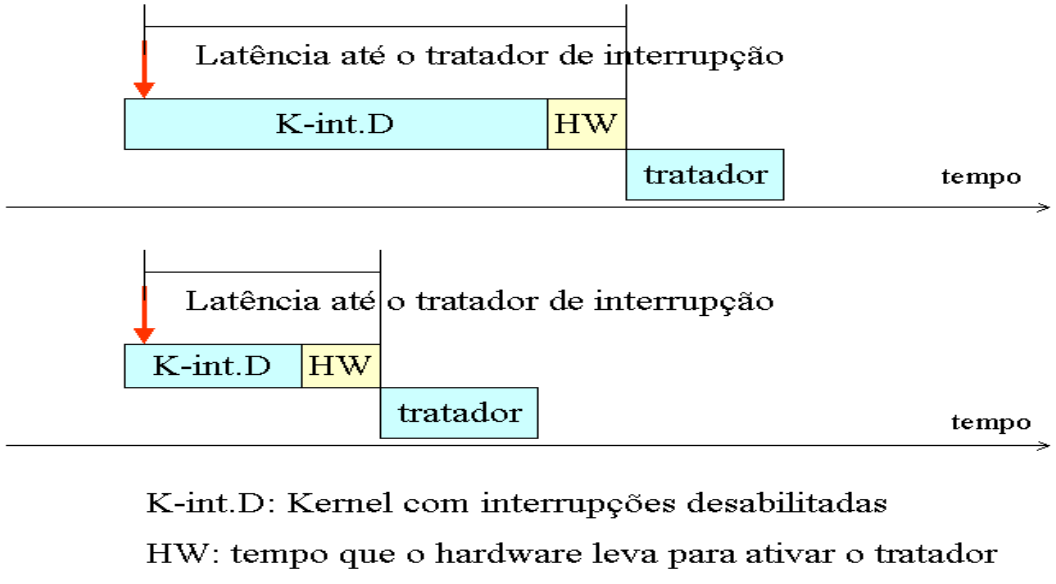


Figura 8.7. Latência até o atendimento de uma interrupção.

Métricas como o tempo de chaveamento e o tempo de latência são úteis no sentido de quanto menor elas forem em um dado SOTR, tanto melhor. Elas também são relativamente fáceis de medir. Entretanto, elas não são as únicas responsáveis pelos tempos de resposta. Também é importante lembrar que as diversas tarefas e interrupções do sistema causam interferências que devem ser contabilizadas. Por sua vez, os conflitos decorrentes de recursos compartilhados, tanto a nível de aplicação como a nível de kernel causam situações de bloqueio e de inversão de prioridades que também contribuem para os tempos de resposta no sistema [Farines et al. 2000].

Na maioria das vezes os requsitos de tempo real aparecem na forma de um prazo (deadline) dentro do qual determinada rotina ou conjunto de rotinas devem ser

executadas. Estas rotinas correspondem à reação do sistema a um evento externo, gerado pelo ambiente. Este evento externo, por exemplo um alarme ou a simples passagem do tempo, é tipicamente sinalizado por interrupções de hardware. Muitas aplicações de tempo real são compostas por tarefas periódicas, que a cada período devem executar determinada ação, como ler um sensor, apresentar um *frame* na tela ou calcular o algoritmo de um laço de controle realimentado. No caso das tarefas periódicas, é a interrupção do *timer* que marca o início da contagem de tempo associado com o deadline.

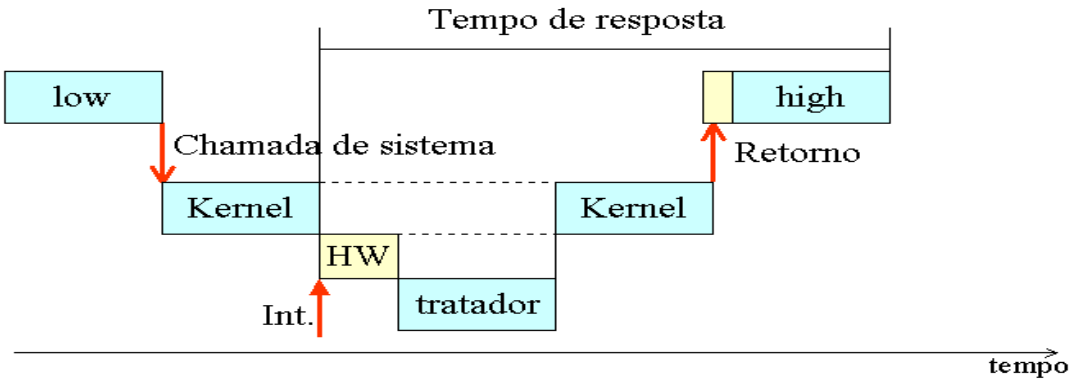
Neste contexto, uma questão importante é determinar de que maneiras o kernel impacta o tempo de resposta das tarefas de tempo real. Em outras palavras, quais são as contribuições do sistema operacional para o tempo total que a rotina demora para ser concluída. Os próximos parágrafos identificam várias dessas contribuições.

Para as aplicações de tempo real o tempo de execução no pior caso é tão ou mais relevante do que o tempo de execução no caso médio. Em geral, a implementação das chamadas de sistema é feita de maneira a minimizar o tempo médio. As aplicações de tempo real são beneficiadas quando o código que implementa as chamadas de sistema apresenta bom comportamento também no pior caso e não apenas no caso médio.

Por vezes o kernel executa trechos de código com as interrupções desabilitadas. Quando isto acontece, existe um atraso no reconhecimento da interrupção de hardware que sinaliza o evento externo que dispara a execução da tarefa de tempo real em questão.

Muitos sistemas operacionais de propósito geral manipulam por conta própria as prioridades dos processos. Por exemplo, o mecanimso de envelhecimento (*aging*) é usado por vezes para aumentar temporariamente a prioridade de um processo que, por ter prioridade muito baixa, nunca consegue executar. Este mecanismo faz sentido em SOPG, quando o objetivo é estabelecer um certo grau de justiça entre processos e evitar postergação indefinida. Ocorre que no contexto de tempo real não existe preocupação com justiça na distribuição dos recursos mas sim com o atendimento dos deadlines. Quando um processo anima o logotipo da empresa na tela do computador e outro abre uma válvula que impede o forno de explodir, deixar o sistema operacional mexer por conta própria nas prioridades dos processos não é conveniente. O mesmo pode ser dito para quando o kernel temporariamente eleva a prioridade de um processo que acabou de ficar bloqueado em função de operação de entrada ou saída.

Um kernel não preemptivo é capaz de gerar grandes inversões de prioridade. Suponha que um processo de baixa prioridade faça uma chamada de sistema e, enquanto o código do kernel é executado, ocorre a interrupção de hardware que deveria liberar um processo de alta prioridade. Nesse tipo de kernel o processo de alta prioridade terá que esperar até que a chamada de sistema do processo de baixa prioridade termine, para então executar. Temos então que o kernel está executando antes o pedido de baixa prioridade, em detrimento do processo de alta prioridade. Um kernel com pontos de preempção reduz o problema, mas ainda permite a inversão de prioridades, quando a chamada de sistema de baixa prioridade prossegue sua execução em detrimento do processo de alta prioridade até encontrar o próximo ponto de preempção. A figura 8.8 ilustra a forma como o kernel não preemptivo penaliza o processo de alta prioridade. A figura 8.9 mostra como o chaveamento para o processo de prioridade mais alta ocorre antes no caso de um kernel preemptivo.



low: processo com baixa prioridade

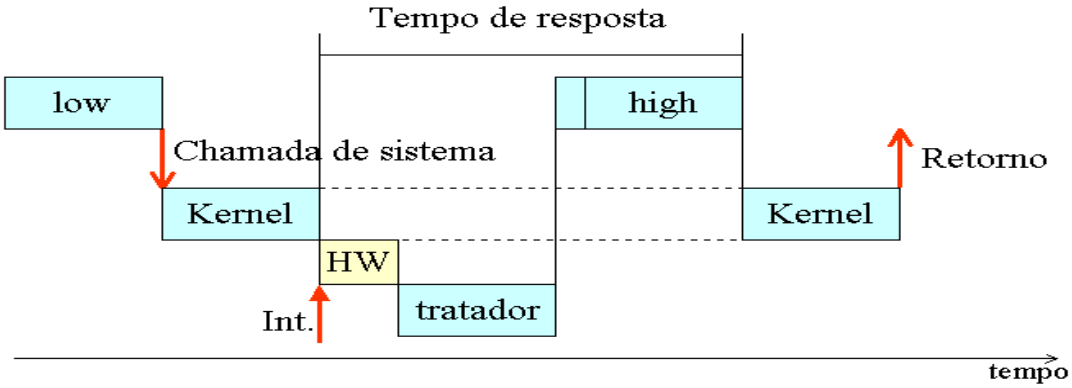
high: processo com alta prioridade

Kernel: Não preemptivo, high espera conclusão da chamada

HW: tempo que o hardware leva para ativar o tratador

tratador: que libera processo high

Figura 8.8. Chavemanto no kernel não preemptivo.



low: processo com baixa prioridade

high: processo com alta prioridade

Kernel: Preemptivo, high executa imediatamente

HW: tempo que o hardware leva para ativar o tratador

tratador: que libera processo high

Figura 8.9. Chaveamento no kernel preemptivo.

Um kernel preemptivo é capaz de chavear imediatamente para o processo de alta prioridade quando o mesmo é ativado. Entretanto, inversão de prioridade dentro do kernel ainda é possível quando o processo de alta prioridade recém ativado faz uma chamada de sistema mas é bloqueado em função da necessidade de acessar uma estrutura de dados compartilhada que foi anteriormente alocada por um processo de mais baixa prioridade. Manter uma granularidade fina para as seções críticas dentro do kernel, embora aumente a complexidade do código, reduz o tempo que um processo de

alta prioridade precisa esperar até que um processo de baixa prioridade libere uma estrutura de dados compartilhada.

É preciso lembrar que os processos disputam vários outros recursos além do processador. Por exemplo, controlador de disco, controlador de rede, acesso a tela, etc. Idealmente todas as filas do sistema deveriam respeitar as prioridades dos processos, e não apenas a fila do processador. Por exemplo, do ponto de vista dos deadlines a serem cumpridos, as requisições de disco deveriam ser ordenadas conforme a prioridade e não pela ordem de chegada ou para minimizar o tempo de acesso. É claro que isto poderá fazer com que o tempo médio de acesso ao disco aumente, mas em sistemas de tempo real a ênfase não está no tempo médio mas sim no tempo de pior caso para os processos com prioridade mais alta.

Por vezes a execução de um processo de alta prioridade é suspensa temporariamente, quando ocorre uma interrupção de periférico. O processador passa então a executar o tratador de interrupção incluído em *device-driver* associado com o periférico em questão. Muitos sistemas não limitam o tempo de execução desse tratador, permitindo na prática que o código associado com o *device-driver* em questão tenha uma prioridade maior do que qualquer processo no sistema e execute por quanto tempo quiser. Em um SOTR ideal, os tratadores de interrupção associados com *device-drivers* simplesmente liberariam *threads* de kernel as quais seriam responsáveis pela execução do código que efetivamente responde ao sinal do periférico. Fazendo com que as *threads* de kernel respeitem a estrutura de prioridades do sistema fica restaurado o desejo do programador com respeito aos tempos de resposta dos processos. Por exemplo, o teclado pode ser considerado um periférico de baixa prioridade. Caso aconteça uma interrupção de teclado durante a execução de processo de alta prioridade, o tratador de interrupções do teclado simplesmente coloca a *thread* “Atende Teclado” na fila de aptos e retorna. O processo de alta prioridade conclui a sua execução e somente depois disso o atendimento ao teclado propriamente dito será feito pela correspondente *thread* de kernel.

Muitos sistemas operacionais incluem *threads* de kernel com execução periódica, responsáveis pelas tarefas de manutenção. Por exemplo, escrever página vítima da memória virtual para o disco, escrever partes da *cache* do sistema de arquivos para o disco, atualizar contabilizações, etc. É importante que essas *threads* de kernel façam parte do esquema global de prioridades. Dado o caráter essencial de algumas dessas atividades, pode ser necessário que elas possuam prioridade superior aos processos da aplicação. Isto não é um grande mal, desde que fique bem caracterizado quais são essas tarefas, com o período, o tempo de computação e a prioridade de cada uma. Essas informações permitirão ao projetista da aplicação avaliar o impacto da execução autônoma do kernel nos seus tempos de resposta.

Duas fontes de atraso para o término de uma tarefa de tempo real não foram listadas acima por estarem associadas com a própria construção da aplicação e não com uma limitação do sistema operacional. São elas a interferência e o bloqueio entre tarefas da própria aplicação.

Quando a aplicação de tempo real é composta por várias tarefas, com prioridades diferentes, uma tarefa de baixa prioridade pode ser atrasada em função da execução de uma tarefa de mais alta prioridade. Neste caso é dito que a tarefa de mais alta prioridade interfere com a de baixa prioridade. A interferência nesse caso não é uma

limitação do sistema operacional mas apenas uma conseqüência natural da atribuição de prioridades feita pelo programador da aplicação.

Quando a aplicação de tempo real é composta por várias tarefas que compartilham estruturas de dados, é possível que uma tarefa fique bloqueada em função de necessitar uma estrutura de dados em uso por outra tarefa da aplicação. Novamente, temos um atraso causado pela própria estrutura da aplicação e não por ação do sistema operacional.

Pelo exposto acima, fica claro que a organização interna do kernel tem um importante impacto sobre o tempo de resposta dos processos. Em particular, pode-se listar algumas propriedades desejadas em um SOTR:

- Chamadas de sistema com bom comportamento mesmo no pior caso;
- Manter interrupções habilitadas tanto quanto possível;
- Não alterar por conta própria as prioridades dos processos;
- Kernel preemptivo;
- Granularidade pequena no controle de exclusão mútua dentro do kernel;
- Todas as filas devem respeitar as prioridades e não apenas a fila de aptos;
- Tratadores de interrupção de *device-drivers* não são privilegiados na execução;
- *Threads* de manutenção do kernel são descritas e respeitam as prioridades.

8.4.2 Tipos de Suportes para Tempo Real

A diversidade de aplicações de tempo real gera uma diversidade de necessidades com respeito ao suporte para tempo real, a qual resulta em um leque de soluções com respeito aos suportes disponíveis, com diferentes tamanhos e funcionalidades. De uma maneira simplificada podemos classificar os suportes de tempo real em dois tipos: núcleos de tempo real (NTR) e sistemas operacionais de tempo real (SOTR). O NTR consiste de um pequeno microkernel com funcionalidade mínima mas excelente comportamento temporal. Seria a escolha indicada para, por exemplo, o controlador de uma máquina industrial. O SOTR é um sistema operacional com a funcionalidade típica de propósito geral, mas cujo kernel foi adaptado para melhorar o comportamento temporal. A qualidade temporal do kernel adaptado varia de sistema para sistema, pois enquanto alguns são completamente re-escritos para tempo real, outros recebem apenas algumas poucas otimizações. Por exemplo, o sistema operacional Solaris implementa a funcionalidade Unix, mas foi projetado para fornecer uma boa resposta temporal. Uma descrição detalhada do escalonamento em várias versões do Unix pode ser encontrada em [Vahalia 1996].

A figura 8.10 procura resumir os tipos de suportes encontrados na prática. Esta é uma classificação subjetiva, mas permite entender o cenário atual. Além do NTR e do SOTR descritos antes, existem outras duas combinações de funcionalidade e comportamento temporal. Obter funcionalidade mínima com pouca previsibilidade temporal é trivial, qualquer núcleo oferece isto. Por outro lado, obter previsibilidade temporal determinista em um sistema operacional completo é muito difícil e ainda

objeto de estudo pelos pesquisadores das duas áreas. Embora não seja usual atualmente, é razoável supor que existirão sistemas deste tipo no futuro.

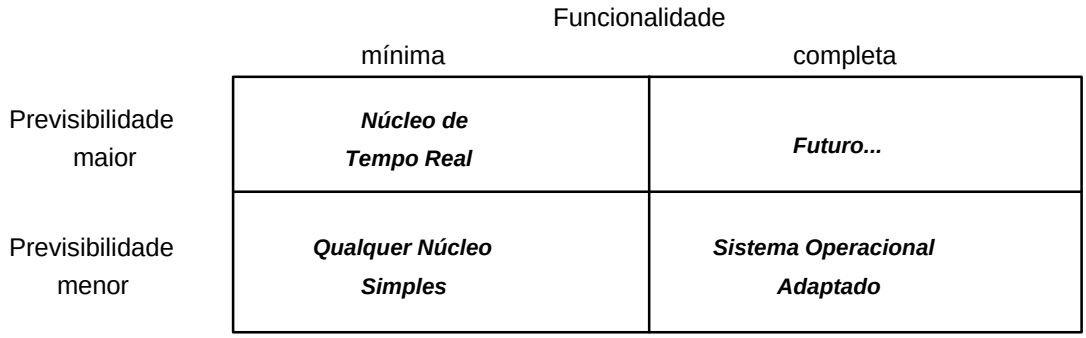


Figura 8.10. Tipos de suportes para aplicações de tempo real.

Na verdade é muito difícil comparar diferentes SOTR com a finalidade de escolher aquele mais apropriado para um dado projeto. Este texto procurou mostrar os diversos fatores que influenciam a previsibilidade temporal de um sistema. Entre os fatores que tornam a escolha difícil podemos citar:

- Diferentes SOTR possuem diferentes abordagens de escalonamento;
- Desenvolvedores de SOTR publicam métricas diferentes;
- Desenvolvedores de SOTR não publicam todas as métricas e todos os dados necessários para uma análise detalhada, detalhes internos do kernel não estão normalmente disponíveis;
- As métricas fornecidas foram obtidas em plataformas diferentes;
- O conjunto de ferramentas para desenvolvimento de aplicações que é suportado varia muito;
- Ferramentas para monitoração e depuração das aplicações variam;
- As linguagens de programação suportadas em cada SOTR são diferentes;
- O conjunto de periféricos suportados pelos SOTR variam;
- O conjunto de plataformas de hardware suportadas varia em função do SOTR;
- Cada SOTR possui um esquema próprio para a incorporação de *device-drivers* e o esforço necessário para desenvolver novos *device-drivers* varia entre sistemas;
- Diferentes SOTR possuem diferentes níveis de conformidade com os padrões;
- A política de licenciamento e o custo associado variam muito conforme o fornecedor do SOTR (desde custo zero até *royaltes* por cópia do produto final).

No mercado de sistemas operacionais para tempo real o Posix ganha cada vez maior destaque. Posix é um padrão para sistemas operacionais, baseado no Unix, criado pela IEEE (Institute of Electrical and Eletronic Engineers). Posix define as interfaces do sistema operacional mas não sua implementação. Logo, é possível falar em Posix API (*Application Program Interface*). Muitos SOTR atualmente já suportam a API do Posix, como pode ser constatado através de uma visita às páginas listadas em http://www.cs.bu.edu/pub/ieee-rts. Uma descrição completa do Posix é capaz de ocupar um livro inteiro [Gallmeister 1995].

A maioria dos sistemas operacionais implementam chamada de sistema através de interrupção de software. O Posix padroniza a sintaxe das rotinas de biblioteca que por sua vez executam as interrupções de software que são a verdadeira interface do kernel. Isto é necessário porque a forma de gerar interrupções de software depende do processador em questão e não pode ser realmente padronizada, ao passo que rotinas de biblioteca podem. Além da sintaxe em C, a semântica das chamadas Posix é definida em linguagem natural. Como parte da semântica, são definidas abstrações e uma terminologia própria.

O padrão Posix é na verdade dividido em diversos componentes. Em sua primeira versão o padrão definia apenas a API de um sistema operacional de propósito geral. Entretanto novos elementos foram incluídos com o passar do tempo, incluindo a interface para serviços que são importantes no contexto de tempo real. Muitos sistemas operacionais de tempo real possuem uma API proprietária. Neste caso, a aplicação fica amarrada aos conceitos e às primitivas do sistema em questão. Um porte da aplicação para outro sistema operacional é dificultado pela imcompatibilidade não somente das chamadas e parâmetros, mas das próprias abstrações suportadas.

8.5. Sistemas Operacionais para Máquinas Paralelas

Tradicionalmente, um computador é visto como uma máquina seqüencial onde o programador, com o auxílio de uma linguagem de programação, descreve um algoritmo como uma seqüência de instruções. Na última década, o decrescente custo de processadores, aliado ao desempenho cada vez maior desses, levou os projetistas de sistemas a considerarem cada vez mais a possibilidade de criar máquinas paralelas. Essa nova categoria de máquinas apresenta necessidades próprias a nível de software, ou seja, do sistema operacional. Nesta seção nós abordaremos justamente esse aspecto. Inicialmente nós situaremos os diferentes tipos de máquinas paralelas para em seguida apresentar aspectos específicos de sistemas operacionais para os multiprocessadores.

8.5.1. Arquitetura de Máquinas Paralelas

Durante muito tempo a classificação proposta por Flynn [Flynn 1972] no início dos anos 70 foi utilizada para caracterizar os diferentes tipos de arquitetura de computadores. Essa classificação é baseada no número de fluxo de dados (*Single Data stream* ou *Multiple Data stream*) e no número de fluxo de instruções (*Single Instruction* stream ou *Multiple Instruction stream*) presentes em uma determinada arquitetura. A combinação dessas possibilidades nos leva a quatro categorias de máquinas: *Single Instruction Single Data* (SISD), *Single Instruction Multiple Data* (SIMD), *Multiple Instruction Single Data* (MISD) e *Multiple Instruction Multiple Data* (MIMD). Duas dessas categorias são particularmente interessante no escopo deste artigo: SISD e MIMD.

As arquiteturas SISD representam o modelo de funcionamento seqüencial que corresponde a arquitetura de Von Neuman. Essa arquitetura é organizada com base em dois elementos básicos: a memória e processador. O processador lê as instruções da memória, uma a uma, e as executa sobre um fluxo de dados provenientes da memória. Esse modelo corresponde a um monoprocessador. Uma arquitetura MIMD é composta por vários processadores, onde cada processador tem a liberdade de executar um fluxo de instruções próprio sobre um fluxo de dados. Temos então vários fluxos de instruções

sobre diferentes fluxos de dados. As máquinas denominadas de multiprocessadores são exemplos típicos dessa categoria.

Os critérios adotados por Flynn são muito genéricos, e por conseqüência sua classificação é inadequada para os sistemas distribuídos porque todos são considerados máquinas MIMD independentes de sua real arquitetura. Atualmente é comum subdividir a classe MIMD em dois grupos: os multiprocessadores e os multicomputadores [Tanenbaum 1992] [Van Der Steen e Dongarra 1996].

Os **multiprocessadores**, também denominados de **sistemas fortemente acoplados**, ou ainda arquiteturas de memória compartilhada, são caracterizados por vários processadores dividirem um mesmo espaço de endereçamento. Em função do tipo de acesso que é feito a memória os multiprocessadores são ainda divididos em máquinas UMA (*Uniform Memory Access*) ou UMA (*Non-uniform Memory Access*). A arquitetura UMA é a mais comum, os processadores, a memória e os dispositivos de entrada e saída são conectadas a um mesmo barramento, sendo portanto compartilhados. A principal vantagem dessa arquitetura é a sua simplicidade de concepção, mas ela apresenta como desvantagem o fato de aumentando o número de processador aumentar também a contenção de acesso a memória. Para limitar o efeito da contenção se utiliza memórias *cache*, porém isso introduz problemas de coerência de dados entre as diferentes *caches* do sistema. Em uma arquitetura NUMA cada processador possui uma memória local que forma com a memória local dos demais processadores um espaço de endereçamento único. Os acesso a endereços que correspondem às memórias "distantes" (as que pertencem a outros processadores) são mais lentos que os acessos a memória local, por isto a referência "acesso de memória não uniforme". Nesse sistemas, é comum, por questões de performance, também se empregar *caches*.

No grupo dos **multicomputadores**, também denominados de **sistemas fracamente acoplados**, ou ainda arquiteturas de memória distribuída, a máquina parelela é definida por um conjunto de nós (computadores) independentes uns dos outros, isto é, cada nó possui seu próprio processador e sua própria memória. A comunicação entre os nós é necessariamente feita por troca de mensagens. Essa classe de máquinas é também conhecida como *clusters* (agregados).

O aspecto que particularmente nos interessa é o sistema operacional dessas máquinas. Nos multicomputadores, considerando que eles podem ser construídos reunindo-se diferentes máquinas "convencionais" interconectadas via rede, cada máquina sendo independente, ela pode possuir seu próprio sistema operacional. Nesse caso a máquina paralela pode ser homogênea, se o *cluster* é composto por máquinas de um mesmo tipo de sistema operacional, ou heterogênea, caso contrário. Entretanto, em ambos os casos, o sistema operacional é o da própria máquina, podendo ser um sistema operacional "convencional", para máquinas monoprocessadas, ou para o caso do nó ser um multiprocessador, um sistema operacional para máquinas multiprocessadas. Nesse caso é comum, através de bibliotecas de comunicação e de softwares especiais (*middleware*), criar a abstração de uma máquina virtual fornecendo ao usuário a "visão" de que essa rede se comporta como uma única máquina de *n* processadores. Esse tipo de sistema não será abordado neste artigo, maioreis informações podem ser encontradas em [Tanenbaum 1995] e [Stallings 2001].

No caso dos multiprocessadores o sistema operacional é único e necessita ser projetado de forma a considerar uma série de aspectos próprios desse tipo de arquitetura. Esse aspectos serão detalhados na próxima seção.

8.5.2. Arquiteturas de Sistemas Operacionais para Multiprocessadores

Um multiprocessador apresenta necessariamente um único sistema operacional. Entretanto, esse sistema operacional pode ser projetados de três formas diferentes: mestre-escravo, funcionalmente assimétrico e simétrico.
Em uma arquitetura mestre-escravo, o sistema operacional executa sempre um determinado processador; isto é, o *kernel* está em um único processador. Os demais processadores executam apenas processos de usuários e eventualmente alguma tarefa específica do sistema operacional. Devido a essa característica, o modelo mestre-escravo é dito assimétrico. O processador mestre é responsável por realizar todas as operações de entrada e saída, atender interrupções e efetuar o escalonamento das *threads* e/ou dos processos. Sempre que um processo, ou *thread*, em execução em um processador escravo necessitar de um serviço do sistema operacional, como por exemplo, uma operação de entrada e saída, ele deve enviar essa requisição para o processador mestre e esperar que o serviço requisitado seja executado. Essa abordagem de projeto de sistema operacional é relativamente simples e implica em pequenas modificações em um núcleo de um sistema operacional monoprocessado para ser adaptado a um multiprocessador. Qualquer conflito de acesso é facilmente resolvido porque apenas um processador tem o controle de toda a memória e dos recursos de entrada e saída. As principais desvantagens dessa abordagem são:

- Uma falha no processador mestre compromete totalmente o funcionamento do sistema, e;
- O processador mestre se torna um gargalo no sistema porque ele deve realizar todas as tarefas de gerenciamento (processo, memória e entrada/saída).

A abordagem funcionalmente assimétrico surge como uma solução para “desafogar” o processador mestre. Nesse caso, diferentes sub-sistemas do kernel são atribuídos a diferentes processadores criando um tipo de especialização desses por tarefas. Por exemplo, um processador pode ser encarregado da execução da pilha de protocolo de comunicação enquanto outro gerencia a memória. Esse tipo de abordagem se mostra mais adequada para a implementação de sistemas de propósitos específicos do que para um sistema operacional genérico [Vahalia 1996]. Como exemplo dessa abordagem pode-se citar o sistema Auxpex NS5000, que implementa um sistema de arquivos de alto desempenho.

Por último, a arquitetura simétrica, onde todos os processadores executam indistintamente o kernel, os processos (*threads*) de usuários estão em competição de igual para igual pelo uso de recursos do sistema (memória e entrada/saída). Nessa arquitetura o kernel deve ser construído como múltiplos processos (ou múltiplas *threads*) permitindo que porções do kernel sejam executadas em paralelo. A abordagem SMP (*Symmetric MultiProcessing*) adiciona um grau bastante grande de complexidade ao sistema operacional. Ele deve garantir que dois processadores não selecionem o mesmo processo e que nenhum processo seja “perdido” nas diferentes filas (apto, bloqueado, suspenso). O projeto de kernel simétrico é bastante delicado e involve uma

série de aspectos abrangendo desde a organização física, as estruturas de interconexão, os mecanismos de comunicação inter-processo, e o software de aplicação.

8.5.3. Multiprocessamento Simétrico

Um sistema operacional para multiprocessadores deve gerenciar processadores e outros recursos computacionas para que o usuário "enxergue" o sistema operacional da mesma forma que um sistema operacional multiprogramado "convencional". Um sistema operacional multiprocessador deve prover toda a funcionalidade de um sistema multiprogramado mais as características necessárias para acomodar o gerenciamento de múltiplos processadores. A primeira grande modificação arquitetural no projeto de um kernel simétrico é o suporte a *threads*. Dessa forma o kernel é organizado em tarefas independentes podendo ser executadas em paralelo nos diversos processadores. Em outros termos, o sistema operacional deve implementar um modelo de *threads* do tipo 1:1. O kernel deve ainda possibilitar que um usuário construa aplicações que, com auxílio do *multithreading*, possam explorar os múltiplos processadores. Entre os principais aspectos necessários a serem considerados no projeto de um sistema operacional SMP, temos [Stallings 2001]:

- **Execução simultânea**: as rotinas do kernel devem ser reentrantes para permitir que vários processadores executem o mesmo kernel simultaneamente. Com vários processadores executando a mesma, ou diferentes partes do kernel, as tabelas e estruturas de gerenciamento internas do kernel devem ser gerenciadas de forma a evitar deadlocks ou operações inválidas.
- **Sincronização**: com múltiplos processos ativos há a possibilidade de acessos simultâneos a espaço de memória e a recursos comuns do sistema, deve-se então garantir a exclusão mútua e ordenação de eventos, provendo o kernel de mecanismo básicos de sincronização.
- **Escalonamento**: qualquer processador deve ser capaz de realizar o escalonamento, por conseqüência conflitos devem ser evitados. Como o kernel é organizado na forma de múltiplas *threads* existe então a oportunidade de escalonar simultaneamente diferentes *threads* de um mesmo processo em vários processadores.
- **Gerenciamento de memória**: o gerenciamento de memória em um multiprocessador deve tratar com os mesmo aspectos existentes em um sistema operacional monoprocessado. Além disso, o sistema operacional deve considerar a capacidade do hardware de explorar paralelismo, como por exemplo, o uso ou não de memórias multiporta. Os mecanismos de paginação devem ser coordenados de forma a garantir a consistência quando vários processadores dividem uma página, ou segmento, e decidem realizar uma substituição dessa página ou segmento.
- **Confiabilidade e tolerância a falhas**: o sistema operacional deve prover uma degradação graciosa no sistema face a falha de um processador. O escalonador e outras partes do sistema operacional devem reconhecer a perda de um processador e proceder a re-organização de tabelas de gerenciamento de acordo com essa falha.

Para melhor avaliar o impacto que esses aspectos representam no projeto de um sistema operacional para multiprocessadores, vamos analisá-los tomando-se como base sistemas operacionais "convencionais" de máquinas monoprocessadas.

Em um kernel monolítico, o código do kernel não é reentrante, e deve ser criado um mecanismo que torne esse código um recurso não compatilhável, isto é, uma seção crítica. Nesse caso, o primeiro processo adquire o "direito de executar código do kernel" e os demais processadores ficam esperando. Considerando a duração de algumas chamadas de sistema, esta solução é muito ineficiente. Embora seja possível imaginar melhoria neste contexto, na prática um kernel monolítico não é usado como base para sistemas SMP. Um ponto bastante crítico dessa abordagem diz respeito com a confiabilidade e a tolerância a falhas. Imagine o que aconteceria se o processo que "adquiriu o direito de executar o kernel", ou o processador onde ele executa, falhasse. Seria bastante provável uma situação de *deadlock* no sistema.

Supondo-se como organização original um kernel convencional preemptivo, nós temos a nossa disposição uma série de mecanismos de sincronização clássicos para evitar que dois ou mais processos corrompam as estruturas de dados. Os mecanismos de sincronização são construídos a partir de algum tipo de operação atômica. Em máquinas monoprocessadas as soluções clássicas empregadas a nível do kernel são a desabilitação de interrupções e utilização de instruções do tipo *test-and-set* ou *swap*. O problema dessas soluções é que em máquinas multiprocessadoras elas deixam de ser efetivas. Desabilitar interrupções não garante a exclusão mútua no acesso às seções críticas, pois o controle de interrupção é especifico a cada processador, por conseqüência, os demais processadores que não executaram a instrução de desabilitação de interrupções podem executar a mesma área de código do processador que desabilitou as suas interrupções na intenção de proteger uma seção crítica.

As intruções *test-and-set*, ou *swap*, baseiam-se na leitura e na escrita de posições de memória empregadas como controle de acesso a seção crítica. Em multiprocessadores, temos, potencialmente a capacidade de dois ou mais processadores realizarem simultaneamente a leitura da mesma posição de memória. Esse problema é agravado se considerarmos que cada processador possui sua própria memória *cache* e, por conseqüência, podem possuir sua própria cópia de uma posição de memória. Nesse ponto entra um aspecto do projeto do hardware de multiprocessador: o arbitramento do barramento. O árbitro de barramento é responsável por controlar e evitar os conflitos de acesso a memória asssim como por manter a consistência entre as memórias *caches* dos vários processadores e a memória principal. Uma descrição mais detalhada desse tipo de arquitetura pode ser encontrada em [Vahalia 1996].

A palavra chave no projeto de um sistema operacional para multiprocessadores é sincronização. A sincronização é o meio utilizado para coordenar as atividades concorrentes susceptíveis a acessar dados compartilhados. O mecanismo de base é a exclusão mútua (mutex) que garante que, em dado momento, uma única *thread*, executa uma porção do código (seção crítica). Entretanto há um custo associado a utilização de mutex. Esse custo é proveniente das próprias primitivas relacionadas ao uso de mutex (*lock* e *unlock*) e da sincronização em si que reduz o paralelismo através de trocas de contexto. Uma primitiva do tipo lock, em mutex ocupado, implica em uma chamada ao escalonador do sistema e duas comutações de contexto, uma da *thread* que está sendo bloqueada e outra da *thread* que está sendo escalonada.

A forma de utilizar variáveis do tipo mutex está fortemente relacionada com a implementação de um algoritmo. Por exemplo, ao invés de alocar e liberar o mutex várias vezes durante a execução de um procedimento (princípio da granularidade pequena), é possível alocá-lo no início desse procedimento e liberá-lo apenas no final. Em outras palavras, empregamos aqui uma granularidade grande. O emprego de uma granularidade grande reduz o custo de chamadas às operações de sincronização porém aumenta a probabilidade de contenção, ou seja, aumenta o número de *threads* que podem se bloquear a espera desse mutex. Esse bloqueio subutiliza, de certa forma, o multiprocessador. É necessário então encontrar um ponto de equilíbrio entre a granularidade (contenção) e o número de operações de sincronização. Para atingir esse objetivo diferentes métodos tentam limitar os conflitos de acesso particionando o acesso aos dados ou eliminando as situações de conflitos. Classicamente, estes métodos são:

- Emprego de primitivas de mais alto nível como variáveis do tipo read-write. Aqui vários "leitores" são autorizados a ler um determinado dado, mas apenas um "escritor" pode modificá-la. Durante a operação de escrita, nenhuma leitura é permitida.
- Particionamento de dados em vários sub-grupos, cada um protegido por um mutex diferente. Um método comum é iniciar com um número pequeno de mutexes de granularidade grande e, segundo o desempenho obtido, reduzir a granularidade criando sub-grupos de dados e associando mutexes a eles.
- Emprego de algoritmos ditos "livres de bloqueio" (*lock-free*). Esses algoritmos são baseados na hipótese que os conflitos de acesso são raros e que não compensa realizar sempre as primitivas de sincronização. Nesse caso, o algoritmo deve ser capaz de detectar se houve um conflito de acesso e iniciar uma recuperação da consistência de dados. Normalmente esta operação representa um custo bastante elevado. Entretanto, se a taxa de conflito for baixa, o emprego esse tipo de algoritmo pode vir a ser interessante.

Esses métodos, na realidade, podem ser empregados tanto em máquina monoprocessadas como multiprocessadas. Apesar de funcionalmente corretas elas apresentam um custo de processamento que não é adequado, por questões de desempenho, para a implementação de um software como um sistema operacional. Para máquinas multiprocessadores a técnica do *spin-lock* é mais apropriada, permitindo a exploração de uma granularidade pequena.

Em uma máquina monoprocessadora, bloquear uma *thread* que espera por um recurso (mutex) é lógico pois o único processador da máquina deve ser liberado para executar outras *threads*. Uma dessas *threads* executará a liberação do recurso esperado e desbloqueará a *thread* a espera desse. Em uma máquina multiprocessadora, esta troca de contexto pode ser evitada permitindo a *thread* testar continuamente a disponibilidade do recurso (mutex). Esse é o princípio do *spin-lock*.

Para que o método do *spin-lock* funcione a contento, é necessário garantir que a *thread* responsável pela liberação do recurso (mutex) execute, o que é completamente viável em multiprocessadores (enquanto uma *thread* realiza *spin-lock* em um processador, a *thread* liberadora executa em outro). Para que esse mecanismo apresente um bom desempenho dois cuidados devem ser levados em conta. O primeiro é utilizar esta técnica apenas para casos em que o recurso (mutex) seja liberado rapidamente. O

objetivo é evitar que a *thread* perca muito tempo realizando o *spin-lock*. Uma técnica emprega é associar uma espécie de time-out, se a *thread* não obter o mutex realizando *spin-lock* dentro de um certo limite de tempo ela é bloqueada. O segundo cuidado é garantir que todas as *threads* sejam escalonadas, isso para garantir que a *thread* liberadora seja executada.

O outro ponto em que o projeto de um sistema operacional para máquinas multiprocessadores difere drasticamente do sistema operacional de uma máquina monoprocessadora é o escalonamento. Em um multiprocessador o escalonamento é mais complexo em função da existência de vários processos no estado de apto (pronto a executar) e de vários processadores. Uma característica importante é a existência de memória *cache* nos processadores.

Idealmente, o escalonador deveria executar a *thread* de mais alta prioridade em qualquer um dos processadores do sistema. Tal política necessita que o escalonador mantenha um conjunto global de todas as filas de execução divididas entre todos os processadores. Isto pode criar um gargalo no sistema desde que todos os processaodres concorrem para realizar um *lock* nessa fila. Entretanto quando uma *thread* executa ela transfere seus dados e instruções para a *cache* do processador que ela executa. Se a *thread* é preemptada e reescanolada em um curtíssimo intervalo de tempo pode ser vantajoso executá-la no mesmo processador, isto traria beneficios de desempenho em função de *thread* ainda encontrar na *cache* seus dados e instruções. Em outros termos, muitas vezes é mais interessante manter uma *thread* suspensa, mesmo existindo processador livre para executá-la, para que ela volte a executar no mesmo processador que executara antes. Essa política não deixa de ter um aspecto conflitante, para melhorar o desempenho não se aproveita um processador livre.

Para conciliar esses objetivos conflitantes, alguns sistemas empregam o que se denomina de política de afinidade. Nesses casos é comum organizar as *threads* em duas classes distintas de filas de escalonamento segundo o tipo de prioridades que é implementada pelo sistema operacional. Para um escalonamento tipo “convencional” como a implementada em vários tipos de UNIX, isto é, *time-sharing*, com prioridades váriaveis, as *threads* são mantidas em uma fila de aptos local a cada processado. Dessa forma elas são normalmente escalonadas no mesmo processador. Isso reduz a contenção na fila de aptos. Para previnir que ocorra um desbalanceamento de carga, o escalonador monitora as diferentes filas de *threads* aptas em cada processsador e as move de uma fila a outra de forma a eqüalizar a carga.

Nos sistemas que implementam uma política de prioridade fixa, as *threads* são escalonadas a partir de uma fila global única e o kernel tenta escaloná-las no processador em que elas executaram pela última vez. Nesse caso, é comum também a presença de uma chamada de sistema que força umaa *thread* a executar em um processador específico. Observa-se ainda que sistemas que adotam essa política de prioridades oferecem normalmente a política de *threads time-sharing* para processos usuários e prioridades fixas para processos que executam em modo considerado privilegiado. O objetivo nesses sistemas é oferecer alguma capacidade de tempo real.

Nas próximas seções nós analisaremos três estudos de casos (Linux, Windows 2000 e Linux para tempo real) salientado o emprego das técnicas apresentadas para cada um desses sistemas operacionais.

8.6 Estudo de Caso: Linux

Linux é um sistema operacional com código fonte aberto, estilo Unix, originalmente criado por Linus Torvalds a partir de 1991 com o auxílio de desenvolvedores espalhados ao redor do mundo. Linux é "software livre" no sentido que pode ser copiado, modificado, usado de qualquer forma e redistribuído sem nenhuma restrição, sendo distribuído sob o "GNU General Public License".

O sistema operacional Linux inclui multiprogramação, memória virtual, bibliotecas compartilhadas, protocolos de rede TCP/IP e muitas outras características consistentes com um sistema multiusuário tipo Unix. Uma descrição completa do Linux não cabe neste texto. Além da página oficial http://www.linux.org, qualquer pesquisa na Internet ou na livraria vai revelar uma enorme quantidade de material sobre o assunto.

O Linux é um sistema operacional em evolução, e existem diferenças importantes entre suas versões. Este texto descreve a versão 2.2 e baseia-se, principalmente, na descrição que aparece em [Bovet e Cesati 2001]. O Linux evolui rapidamente e alguns aspectos descritos aqui já sofreram alterações na versão 2.4. Em alguns momentos serão feitas referências à arquitetura do Pentium Intel, por ser o processador mais usado na execução do Linux. Esta seção descreve principalmente sistemas com um único processador, mas são feitas algumas considerações sobre o ambiente SMP.

O Linux segue a linha do kernel convencional não-preemptivo, mas inclui suporte para multiprocessamento simétrico. Dentro do kernel existem dois tipos principais de fluxo de controle: aqueles originados por um processo realizando uma chamada de sistema e aqueles disparados pela ocorrência de interrupções de hardware ou de proteção, estas últimas chamadas de exceções. Estes fluxos de controle internos ao kernel não possuem um descritor próprio e eles não são escalonados por algum algoritmo central. Muitas vezes executam do início ao fim, pois o kernel é não preemptivo, porém a execução de vários fluxos dentro do kernel pode ser intercalada quando acontecem interrupções ou quando o processo solicita seu próprio bloqueio, liberando o processador voluntariamente. Este entrelaçamento da execução dentro do kernel exige cuidados com as estruturas de dados compartilhadas.

Um processo executando em modo kernel não pode ser substituído por outro processo de mais alta prioridade, a não ser quando o processo executando no kernel solicita explicitamente o seu próprio bloqueio, seguido de um chaveamento. O processo dentro do kernel pode ser interrompido pela ocorrência de interrupções de hardware e de proteção, mas ao término do respectivo tratador ele retoma a sua execução. O fluxo de controle associado com uma interrupção somente pode ser interrompido pela ocorrência de outra interrupção de hardware ou proteção. Uma conseqüência dessa organização é que chamadas de sistema não bloqueantes são atômicas com respeito às chamadas de sistema executadas por outros processos. Logo, uma estrutura de dados do kernel não acessada por tratadores de interrupção de hardware ou de proteção (exceções) não precisa ser protegida. Apenas é necessário que, antes de liberar o processador, o processo executando código do kernel coloque as estruturas de dados em um estado consistente. Esta é a maior vantagem da não preempção. A maior desvantagem está em fazer um processo de alta prioridade esperar que um processo de baixa prioridade termine a sua chamada de sistema antes de obter o processador.

Em alguns momentos são utilizadas "operações atômicas" para garantir a consistência de uma operação, mesmo com interrupções habilitadas. Uma operação atômica neste contexto corresponde a uma única instrução de máquina, a qual não pode ser interrompida durante sua execução. No caso do Pentium Intel podemos usar uma instrução do tipo lê/modifica/escreve no caso de monoprocessador, ou uma instrução do tipo lê/modifica/escreve precedida da instrução de máquina *lock*, no caso de multiprocessadores.

Muitas vezes as interrupções são simplesmente desabilitadas durante o acesso a uma estrutura de dados, para garantir que não haverá interferência de nenhum tipo. Este é o mecanismo mais freqüente no Linux (versão monoprocessada). É importante observar que desabilitar interrupções não impede as interrupções de proteção (exceções). Também podem existir seções críticas aninhadas, quando as interrupções já desabilitadas são novamente desabilitadas. Neste caso, não basta habilitar as interrupções na saída da seção crítica, é necessário recolocar o valor original da flag EF encontrado no início de cada seção crítica. A seqüência fica sendo: salva as flags em uma variável, desabilita interrupções, executa a seção crítica, restaura as flags a partir da variável usada.

No Linux, um tratador de interrupção não precisa realizar todo o trabalho associado com a interrupção que ocorreu. Ele pode "anotar" parte do trabalho para ser realizado mais tarde. No Linux 2.2 este trabalho postergado é chamado de *bottom-half* e será executado em um momento conveniente para o kernel, tal como no término de uma chamada de sistema, no término do tratamento de uma exceção, no término do tratamento de uma interrupção ou no momento do chaveamento entre dois processos. A capacidade de postergar parte de seu trabalho é importante, pois como tratadores de interrupção não podem ficar bloqueados como se fossem processos, eles não podem depender da disponibilidade de estruturas de dados compartilhadas para terminar seu trabalho. Existem vários tipos de *bottom-half*, e o momento de execução de cada um depende dos recursos que eles necessitam. O mecanismo do *bottom-half* é um passo na direção de *threads* de kernel, as quais representam uma solução mais elegante.

Em alguns momentos o código do kernel utiliza semáforos para proteger seções críticas. Os semáforos são definidos da forma tradicional, como uma estrutura de dados composta por um contador inteiro, uma lista de processos bloqueados e mais campos auxiliares. Na implementação das operações P e V, as interrupções são desabilitadas para garantir sua atomicidade em monoprocessadores. Em alguns momentos dentro do kernel o fluxo de controle não pode ficar bloqueado, e neste caso é utilizada uma variação da operação P que, em caso do recurso ocupado, retorna um código de erro mas não causa o bloqueio. Também existe uma variação da operação P usada em *device-drivers*, na qual o processo pode ficar bloqueado mas continua sensível aos sinais Unix, os quais podem fazer este processo acordar e desistir de obter o semáforo.

Como o kernel não é preemptivo, poucos semáforos são utilizados. No caso de monoprocessamento, semáforos são usados apenas quando o processo poderá ficar bloqueado mais tarde em função de um acesso a disco, ou quando um tratador de interrupção pode acessar uma estrutura de dados global do kernel. Eles são encontrados principalmente na gerência de memória e no sistema de arquivos. Deadlock é evitado através da ordenação dos semáforos e alocação deles em uma certa ordem [Silberschatz e Galvin 1999].

Em máquinas SMP diversos processadores podem executar em modo supervisor simultaneamente, logo os benefícios de um kernel não preemptivo são reduzidos. É necessária uma sincronização explícita nas seções críticas do kernel através de semáforos e/ou *spin-locks*. Linux versão 2.0 usava uma solução bem simples: A qualquer instante, no máximo um processador podia acessar as estruturas de dados do kernel e tratar interrupções. Esta solução limitava o paralelismo no sistema.

Na versão 2.2 muitas seções críticas foram tratadas individualmente e a restrição geral foi removida. *Spin-locks* são usados. A solução implica em *busy-waiting*, pois o processador continua a executar o laço do *spin-lock* enquanto espera a seção crítica ser liberada. Se a seção crítica for rápida, ainda é melhor isto do que realizar um chaveamento de processo. Observe que o *spin-lock* impede o acesso à estrutura de dados por outro processador, mas não impede que tratadores de interrupção disparados no mesmo processador que obteve o *spin-lock* acessem estas estruturas. Logo, são usadas macros que, ao mesmo tempo, adquirem o *spin-lock* e desabilitam as interrupções. Além do *spin-lock* simples (libera ou proibe qualquer tipo de acesso), o kernel do Linux utiliza uma variação chamada "*read/write spin-lock*", o qual permite o acesso simultâneo de vários processos leitores, mas garante exclusão mútua na escrita.

A versão 2.2 ainda inclui um *spin-lock* global do kernel, usado para proteger coletivamente todas as estruturas de dados usadas no acesso a arquivos, a maioria das relacionadas com comunicação via rede, todas as relacionadas com IPC (*Inter-Process Communication*) entre processos de usuário e ainda outras mais. Qualquer rotina que acesse uma dessas estruturas de dados precisa obter este *spin-lock* antes, o que significa que um grande número de chamadas de sistema no Linux não podem executar simultaneamente em diferentes processadores. Fosse o Linux um kernel convencional preemptivo, o paralelismo potencial do SMP seria muito melhor aproveitado. Mesmo assim, várias estruturas de dados já estão protegidas por *spin-locks* individuais. A granularidade menor no mecanismo de sincronização proporciona maior paralelismo e desempenho. Entretanto, sua inclusão em um kernel originalmente monoprocessado e não preemptivo deve ser muito cuidadosa para evitar *deadlock* e outras anomalias.

É importante destacar que o kernel do Linux está em constante evolução. Por exemplo, já na versão 2.4 o mecanismo de *bottom-half* foi melhorado de tal forma que possa existir execução simultânea destes blocos de código em diferentes processadores, aumentando o desempenho do sistema. Em geral, é possível dizer que a evolução do kernel do Linux o leva, a cada passo, mais perto da idéia de um programa concorrente onde processos executam concorrentemente no kernel e podem ser preemptados a qualquer momento. A longo prazo o kernel do Linux deverá tornar-se mais um programa concorrente típico e menos um aglomerado de tratamentos diferenciados para dezenas de situações específicas. A descrição do kernel do Linux feita aqui foi simplificada. Em [Bovet e Cesati 2001] são dedicadas mais de 600 páginas ao assunto.

8.7. Estudo de Caso: Windows 2000

O Windows NT é um sistema operacional proprietário, desenvolvido pela Microsoft, que surgiu com o objetivo ser um sistema operacional da década de 90, atento aos novos avanços tecnológicos e às exigências do mercado. Desde o seu lançamento, em 1993, com a versão 3.1, o Windows NT teve a preocupação de fornecer suporte a ambientes de rede e distribuídos. Sua evolução, até chegar ao Windows NT 5.0, comercialmente

conhecido como Windows 2000, foi orientada à interoperabilidade com outros sistemas operacionais. A estrutura básica da arquitetura do Windows NT manteve-se praticamente inalterada desde o seu lançamento até as versões mais recentes.

Como vimos um sistema operacional é um *software* extremamente complexo, por isso, vários modelos de arquiteturas foram propostos para melhor organizar os detalhes de sua implementação. Esses modelos, conforme já apresentado, vão desde sistemas baseados em *kernel* monolítico até sistemas totalmente moduláveis, baseados em micronúcleo (*microkernel*).

A arquitetura do Windows 2000 é fortemente inspirada no princípio de micronúcleo. Assim, cada funcionalidade do sistema é oferecida e gerenciada por um único componente do sistema operacional. Os demais componentes do sistema operacional e todas as aplicações acessam os serviços providos por um determinado componente através de uma interface bem definida. Teoricamente, cada módulo (componente) pode ser removido, atualizado, ou substituído sem necessitar de alterações nas demais partes do sistema. O Windows 2000 não é puramente orientado à filosofia micronúcleo porque módulos fora do micronúcleo executam operações em modo protegido (modo kernel). A justificativa para essa decisão de projeto está no desempenho. Em uma abordagem orientada a micronúcleo "pura", uma aplicação que necessite executar uma operação privilegiada deve solicitar esse serviço ao micronúcleo. Esse procedimento envolve uma série de trocas de contexto. No Windows 2000, para evitar essa troca de contexto, certos subsistemas (módulos ou componentes) passam de modo usuário para modo protegido e implementam diretamente a função desejada, evitando assim a passagem pelo núcleo e as trocas de contexto que isso implica.

O Windows 2000 segue também uma organização em camadas. Nessa abordagem, o sistema operacional é dividido em módulos que são dispostos uns sobre os outros, em camadas. Cada camada oferece um conjunto de serviços à camada superior e só pode utilizar serviços fornecidos pela camada imediatamente inferior. Outro conceito explorado pelo Windows 2000 é o modelo orientado a objetos. Nesse modelo, recursos do sistema, arquivos, memória e dispositivos físicos, são implementados por objetos e manipulados através de métodos (serviços) associados a esses objetos.

O Windows 2000 foi projetado de forma a permitir a execução de aplicações escritas para outros sistemas operacionais. Essa facilidade é suportada a partir de subsistemas que, implementados como um processo separado, fornecem um ambiente de execução compatível a um determinado sistema operacional. Esse ambiente é composto, além de uma interface gráfica e de um interpretador de comandos, por uma interface de programação (API) compatível com os serviços (chamadas de sistema) do sistema operacional que o subsistema implementa. Isso implica que uma aplicação escrita para um sistema operacional particular pode executar sem alterações no Windows 2000 por "enxergar" as mesmas funções existentes no sistema nativo para o qual foi escrito. O mais importante dos subsistemas do Windows 2000 é o Win32, que possibilita que aplicações escritas para outros sistemas operacionais Microsoft executem no Windows 2000 sem problemas. Outros subsistemas disponíveis são o subsistema OS/2 e o subsistema POSIX.

A estrutura do Windows 2000 pode ser dividida em duas partes: modo usuário (onde estão localizados os subsistemas protegidos) e modo kernel (o executivo). Os subsistemas protegidos são assim denominados porque residem em processos separados cuja memória é protegida do acesso de outros processos. Os subsistemas interagem entre si através de um mecanismo de troca de mensagens (*Local Procedure Call - LPC*). No modo kernel, rodam os componentes do sistema operacional que necessitam de desempenho e por isso interagem com o hardware e um com o outro sem estarem sujeitos a trocas de contexto e de modo. Todos os componentes estão protegidos das aplicações porque estas não possuem acesso à parte protegida do sistema operacional. Ainda, cada componente está protegido um do outro devido à adoção da orientação a objetos. Todo acesso a um objeto é feito através de um método.

O modo kernel é estruturado em três grandes módulos funcionais: *hardware abstraction layer*, drivers de dispositivos e o executivo. A camada denominada de *hardware abstraction layer* (HAL) é um módulo carregável do núcleo. Esse módulo respeita uma interface padrão de serviços, porém possui uma implementação específica para o hardware no qual o Windows 2000 está executando. Todas as funcionalidades que são dependentes de um determinado hardware, como interfaces de E/S, controladores de dispositivos e de interrupções, ou ainda, o próprio processador, são implementadas dentro desse módulo. Esse tipo de projeto permite que todos os componentes do sistema operacional acima do módulo HAL executem de forma independente do hardware, fornecendo assim o grau de portabilidade necessário a um sistema que visa a operar em ambientes heterogêneos.

Os drivers de dispositivos são outra categoria de módulos carregáveis do núcleo. Esses drivers oferecem, dentro do executivo do Windows 2000, uma interface entre o sistema de E/S e o HAL. O gerenciamento dos drivers de dispositivos foi um dos aspectos em que o Windows 2000 (NT 5.0) apresentou uma evolução significativa em relação a sua versão anterior (Windows NT 4.0) através da integração de suporte a *plug-and-play*.

O executivo constitui o núcleo do sistema operacional Windows 2000. É ele que implenta os serviços básicos do Windows 2000, exportando funções para serem utilizadas em modo usuário e funções que só são acessíveis por componentes (módulos) pertencentes ao próprio núcleo. Os principais componentes do executivo são: gerência de objetos, gerência de processos e *threads*, gerência de memória virtual, monitor de segurança, módulo de suporte a *Local Procedure Call* (LPC) e gerência de entrada e saída.

Uma questão se apresenta no momento: existem, uma ou duas versões de Windows 2000? Há uma versão para monoprocessadores e outra para multiprocessadores? Afora a camada HAL, que por sua própria funcionalidade difere de uma arquitetura a outra, apenas um arquivo é diferente entre a versão para monoprocessadores e multiprocessadores. Esse arquivo corresponde a implementação do executivo. Todos os demais arquivos que compõem o Windows 2000, isto é, utilitários, bibliotecas e os *drivers* de dispositivos, foram concebidos de forma a serem executados corretamente, sem alterações, tanto em sistemas monoprocessados como multiprocessados. A diferença básica entre as versões monoprocessador e multiprocessador do executivo está em operações de sincronizações adicionais necessárias a versão multiprocessador. Essa decisão de projeto faz com que o sistema

monoprocessador “não pague” pelo custo adicional dessa sincronização. No momento da instalação do Windows é selecionado, em função do hardware, qual versão deve ser utilizada. Outra diferença importante entre as versões do monoprocessador e multiprocessador do Windows 2000 é relacionada ao escalonamento.

Cada sistema operacional tem a sua própria forma de implementar processos; as variações estão nas estruturas de dados utilizadas para representar fluxos de execução, sua denominação, como são protegidos uns em relação aos outros e na forma de inter-relacionamento. O Windows 2000 implementa o conceito de processo a partir de dois objetos: objeto processo e objeto *thread*. O objeto processo é a entidade que corresponde a recursos do sistema tais como memória, arquivos, etc. O objeto *thread*, por sua vez, constitui uma unidade de trabalho que é executada de forma seqüencial e podendo ser interrompida em qualquer ponto.

A criação de um processo em Windows 2000 corresponde a instanciar (criar) um objeto do tipo processo, o qual é uma espécie de “molde” para novos processos. Nesse momento, uma série de atributos são inicializados para esse novo processo, como, por exemplo, um identificador de processo (*pid*), descritores de proteção, prioridades, quotas, etc. A unidade de escalonamento do Windows 2000 é o conceito de *thread*. A cada processo está associada, no mínimo, uma *thread*. Cada *thread* pode criar outras *threads*. Essa organização permite a execução concorrente dos processos, além de possibilitar uma concorrência entre as *threads* que pertencem a um mesmo processo. Em outros termos, a unidade de escalonamento do windows NT é a *thread.*

As *threads* de qualquer processo, inclusive as do executivo do Windows 2000, podem, em máquinas multiprocessadoras, ser executadas em qualquer processador. Dessa forma, o escalonador do Windows 2000 atribui uma *thread* pronta a executar (apta) para o próximo processador disponível. Múltiplas *threads* de um mesmo processo podem estar em execução simultaneamente.

O escalonador do Windows 2000 é preemptivo com prioridades. As prioridades são organizadas em duas classes: tempo real e variável. Cada classe possui 16 níveis de prioridades, sendo que as *threads* da classe tempo real têm precedência sobre as *threads* da classe variável, isto é, sempre que não houver processador disponível, uma *thread* de classe variável é preemptada em favor de uma *thread* da classe tempo real. Todas as *threads* prontas para executar são mantidas em estruturas de filas associadas a prioridades em cada uma das classes. Cada fila é atendida por uma política *Round-robin*. A atribuição de prioridades a *threads* é diferente para cada uma das classes. Enquanto na classe de tempo real, as *threads* possuem prioridade fixa, determinada no momento de sua criação, as *threads* da classe variável têm suas prioridades atribuídas de forma dinâmica (por isso o nome de “variável” para essa classe). Dessa forma, uma *thread* de tempo real, quando criada, recebe uma prioridade e será sempre inserida na fila dessa prioridade, ao passo que uma *thread* da classe variável poderá migrar entre as diferentes filas de prioridades. Em outros termos, o Windows 2000 implementa um esquema de múltiplas filas para as *threads* da classe real e múltiplas filas com realimentação para as *threads* da classe variável.

Em máquinas monoprocessadoras, a *thread* de mais alta prioridade está sempre ativa a menos que esteja bloqueada esperando por um evento (E/S ou sincronização). Caso exista mais de uma *thread* com um mesmo nível de prioridade o processador é compartilhado de forma *round-robin* entre essas *threads*. Em um sistema

multiprocessador com *n* processadores, as *threads* de mais alta prioridade executam nos *n-1* processadores extras. As *threads* de mais baixa prioridade disputam o processador restante.

Na realidade, existe ainda um fator adicional que influencia fortemente o escalonamento do Windows 2000: a afinidade de uma *thread*. Uma *thread* pode definir sobre qual (ou quais) processador(es) ela deseja executar. Nesse caso, se a *thread* estiver apta a executar, porém o processador não estiver disponível, a *thread* é forçada a esperar, e uma outra *thread* é escalonada em seu lugar. Como visto anteriormente o conceito de afinidade é motivado pela tentativa de reaproveitar os dados armazenados na *cache* do processador pela execução anterior de uma *thread*. O Windows 2000 possibilita dois tipos de afinidade: *soft* e *hard*. Por *default*, a política de afinidade *soft* é utilizada para escalonar *threads* a processadores. Nesse caso, o *dispatcher* tenta alocar uma *thread* ao mesmo processador que ela executou anteriormente, porém, se isso, não for possível, a *thread* poderá ser alocada a outro processador. Já com a política de afinidade *hard*, uma *thread* (com os devidos privilégios) executa em apenas um determinado processador.

Nesta seção nós descrevemos suscintamente o kernel do Windows 2000, centrando a discussão no seu modelo arquitetural e no suporte para multiprocessamento simétrico. Assim como no caso do Linux é impossível descrever detalhadamente o kernel do Windows no contexto deste artigo. Uma boa referência para os leitores interessados em conhecer mais sobre esse sistema operacional é [Solomon 1998].

8.8. Estudo de Caso: Linux para Tempo Real

O Linux convencional segue o estilo de um kernel Unix tradicional. Por exemplo, em [Euler e Silva 1999] é descrita uma aplicação de controle de aproximação de aeronaves em aeroportos que é executada simultaneamente com outros programas que representam uma carga tipicamente encontrada em sistemas operacionais de tempo real. A aplicação em questão utiliza um servidor gráfico X e deve apresentar as informações dentro de certos limites de tempo, o que a caracteriza como uma aplicação de tempo real brando (*soft real-time*). O sistema operacional Linux kernel 2.0 foi utilizado. Mesmo quando a aplicação tempo real executa na classe *"real-time"* e as demais aplicações executam na classe *"time-sharing"* o desempenho não foi completamente satisfatório. Em especial, processos *daemon* que executam serviços tanto para aplicações de tempo real como para aplicações convencionais executam com sua própria prioridade e atendem as requisições pela ordem de chegada. Neste momento, os processos de tempo real perdem a vantagem que tem com respeito a política de escalonamento e passam a ter o mesmo atendimento que qualquer outro processo.

Entretanto, o Linux possui um recurso que facilita sua adaptação para o contexto de tempo real. Embora o kernel ocupe um único espaço de endereçamento, ele aceita módulos carregáveis em tempo de execução, os quais podem ser incluídos e excluídos do kernel sob demanda. Esses módulos executam em modo privilegiado e são usados normalmente na implementação de tratadores de dispositivos (*device-drivers*), sistemas de arquivos e protocolos de comunicação. No caso dos sistemas de tempo real, esta característica facilita a transferência de tecnologia da pesquisa para a prática. Soluções de escalonamento tempo real podem ser implantadas dentro do kernel de um sistema operacional de verdade. Como o código fonte do Linux é aberto, é possível estudar o

seu comportamento temporal, algo que é impossível com SOTR comerciais cujo kernel é tipicamente uma caixa preta.

Na página http://www.linux.org/projects/software.html estão listados vários projetos de software ligados ao Linux, os quais incluem novos componentes, aplicações e *device-drivers*. Entre eles existem três projetos envolvendo adaptações do Linux para tempo real, os quais serão descritos a seguir. Vários outros projetos e experiências também estão sendo conduzidos em todo o mundo. Por exemplo, o LINUX-SMART, desenvolvido no Brasil pelo IME-USP [Vieira 1999], e o RTAI descrito em http://www.aero.polimi.it/~rtai/. Farto material sobre variações do Linux para tempo real pode ser encontrado nas páginas da Real Time Linux Foundation, Inc. em http://www.realtimelinuxfoundation.org, inclusive sua página de projetos em andamento em http://www.realtimelinuxfoundation.org/variants/variants.html.

8.8.1 Real-Time Linux

O RT-Linux (http://www.fsmlabs.com/) é uma extensão do Linux que se propõe a suportar tarefas com restrições temporais críticas. O seu desenvolvimento teve inicio no Department of Computer Science do New Mexico Institute of Technology. Atualmente o sistema é mantido pela empresa FSMLabs.

O RT-Linux é um sistema operacional no qual um microkernel de tempo real co-existe com o kernel tradicional do Linux. O objetivo desse arranjo é permitir que aplicações utilizem os serviços sofisticados e o bom comportamento no caso médio do Linux tradicional, ao mesmo tempo que permite tarefas de tempo real operarem sobre um ambiente mais previsível e com baixa latência. O microkernel de tempo real executa o kernel tradicional como sua tarefa de mais baixa prioridade (Tarefa Linux), usando o conceito de máquina virtual para tornar o kernel tradicional e todas as suas aplicações completamente interrompíveis todo o tempo.

Todas as interrupções são inicialmente tratadas pelo microkernel de tempo real, e são passadas para a Tarefa Linux somente quando não existem tarefas de tempo real para executar. Para minimizar mudanças no kernel tradicional, o hardware que controla interrupções é emulado. Assim, quando o kernel tradicional "desabilita interrupções", o software que emula o controlador de interrupções passa a enfileirar as interrupções que acontecerem e não forem completamente tratadas pelo microkernel de tempo real.

Tarefas de tempo real não podem usar as chamadas de sistema convencionais nem acessar as estruturas de dados do kernel Linux. Tarefas de tempo real e processos convencionais podem comunicar-se através de filas e memória compartilhada. As filas, chamadas de RT-FIFO, são na verdade *buffers* utilizados para a troca de mensagens, projetadas de tal forma que tarefas de tempo real nunca são bloqueadas.

Uma aplicação tempo real típica consiste de tarefas de tempo real incorporadas ao sistema na forma de módulos de kernel carregáveis e também processos Linux convencionais, os quais são responsáveis por funções tais como o registro de dados em arquivos, atualização da tela, comunicação via rede e outras funções sem restrições temporais. Um exemplo típico são aplicações de aquisição de dados. Observe que tanto as tarefas de tempo real como o próprio microkernel são carregadas como módulos adicionais ao kernel tradicional.

Os serviços oferecidos pelo microkernel de tempo real são mínimos: tarefas com escalonamento baseado em prioridades fixas e alocação estática de memória. A comunicação entre tarefas de tempo real utiliza memória compartilhada e a sincronização pode ser feita via desabilitação das interrupções de hardware. Existem módulos de kernel opcionais que implementam outros serviços, tais como outros algoritmos de escalonamento e semáforos. O mecanismo de módulos de kernel do Linux permite que novos serviços sejam disponibilizados para as tarefas de tempo real. Entretanto, quanto mais complexos forem estes serviços menor será a previsibilidade das tarefas de tempo real.

A descrição feita aqui refere-se a primeira versão do RT-Linux, a qual provia apenas o essencial para tarefas de tempo real. A segunda versão manteve o projeto básico mas aumentou o conjunto de serviços disponíveis para tarefas de tempo real, ao mesmo tempo que procurou fornecer uma interface Posix também a nível do microkernel. A inclusão do Posix deve-se principalmente à demanda de empresas que gostariam de portar aplicações existentes para este sistema, e a disponibilidade de uma interface Posix para o microkernel torna este trabalho mais fácil.

8.8.2 RED-Linux

O objetivo do projeto RED-Linux (http://linux.ece.uci.edu/RED-Linux/) é fornecer suporte de escalonamento tempo real para o Linux, através da integração de escalonadores baseados em prioridade, baseados no tempo e baseados em compartilhamento de recursos. O objetivo é suportar algoritmos de escalonamento dependentes da aplicação, os quais podem ser colecionados em uma biblioteca de escalonadores e reusados em outras aplicações. O RED-Linux inclui também uma alteração no kernel para diminuir a latência das interrupções. Além disto, ele incorpora soluções do RT-Linux para temporizadores de alta resolução e o mecanismo para emulação de interrupções.

O escalonador implementado no RED-Linux é dividido em dois componentes: o Alocador e o Disparador. O Disparador implementa o mecanismo de escalonamento básico, enquanto o Alocador implementa a política que gerencia o tempo do processador e os recursos do sistema com o propósito de atender aos requisitos temporais dos processos da aplicação. A política de escalonamento (Alocador) pode ser modificada sem alterar os mecanismos de escalonamento de baixo nível (Disparador).

O Disparador é implementado como um módulo do kernel. Ele é responsável por escalonar processos de tempo real que foram registrados com o Alocador. Processos convencionais são escalonados pelo escalonador original do Linux quando nenhum processo de tempo real estiver pronto para executar ou durante o tempo reservado para processos convencionais pela política em uso. O Disparador utiliza um mecanismo de escalonamento bastante flexível, o qual pode ser usado para emular o comportamento dos algoritmos de escalonamento tempo real mais conhecidos, bastando para isto configurar de maneira apropriada os parâmetros que governam o escalonamento de cada processo. O artigo [Wang e Lin1999] descreve este mecanismo.

O Alocador é utilizado para definir os parâmetros de escalonamento de cada novo processo tempo real. Na maioria das aplicações ele pode executar fora do kernel, como parte da aplicação ou um servidor auxiliar. Executando fora do kernel ele pode ser mais facilmente substituido pelo desenvolvedor da aplicação, se isto for necessário. O

RED-Linux inclui uma API para que o Alocador possa interagir com o Disparador. Processos de tempo real inicialmente registram-se com o Alocador que, por sua vez, informa os seus parâmetros para o Disparador. O Alocador executa como o processo tempo real de mais alta prioridade e faz o mapeamento da política de escalonamento como selecionada pela aplicação para o mecanismo disponível no kernel do RED-Linux.

8.8.3 KU Real-Time Linux

O KURT-Linux (http://www.ittc.ukans.edu/kurt/), ou Kansas University Real Time Linux Project é um sistema operacional de tempo real que permite o escalonamento explícito e relativamente preciso no tempo de qualquer evento, inclusive a execução de tarefas com restrições de tempo real.

KURT-Linux é uma modificação do kernel tradicional, onde destaca-se o aumento na precisão da marcação da passagem do tempo real e, como conseqüência, na maior precisão de qualquer ação associada com a passagem do tempo. Uma descrição do mecanismo usado pode ser encontrada em [Srinivasan et al. 1998]. Uma importante constatação foi que o escalonamento explícito de cada evento do sistema usando uma resolução de microsegundos gerou uma carga adicional muito pequena ao sistema. Várias técnicas foram usadas em conjunto para obter maior precisão de relógio.

Módulos de tempo real pertencentes a aplicação são incorporados ao kernel. O escalonamento é feito através de uma lista que informa qual rotina presente em um módulo de tempo real deve executar quando. Toda rotina de tempo real executa a partir da hora marcada e suspende a si própria usando uma chamada de sistema apropriada. Desta forma, a precisão do relógio é transferida para o comportamento das tarefas. O sistema operacional supõe que os módulos de tempo real são bem comportados e não vão exceder o tempo de processador previamente alocado. Esta semântica para tempo real está descrita em [Hill et al. 1998]. Embora este tipo de escalonamento seja menos flexível do que o baseado em prioridades, ele é suficiente para um grande conjunto de aplicações. Os autores do KURT-Linux atestam que nunca foi o objetivo do projeto suportar todos os algoritmos de escalonamento tempo real presentes na literatura.

Uma extensão ao modelo original permite que uma tarefa programada como um laço infinito (como um servidor, por exemplo) execute em determinados momentos previamente reservados. Por exemplo, execute 100 microsegundos dentro de cada milisegundo. Soluções de escalonamento baseadas na utilização do processador podem ser implementadas através deste mecanismo.

8.9. Conclusão

Ao longo dos últimos 40 anos, muito esforço foi empreendido no sentido de criar as estruturas de software mais apropriadas para organizar internamente os sistemas operacionais. Existem diferenças entre os serviços oferecidos por um pequeno sistema operacional de propósito específico e um sistema operacional de propósito geral. Também existem diferenças entre o ambiente computacional da década de 60 e este do início do século 21. Apesar disso, existe um pequeno número de diferentes organizações para sistemas operacionais que representam a maioria dos sistemas existentes. Também é razoável afirmar que, desde o início dos anos 70, o sistema operacional Unix e todas as suas variações exerceram forte influência sobre pesquisadores e projetistas da área.

Após uma rápida revisão de programação concorrente, este capítulo discutiu as possíveis organizações para um kernel (núcleo) de sistema operacional. Foi apresentada uma taxonomia para orientar o leitor no entendimento das organizações existentes e das conseqüências que cada uma delas acarreta para aplicações e usuários. Em muitas das diversas organizações possíveis, o kernel aparece como um programa concorrente, onde a preocupação com a sincronização entre processos está sempre presente. Além dos ambientes tradicionais de propósito geral, o texto tratou também duas classes especiais de sistemas operacionais: tempo real e máquinas paralelas. Primeiro foi descrita a problemática associada com a construção de suportes para aplicações com restrições de tempo real e indicadas as propriedades desejadas em tais sistemas. Em seguida, as características próprias dos sistemas operacionais para máquinas paralelas foram exploradas. Com o propósito de tornar mais palpáveis os conceitos apresentados, três estudos de caso foram apresentados: Linux, Windows2000 e Linux para tempo real.

O sistema operacional sempre teve o papel de um administrador, tentando compatibilizar, da melhor forma possível, as demandas dos programas de usuário com os recursos existentes no hardware. Atualmente estão acontecendo importantes mudanças nessas duas áreas. Os recursos disponíveis no hardware continuam a aumentar a taxas geométricas, sendo a capacidade dos discos, a velocidade das redes, o tamanho da memória principal e a interconectividade as grandes vedetes. Por outro lado, aplicações distribuídas e/ou envolvendo mídias contínuas são cada vez mais requisitadas e problemas com segurança aumentam a cada dia. Também estão sendo dissiminadas plataformas do tipo *wireless* e *embedded*, com demandas específicas e recursos limitados. Existe o sentimento de que os sistemas operacionais deverão em breve necessitar um salto de flexibilidade em sua organização, para atender as demandas da computação no século 21. Com certeza, as organizações apresentadas neste texto continuarão a evoluir nos próximos anos.

Referências

Boehm, B. (1981) “Software Engineering Economics”, Prentice-Hall.

Bovet, D. P. e Cesati, M. (2001) “Understanding the Linux Kernel”, O’Reilly & Associates.

Dijkstra, E. W. (1968) “The Structure of the THE Multiprogramming System”, Communications of the ACM, vol. 11, n. 5, pp. 341-346.

Euler, J. M. do C. Noronha e Silva, D. M. da (1999) “Estudo de Caso: Desempenho do Sistema Operacional Linux para Aplicações Multimídia em Tempo Real”, Anais do II Workshop de Tempo Real, Salvador-BA.

Farines, J.-M.; Fraga, J da S. e Oliveira, R. S. de (2000) “Sistemas de Tempo Real”, Escola de Computação, IME-USP, São Paulo-SP.

Flynn, M. J. (1972) “Some computer organization and their effectivenness”, IEEE Transactions on Computers, C-21(9):948-960.

Gallmeister, B. O. (1995) “POSIX.4 Programming for the Real World”, O'Reilly & Associates.

Hill, R.; Srinivasan, B.; Pather, S. e Niehaus, D. (1998) “Temporal Resolution and Real-Time Extensions to Linux”, Technical Report ITTC-FY98-TR-11510-03,

Information and Telecommunication Technology Center, Electrical Engineering and Computer Science Department, University of Kansas.

Holt, R. C. (1983) “Concurrent Euclid, The Unix System, and Tunis”, Addison-Wesley.

Oliveira, R. S. de; Carissimi, A. da S. e Toscani, S. S. (2001) “Sistemas Operacionais”, 2ª. edição, Sagra-Luzzatto.

Oliveira, Rômulo S. de; Carissimi, Alexandre da Silva e Toscani, Simão S. (2002) “Sistemas Operacionais como Programas Concorrentes”, Anais da 2ª Escola Regional de Alto Desempenho, São Leopoldo-RS.

Pressman, R. S. (2001) “Software Engineering: A Practitioner’s Approach”, 5th edition, McGraw-Hill Higher Education.

Rembold, U.; Nnaji, B. O. e Storr, A. (1993) “Computer Integrated Manufacturing and Engineering”, Addison-Wesley Publishing Company.

Silberschatz, A. e Galvin, P. B. (1999) “Operating Systems Concepts”, 5th edition, John Wiley & Sons.

Solomon, D. A. (1998) “Inside Microsoft Windows NT”, 2nd edition, Microsoft Press.

Srinivasan, B.; Pather, S.; Hill, R.; Ansari, F. e Niehaus, D. (1998) “A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software”, Proceedings of the IEEE Real-Time Technology and Applications Symposium.

Stallings, W. (2001) “Operating Systems”, 4th edition, Prentice-Hall.

Stankovic, J. A. (1988) “Misconceptions about real-time computing”, IEEE Computer, vol 21 (10).

Tanenbaum, A. S. (1992) “Modern Operating Systems”, Prentice Hall.

Tanenbaum, A. S. (1995) “Distributed Operating Systems”, Prentice-Hall.

Tanenbaum, A. S. e Woodhull, A.S. (2000) “Sistemas Operacionais: Projeto e Implementação”, 2ª edição, Bookman.

Timmeman, M.; Beneden, B. V. e Uhres, L. (1998) “RTOS Evaluation Kick Off”, Real-Time Magazine, 1998-Q33, http://www.realtime-info.be, (atualmente Dedicated Systems Magazine).

Vahalia, U. (1996) “Unix Internals: The New Frontiers”, Prentice-Hall.

Van Der Steen, Aaa J. e Dongarra, Jack J. (1996) “Overview of recent supercomputer”, Technical Report UT-CS-96-325, Department of Computer Science, University of Tennessee.

Vieira, J. E. (1999) “LINUX-SMART: Melhoria de Desempenho para Aplicações Real-Time Soft em Ambiente Linux”, Dissertação de mestrado, Instituto de Matemática e Estatística, Universidade de São Paulo.

Wang, Y.-C. e Lin, K.-J. (1999) “Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel”, Proceedings of the IEEE Real-Time Systems Symposium.