Capítulo 3

Suportes para Aplicações de Tempo Real

Em geral, aplicações são construídas a partir dos serviços oferecidos por um sistema operacional. No caso das aplicações de tempo real, o atendimento dos requisitos temporais depende não somente do código da aplicação, mas também da colaboração do sistema operacional no sentido de permitir previsibilidade ou pelo menos um desempenho satisfatório. Muitas vezes os requisitos temporais da aplicação são tão rigorosos que o sistema operacional é substituído por um simples núcleo de tempo real, o qual não inclui serviços como sistema de arquivos ou gerência sofisticada de memória. Núcleos de tempo real oferecem uma funcionalidade mínima, mas são capazes de apresentar excelente comportamento temporal em função de sua simplicidade interna.

Este capítulo discute aspectos de sistemas operacionais cujo propósito é suportar aplicações de tempo real. O objetivo inicial é estabelecer as demandas específicas das aplicações de tempo real sobre o sistema operacional e definir a funcionalidade mínima que vai caracterizar os núcleos de tempo real. Também são apresentadas algumas soluções existentes no mercado. Uma questão importante a ser discutida é a capacidade dos sistemas operacionais de serem analisados com respeito a escalonabilidade, como discutido no capítulo anterior.

3.1 Introdução

Assim como aplicações convencionais, aplicações de tempo real são mais facilmente construídas se puderem aproveitar os serviços de um sistema operacional. Desta forma, o programador da aplicação não precisa preocupar-se com a gerência dos recursos básicos (processador, memória física, controlador de disco). Ele utiliza as abstrações de mais alto nível criadas pelo sistema operacional (tarefas, segmentos, arquivos).

Sistemas operacionais convencionais encontram dificuldades em atender as demandas específicas das aplicações de tempo real. Fundamentalmente, sistemas operacionais convencionais são construídos com o objetivo de apresentar um bom comportamento médio, ao mesmo tempo que distribuem os recursos do sistema de forma eqüitativa entre as tarefas e os usuários. Em nenhum momento existe uma preocupação com previsibilidade temporal. Mecanismos como caches 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 uma

tarefa em particular frente às restrições temporais.

Aplicações com restrições de tempo real estão menos interessadas em uma distribuição uniforme dos recursos do sistema e mais interessadas em atender requisitos tais como períodos de ativação e *"deadlines"*.

O atendimento de tais requisitos, em geral, demanda cuidados na gerência dos recursos do sistema que não são tomados em sistemas operacionais convencionais. Um exemplo claro é o conceito de "inversão de prioridade" (ver capítulo 2), o qual é muito importante no contexto de tempo real mas completamente ignorado em sistemas operacionais convencionais.

Sistemas operacionais de tempo real ou SOTR são sistemas operacionais onde especial atenção é dedicada ao comportamento temporal. Em outras palavras, são sistemas operacionais cujos serviços são definidos não somente em termos funcionais mas também em termos temporais. Estes aspectos serão discutidos na seção 3.3.

Além do aspecto temporal, algumas funcionalidades específicas são normalmente exigidas em um SOTR. Estas exigências decorrem do fato da maioria das aplicações de tempo real serem construídas como programas concorrentes. Logo, é imperativo que o SOTR forneça as abstrações necessárias, isto é, tarefas e mecanismos para comunicação entre tarefas. A seção 3.2 discute os aspectos funcionais dos SOTR.

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 SOTR onde a aplicação vai executar quando concluída. Por exemplo, pode ser o computador embarcado em um telefone celular. A plataforma de desenvolvimento inclui o hardware e o SO onde o sistema é desenvolvido, 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. Tipicamente a plataforma de desenvolvimento é usada enquanto for possível, sendo as etapas finais de depuração realizadas na plataforma alvo. Em qualquer caso, o objetivo deste capítulo é analisar sistemas operacionais de tempo real unicamente no papel de plataforma alvo.

3.2 Aspectos Funcionais de um Sistema Operacional Tempo Real

Como qualquer sistema operacional, um SOTR procura tornar a utilização do computador mais eficiente e mais conveniente. A utilização mais eficiente significa mais trabalho obtido a partir do mesmo hardware. Isto é obtido através da distribuição dos recursos do hardware entre as tarefas. Uma utilização mais conveniente diminui o

tempo necessário para a construção dos programas. Isto implica na redução do custo do software, na medida em que são necessárias menos horas de programador. Por exemplo, através de funções que simplificam o acesso aos periféricos, escondendo os detalhes do hardware, ou ainda funções que gerenciam o espaço em disco ou na memória principal, livrando o programador da aplicação deste trabalho.

Em geral, as facilidades providas por um sistema operacional de propósito geral são bem vindas em um SOTR. O objetivo deste capítulo não é descrever sistemas operacionais em geral, mas sim tratar dos serviços que são fundamentais para um SOTR. Desta forma, esta seção trata apenas dos seguintes aspectos: tarefas e *"threads"*, a comunicação entre elas, instalação de tratadores de dispositivos e interrupções e a disponibilidade de temporizadores.

Entretanto, é bom lembrar que a maioria das aplicações tempo real possui uma parte (talvez a maior parte) de suas funções sem restrições temporais. Logo, é preciso considerar que o SOTR deveria, além de satisfazer as necessidades das tarefas de tempo real, fornecer funcionalidade apropriada para as tarefas convencionais. Aspectos como suporte para interface gráfica de usuário, protocolos de comunicação para a Internet, fogem do escopo de um SOTR que execute apenas tarefas de tempo real. Porém, são aspectos importantes quando considerado que uma aplicação tempo real também possui vários componentes convencionais.

3.2.1 Tarefas e *"Threads"*

Um programa que é executado por apenas uma tarefa é chamado de programa sequencial. A grande maioria dos programas escritos são programas sequenciais. Neste caso, existe somente um fluxo de controle durante a execução. Um programa concorrente é executado simultaneamente por diversas tarefas que cooperam entre si, isto é, trocam informações. Neste contexto trocar informações significa trocar dados ou realizar algum tipo de sincronização. É necessário a existência de interação entre tarefas para que o programa seja considerado concorrente.

Na literatura de sistemas operacionais os termos *tarefa* e *processo* são frequentemente utilizados com o mesmo sentido. Tarefas ou processos são abstrações que incluem um espaço de endereçamento próprio (possivelmente compartilhado), um conjunto de arquivos abertos, um conjunto de direitos de acesso, um contexto de execução formado pelo conjunto de registradores do processador, além de vários outros atributos cujos detalhes variam de sistema para sistema. O tempo gasto para chavear o processador entre duas tarefas é definido por este conjunto de atributos, isto é, o tempo necessário para mudar o conjunto de atributos em vigor.

Uma forma de tornar a programação concorrente ao mesmo tempo mais simples e mais eficiente é utilizar a abstração *"thread"*. *"Threads"* são tarefas leves, no sentido que os únicos atributos particulares que possuem são aqueles associados com o contexto de execução, isto é, os registradores do processador. Todos os demais atributos de uma

"thread" são herdados da tarefa que a hospeda. Desta forma, o chaveamento entre duas *"threads"* de uma mesma tarefa é muito mais rápido que o chaveamento entre duas tarefas. Por exemplo, como todas as *"threads"* de uma mesma tarefa compartilham o mesmo espaço de endereçamento, a MMU (*"memory management unit"*) não é afetada pelo chaveamento entre elas. No restante deste capítulo será suposto que o SOTR suporta *"threads"*. Logo, o termo tarefas será usado para denotar um conjunto de recursos tais como espaço de endereçamento, arquivos, *"threads"*, etc, ao passo que *"thread"* será usado para denotar um fluxo de execução específico.

Aplicações de tempo real são usualmente organizadas na forma de várias *"threads"* ou tarefas concorrentes. Logo, um requisito básico para os sistemas operacionais de tempo real é "oferecer suporte para tarefas e *"threads"*. Embora programas concorrentes possam ser construídos a partir de tarefas, o emprego de *"threads"* aumenta a eficiência do mesmo. Devem ser providas chamadas de sistema para criar e destruir tarefas e *"threads"*, suspender e retomar tarefas e *"threads"*, além de chamadas para manipular o seu escalonamento.

Durante a execução da aplicação as *"threads"* passam por vários estados. A figura 3.1 procura mostrar, de maneira simplificada, quais são estes estados. Após ser criada, a *"thread"* está *pronta* para receber o processador. Entretanto, como possivelmente várias *"threads"* aptas disputam o processador, ela deverá esperar até ser selecionada para execução pelo escalonador. Uma vez selecionada, a *"thread"* passa para o estado *executando*. A *"thread"* pode enfrentar situações de bloqueio quando solicita uma operação de entrada ou saída, ou então tenta acessar uma estrutura de dados que está em uso por outra *"thread"*. Neste momento ela para de executar e passa para o estado *bloqueada*. Quando a causa do bloqueio desaparece, ela volta a ficar pronta para executar. A figura 3.1 diferencia como um tipo especial de bloqueio a situação na qual a *"thread"* solicita sua suspensão por um intervalo de tempo, ou até uma hora futura pré-determinada. Neste caso, a *"thread"* é dita *inativa*. Ela voltará a ficar pronta quando chegar o momento certo. Este estado é típico de uma *"thread"* com execução periódica, quando a ativação atual já foi concluída e ela aguarda o instante da próxima ativação. Observe que *"threads"* periódicas também podem ser implementadas através da criação de uma nova *"thread"* no início de cada período e de sua destruição tão logo ela conclua seu trabalho relativo àquele período. Entretanto, o custo (*overhead*) associado com a criação e a destruição de *"threads"* é maior do que o custo associado com a suspensão e reativação, resultando em um sistema menos eficiente. É importante notar que a figura 3.1 descreve o funcionamento típico de um sistema operacional genérico. Uma descrição exata da semântica de cada estado depende de detalhes que, na prática, variam de sistema para sistema. Por exemplo, quando um escalonamento estático garante que todos os recursos que a *"thread"* precisa para executar estarão disponíveis no momento certo, então ela nunca passará pelo estado bloqueada.

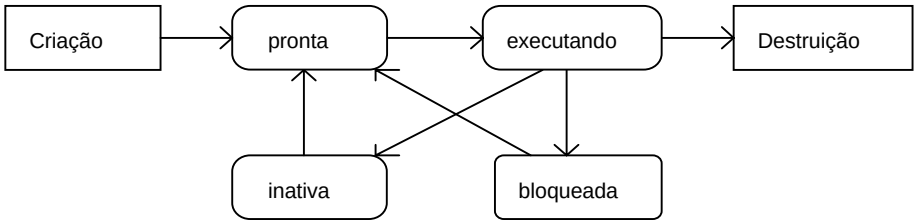

Figura 3.1 - Estados de uma *"thread"*.

Uma questão sempre presente quando o assunto é *"thread"* é a conveniência de implementa-las a nível de *"kernel"* ou a nível de biblioteca. Quando implementadas a nível de biblioteca, uma única *"thread"* reconhecida pelo *"kernel"* é usada para executar diversas *"threads"* da aplicação, através de uma multiprogramação implementada por rotinas ligadas com a aplicação e ignoradas pelo *"kernel"*. Este tipo de *"thread"* oferece um chaveamento de contexto mais rápido e menor custo para criação e destruição. Entretanto, como o *"kernel"* reconhece a existência de apenas uma *"thread"*, se esta ficar bloqueada então todas as *"threads"* da tarefa ficarão bloqueadas. Além disto, a solução a nível de biblioteca não é capaz de aproveitar multiprocessamento quanto este existe. A discussão sobre as vantagens de um tipo e outro é longa e pode ser encontrada em livros de sistemas operacionais tal como [SiG98]. Neste texto é suposto que as *"threads"* são implementadas pelo *"kernel"*, o qual é também responsável pelo seu escalonamento.

3.2.2 Comunicação entre Tarefas e *"Threads"*

Uma aplicação tempo real é tipicamente um programa concorrente, formado por tarefas e *"threads"* que se comunicam e se sincronizam. A literatura sobre sistemas operacionais convencionais trata este assunto com bastante detalhe. Existem duas grandes classes de soluções para a construção de um programa concorrente: troca de mensagens e variáveis compartilhadas.

A troca de mensagens é baseada em duas operações simples: "enviar mensagem" e "receber mensagem". Na verdade estas duas operações oferecem a possibilidade de um grande número de variações, na medida em que são alteradas características como forma de endereçamento, armazenamento intermediário, situações de bloqueio, tolerância a falhas, etc. Uma descrição completa do mecanismo foge ao escopo deste livro e pode ser encontrada em [SiG98] ou [TaW97]. O importante é notar que tanto a comunicação como a sincronização são feitas através das mesmas operações. Enquanto a comunicação acontece através da mensagem enviada, a sincronização acontece através do bloqueio da *"thread"* até que ela receba uma mensagem ou até que a mensagem enviada por ela seja lida pela *"thread"* destinatária. Esta solução é especialmente conveniente em sistemas distribuídos.

Nas soluções baseadas em variáveis compartilhadas o sistema operacional oferece,

além da memória compartilhada para hospedar estas variáveis, algum mecanismo de sincronização auxiliar. A comunicação entre *"threads"* acontece através da leitura e escrita de um conjunto compartilhado de variáveis. A sincronização deve ser explícita, através de semáforos, monitores, *"spin-locks"*, regiões críticas condicionais, ou algum outro mecanismo deste tipo (uma descrição destes mecanismos pode ser encontrada em [SiG98]). Em geral esta solução somente é possível quando as *"threads"* envolvidas executam no mesmo computador. Embora existam implementações de memória virtual compartilhada, onde o mecanismo de memória virtual é usado para criar a ilusão de uma memória compartilhada que não existe no hardware, este mecanismo possui um custo elevado e não é viável em sistemas de tempo real.

Estas duas classes de soluções são equivalentes no sentido de que qualquer programa concorrente pode ser implementado com uma ou com outra. Entretanto, a forma como o programa concorrente é estruturado deve levar em conta o mecanismo de comunicação adotado. Em geral, memória compartilhada é mais eficiente que troca de mensagens. Com memória compartilhada não existe a necessidade de copiar os dados da memória do remetente para uma área do sistema operacional e depois para a memória do destinatário. As tarefas alteram diretamente as variáveis compartilhadas. Entretanto, em sistemas distribuídos mensagens são a forma natural de comunicação. Muitos sistemas operacionais oferecem os dois mecanismos.

Observe que a correta programação da comunicação e sincronização das tarefas em um programa concorrente garante o seu comportamento funcional, mas não temporal. Isto é, o resultado lógico do programa será sempre correto, independentemente da ordem na qual as tarefas são executadas. Entretanto, o correto comportamento temporal vai depender do escalonamento das tarefas e da capacidade do hardware de cumprir com os requisitos temporais.

No sentido inverso, é possível afirmar que algumas soluções de escalonamento tempo real já citadas resolvem também o problema de sincronização entre tarefas e *"threads"*. Por exemplo, o escalonamento baseado em executivo cíclico (ver capítulo 2) determina "que tarefa executa quando" ainda em tempo de projeto. A construção da grade de execução pode ser feita de tal forma que, além de atender aos requisitos temporais, ela também resolve os problemas de exclusão mútua e relações de precedência entre as tarefas. Neste caso, não existe a necessidade de mecanismos explícitos de sincronização.

3.2.3 Instalação de Tratadores de Dispositivos

Freqüentemente os sistemas de tempo real lidam com periféricos especiais, diferentes daqueles normalmente encontrados em computadores de escritório. Por exemplo, aplicações visando automação industrial ou o controle de equipamentos em laboratório empregam diferentes tipos de sensores e atuadores. Algumas vezes dispositivos são desenvolvidos sob medida para o projeto em questão.

É fundamental que o projetista da aplicação possa desenvolver os seus próprios tratadores de dispositivos (*"device drivers"*) e, de alguma forma, incorpora-los ao sistema operacional. Esta é na verdade uma técnica usual. Normalmente os vendedores de periféricos fornecem, além do periférico propriamente dito, tratadores apropriados para os sistemas operacionais mais populares. No caso dos sistemas operacionais de tempo real a situação é mais complexa pois:

- Se o SOTR usado não for bastante conhecido, o fornecedor do periférico poderá não fornecer um tratador apropriado para ele. Isto obriga o projetista da aplicação a portar para o SOTR usado um tratador para aquele periférico que tenha sido originalmente desenvolvido para outro sistema operacional.
- Se o periférico for desenvolvido sob medida, então um tratador de dispositivo apropriado para ele terá que ser desenvolvido.

Muitas vezes a aplicação e o periférico estão fortemente integrados, e o código da aplicação confunde-se com o código do que seria o tratador do dispositivo. Isto acontece principalmente no contexto dos sistemas embutidos (*"embedded systems"*). Neste caso é importante que o SOTR permita que a aplicação instale os seus próprios tratadores de interrupções, para que as interrupções do periférico sejam atendidas pelo código apropriado. Em sistemas operacionais de propósito geral, multi-usuários, a instalação de tratadores de interrupção é considerada uma operação perigosa e, portanto, permitida apenas ao próprio *"kernel"* do sistema operacional ou tarefas especiais. Sistemas operacionais de tempo real frequentemente executam apenas uma aplicação, para um único usuário, que é o conhecedor da aplicação e de suas necessidades. Desta forma, a instalação de tratadores de interrupção por tarefas normais passa a ser aceitável.

3.2.4 Temporizadores

Embora seja possível conceber uma aplicação tempo real que nunca precise "ler a hora" ou "aguardar um certo intervalo de tempo", esta não é a situação mais comum. Tipicamente as aplicações precisam realizar operações que envolvem a passagem do tempo, tais como:

- Ler a hora com o propósito de atualizar um histórico;
- Realizar determinada ação a cada X unidades de tempo (ativação periódica);
- Realizar determinada ação depois de Y unidades de tempo a partir do instante atual;
- Realizar determinada ação a partir do instante absoluto de tempo Z.

Tais operações permitem a implementação de tarefas periódicas, *"time-outs"*, *"watch-dogs"*, etc. É importante que o SOTR ofereça um conjunto de serviços que atenda estas necessidades. Tipicamente o sistema possui pelo menos um temporizador

(*"timer"*) implementado em hardware, o qual é capaz de gerar interrupções com uma dada frequência. Cabe ao SOTR utilizar este temporizador do hardware para criar a ilusão de múltiplos temporizadores lógicos. A cada interrupção do temporizador em hardware o SOTR atualiza cada um dos temporizadores lógicos e executa as operações necessárias.

Uma questão importante ligada aos temporizadores é a sua resolução ou granularidade. Esta é dada pela frequência do temporizador do hardware. Para o SOTR o tempo avança de maneira discreta, um período do temporizador do hardware de cada vez. Por exemplo, suponha que este gere uma interrupção a cada 100 ms, isto é, sua resolução é de 100 ms. Se uma tarefa qualquer solicitar ao SOTR que determinada rotina seja executada uma vez a cada 1250 ms, na verdade o intervalo de tempo entre duas ativações sucessivas desta rotina será de 1200 ms ou de 1300 ms. Dependendo da implementação do SOTR, a rotina será ativada uma vez a cada 1200 ms, uma vez a cada 1300 ms ou ainda através da intercalação de intervalos de tempo de 1200 ms e 1300 ms. Esta última forma é provavelmente a melhor, pois resulta em um tempo médio de espera que aproxima-se dos 1250 ms. Em geral a resolução pode ser aumentada alterando-se a programação do temporizador em hardware. Entretanto, o tratador destas interrupções, que implementa os diversos temporizadores lógicos, representa um custo (*"overhead"*) para o sistema. Aumentar a frequência das interrupções significa aumentar este custo. O importante é que a resolução dos temporizadores seja apropriada para a aplicação em questão, isto é, os erros devido a granularidade dos relógios do sistema possam ser desprezados.

Existem outras fontes de imprecisão além da resolução do temporizador no hardware. Mesmo que a tarefa solicite sua suspensão pelo tempo equivalente a um número inteiro de interrupções, a solicitação pode ocorrer no meio entre duas interrupções. Além disto, quando passar o tempo estipulado, a fila do processador pode incluir tarefas ou *"threads"* com prioridade mais alta. Em resumo, qualquer temporização realizada pelo SOTR será sempre uma aproximação. Ela será tão mais exata quanto maior for a resolução do temporizador em hardware e maior for a prioridade da ação associada com a temporização.

Um serviço adicional que o SOTR pode disponibilizar é a sincronização da hora do sistema com a UTC (*"Universal Time Coordinated"*). Os cristais de quartzo utilizados nos computadores como fonte para as oscilações que permitem medir a passagem do tempo são imprecisos. Desta forma, alguma referência externa deve ser usada para manter o relógio do sistema sincronizado com a UTC. Uma forma cada vez mais popular é o sistema GPS (*"Global Possitioning System"*), que utiliza uma antena local recebendo sinais de satélites. Embora o posicionamento da antena seja crítico neste tipo de sistema, ele é relativamente barato e fácil de instalar.

No caso de um sistema distribuído, cabe também ao sistema operacional manter os relógios dos diferentes computadores sincronizados entre si. É claro que, se cada computador do sistema possuir o seu próprio receptor GPS, a sincronização entre eles será automática. Entretanto, em função do custo e da necessidade de colocar antenas, tipicamenrte apenas alguns computadores da rede terão seu relógio sincronizado com o

mundo exterior. Os demais deverão sincronizar-se com estes, através de algum protocolo. Exemplos podem ser encontrados em [VRC97] e [FeC97]. Todo protocolo de sincronização de relógios deixa um erro residual. A implementação deste protocolo no *"kernel"* do sistema operacional deixa um erro residual menor do que quando ele é implementado pela própria aplicação, pois o código do *"kernel"* está menos sujeito a interferências durante a sua execução do que o código da aplicação. Existem também soluções de sincronização de relógio via hardware, as quais são mais caras porém muito mais precisas.

Em geral programas também precisam de rotinas para manipular datas e horas em formatos tradicionais. Por exemplo, converter entre diferentes formatos e realizar operações aritméticas como calcular a diferença em horas entre duas datas ou calcular em que dia da semana caiu determinada data. Tipicamente estes serviços não são providos pelo SOTR e sim por bibliotecas associadas com o suporte de execução da linguagem de programação usada. Como formatos para data e hora podem variar de linguagem de programação para linguagem de programação, tais operações são melhor providas pelo suporte da própria linguagem.

3.3 Aspectos Temporais de um Sistema Operacional Tempo Real

Além dos aspectos funcionais, também presentes em sistemas operacionais de propósito geral, os aspectos temporais de um SOTR são muito importantes. Eles estão relacionados com a capacidade do SOTR fornecer os mecanismos e as propriedades necessários para o atendimento dos requisitos temporais da aplicação tempo real. O propósito desta seção é estabelecer alguns critérios para avaliar a qualidade temporal de um dado sistema operacional.

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. O projetista da aplicação pode ignorar completamente a função desta rotina, mas não pode ignorar o seu efeito temporal, isto é, a interferência que ela causa na execução da aplicação. Esta interferência deve ser de alguma forma incluída nos testes de escalonabilidade descritos no capítulo 2.

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"*.

Em resumo, com respeito ao comportamento temporal do sistema, qualquer análise deve considerar conjuntamente aplicação e sistema operacional. Isto equivale a dizer que 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, com diferentes classes de requisitos temporais, 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. O capítulo 2, ao explorar as soluções de escalonamento tempo real existentes, deu uma idéia da diversidade existente nesta área.

Neste ponto é importante destacar que a teoria de tempo real, descrita no capítulo anterior, é recente. Embora a referência mais antiga seja sempre [LiL73], somente na década de 90 os modelos de tarefas suportados pela teoria de escalonamento foram estendidos a ponto de tornarem-se verdadeiramente úteis. Estes avanços da teoria estão sendo gradativamente absorvidos pelos desenvolvedores de SOTR. Entretanto, ainda hoje (início de 2000), existe uma distância entre a teoria de escalonamento e a prática no desenvolvimento de sistemas de tempo real. O texto deste capítulo reflete esta dicotomia. 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".

3.3.1 Limitações dos Sistemas Operacionais de Propósito Geral

As aplicações tempo real desenvolvidas estão cada vez mais complexas, o que exige uma sempre crescente funcionalidade do suporte de tempo real. Por exemplo, é cada vez mais comum a necessidade de interfaces gráficas, conexão via rede local ou mesmo Internet, algoritmos de controle mais inteligentes.

Ao mesmo tempo, existem razões de ordem econômica para a utilização de soluções de prateleira (*"off-the-shelf"*). Em particular, usar um sistema operacional de propósito geral no projeto significa usar um sistema operacional tipicamente mais barato, para o qual existe uma grande quantidade de ambientes de desenvolvimento e também é mais fácil contratar programadores experientes.

O que impede o emprego de sistemas operacionais de propósito geral são as restrições temporais da aplicação. Assim, uma das primeiras decisões que o projetista de uma aplicação tempo real deve tomar é: "É realmente necessário usar um SOTR" ?

A dificuldade desta decisão é minimizada pela existência de uma classe de SOTR que é simplesmente um sistema operacional popular adaptado para o contexto de tempo real. Estes sistemas foram adaptados no sentido de mostrar alguma preocupação com a

resposta em tempo real. O resultado final obtido com eles é melhor do que quando um sistema operacional de propósito geral é utilizado, mas não é capaz de oferecer previsibilidade determinista.

Independentemente do contexto em questão, diversas técnicas populares em sistemas operacionais de propósito geral são especialmente problemáticas 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 uma *"thread"*. Os mecanismos tradicionais usados em sistemas de arquivos, tais 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 (não usar memória virtual);
- Usando o mecanismo apenas em tarefas sem requisitos temporais rigorosos (acesso a disco feito por tarefas sem requisitos temporais, ou apenas requisitos brandos).

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.

Outro aspecto central é o algoritmo de escalonamento empregado. Tipicamente qualquer sistema operacional dispõe de escalonamento baseado em prioridades. Neste caso, bastaria que a prioridade das *"threads"* de tempo real fossem mais elevadas do que as *"threads"* associadas com tarefas convencionais (*"time-sharing"* e *"background"*). Entretanto, a maioria dos sistemas operacionais de propósito geral inclui mecanismos que reduzem automaticamente a prioridade de uma *"thread"* na medida que ela consome tempo de processador. Este tipo de mecanismo é utilizado para favorecer as tarefas com ciclos de execução menor e diminuir o tempo médio de resposta no sistema. Entretanto, em sistemas de tempo real a justa distribuição de recursos entre as tarefas é menos importante do que o atendimento dos requisitos temporais.

Considere, por exemplo, o algoritmo de escalonamento utilizado em versões tradicionais do sistema operacional Unix, tais como o Unix System V release 3 (SVR3) ou o Berkeley Software Distribution 4.3 (4.3BSD).

O Unix tradicional emprega prioridade variável. Quando uma tarefa é liberada e possui prioridade maior do que a tarefa que está executando, existe um chaveamento de contexto e a tarefa recém liberada passa a ser executada. Tarefas com a mesma prioridade dividem o tempo do processador através do mecanismo de fatias de tempo. Entretanto, duas características tornam este sistema problemático para tempo real.

Primeiro, a prioridade de cada tarefa varia conforme o seu padrão de uso do processador, como será descrito a seguir. Depois, uma tarefa executando código do *"kernel"* não pode ser preemptada. Desta forma, a latência até o disparo de uma tarefa de alta prioridade inclui o maior caminho de execução existente dentro do *"kernel"*.

As prioridades variam entre 0 e 127, onde um número menor representa prioridade mais alta. Os valores entre 0 e 49 são reservados para tarefas executando código do *"kernel"*. Os valores entre 50 e 127 são para tarefas em modo usuário.

O descritor de tarefa contém, entre outras informações:

- A prioridade atual da tarefa, p_pri;
- A prioridade desta tarefa quando em modo usuário, p_usrpri;
- Uma medida da utilização recente de processador por esta tarefa, p_cpu;
- Um fator de gentileza definido pelo programador ou administrador do sistema, p_nice;

Quando em modo usuário, a tarefa possui sua prioridade definida por p_usrpri, isto é, p_pri = p_usrpri. Quando uma tarefa é liberada dentro do *"kernel"* após ter acordado de um bloqueio ela recebe um "empurrão temporário", na forma de uma prioridade p_pri que é mais alta (número menor) do que o seu p_usrpri. Cada razão de bloqueio tem uma *"sleep priority"* associada, a qual determina a "força do empurrão". Por exemplo, a prioridade após ficar esperando por entrada de terminal é 28 e a prioridade após ficar esperando por um acesso ao disco é 20, independentemente do que a tarefa faz ou de seus requisitos temporais, caso eles existam. Quando uma tarefa acorda, p_pri recebe a *"sleep priority"* correspondente ao bloqueio. A prioridade da tarefa retornará para o valor p_usrpri quando esta voltar para modo usuário.

O valor de p_usrpri depende dos valores de p_cpu e de p_nice da tarefa em questão. O fator de gentileza é um número entre 0 e 39, cujo valor padrão (*"default"*) é 20. O valor de p_cpu inicial é zero na criação da tarefa. A cada interrupção do temporizador (*"tick"*), o valor p_cpu da tarefa em execução naquele instante é incrementado, até um valor máximo de 127.

Simultaneamente, a cada segundo, os valores p_cpu de todas as tarefas são reduzidos por um fator de decaimento (*"decay factor"*). Por exemplo, são multiplicados por 1/2, ou são multiplicados por decay, onde
decay = (2 * load_average) / (2 * load_average + 1) e o valor load_average é o número médio de tarefas aptas a executar dentro do último segundo.

A prioridade da tarefa quando executando código da aplicação é calculada através da fórmula p_usrpri = 50 + (p_cpu / 4) + (2 * p_nice). Em função deste recalculo, pode haver um chaveamento de contexto. Isto acontece quando a tarefa em execução fica com prioridade mais baixa do que qualquer outra tarefa apta a executar, considerando-se os novos valores de p_usrpri de todas as tarefas.

A solução de escalonamento do SVR3 e do 4.3BSD é engenhosa e permite um bom

desempenho quando as maiores preocupações são a justa distribuição de recursos entre as tarefas e o tempo médio de resposta do sistema. Entretanto, quando a qualidade temporal de sistema é avaliada em termos do número de *"deadlines"* cumpridos ou do atraso médio em relação ao *"deadline"* de cada tarefa, esta solução não é mais apropriada. Seu comportamento depende em grande parte da dinâmica do sistema, tirando do projetista o poder de controlar com maior exatidão o tratamento que cada tarefa recebe. Um dos aspectos centrais de um SOTR é usar um algoritmo de escalonamento que permita ao programador um controle maior sobre a execução das tarefas.

3.3.2 Chaveamento de Contexto e Latência de Interrupção

Fornecedores de SOTR costumam divulgar métricas para mostrar como o sistema em questão é mais ou menos apropriado para suportar aplicações de tempo real. Estas métricas refletem a prática da construção de aplicações tempo real nas últimas décadas, e muitas vezes estão mais ligadas à desempenho do que ao cumprimento de restrições temporais. A seção 3.3.2 descreve as métricas como encontradas na literatura. A seção 3.3.3 procura analisar a sua relação com o tempo de resposta das tarefas.

Uma métrica muito utilizada é o tempo para chaveamento entre duas tarefas. Este tempo inclui salvar os registradores da tarefa que está executando e carregar os registradores com os valores da nova tarefa, incluindo qualquer informação necessária para a MMU (*"memory management unit"*) funcionar corretamente. Em geral esta métrica não inclui o tempo necessário para decidir qual tarefa vai executar, uma vez que isto depende do algoritmo de escalonamento utilizado.

Outra métrica frequentemente utilizada pelos fornecedores de SOTR é a latência até o inicio do tratador de uma interrupção do hardware. Imagina-se que eventos importantes e urgentes no sistema serão sinalizados por interrupções de hardware e, desta forma, é importante iniciar rapidamente o tratamento destas interrupções. Observe que, no caso mais simples, é suposto que a rotina que trata a interrupção é capaz de sozinha gerar uma resposta apropriada para o evento sinalizado. Colocar código da aplicação no tratador de interrupções é uma solução perigosa, pois este código executa com direitos totais dentro do *"kernel"*. Entretanto, esta forma permite respostas extremamente rápidas da aplicação à um evento externo. Do ponto de vista da análise de escalonabilidade, este tratador de interrupções corresponderia a uma tarefa com a prioridade mais alta do sistema. Ele gera interferência sobre as demais tarefas, mas não recebe interferência de nenhuma outra. A latência de interrupção equivale ao *"release jitter"* desta tarefa especial.

A latência no disparo de um tratador de interrupção inclui o tempo que o hardware leva para realizar o processamento de uma interrupção, isto é, salvar o contexto atual (*"program counter"* e *"flags"*) e desviar a execução para o tratador da interrupção. Também é necessário incluir o tempo máximo que as interrupções podem ficar

desabilitadas. Por exemplo, um *"kernel"* que executa com interrupções desabilitadas inclui na latência a maior sequência de instruções que podem ser executadas dentro dele, tipicamente uma chamada de sistema complexa. Por outro lado, esta solução tem a vantagem de permitir ao tratador da interrupção acessar livremente as estruturas de dados do *"kernel"*, pois é garantido que estão em um estado consistente quando ele é ativado. Este acesso ocorre quando o tratador de interrupção necessita algum serviço do *"kernel"*, como liberar uma tarefa para execução futura.

Sistemas mais modernos, como o Unix SVR4, utilizam *"kernel"* com pontos de preempção previamente programados. Desta forma, quando uma interrupção de hardware é acionada e a tarefa executando está dentro do *"kernel"*, o tratador da interrupção não precisará esperar que a tarefa executando complete a chamada, mas apenas chegue no próximo ponto de preempção, quando o chaveamento de contexto ocorrerá. Estes pontos de preempção são colocados no código do *"kernel"* em pontos onde suas estruturas de dados estão consistentes. Desta forma, o tratador da interrupção pode acessar livremente as estruturas de dados do *"kernel"*, desde que também as deixe em um estado consistente ao final de sua execução.

Finalmente, a forma mais apropriada para um SOTR, é usar um *"kernel"* completamente interrompível. Desta forma, não importa se a tarefa em execução está ou não executando código do *"kernel"*, o tratador de interrupção é imediatamente disparado. Se as estruturas de dados do *"kernel"* são compartilhadas entre a tarefa executando o código do *"kernel"* e o tratador de interrupções, elas devem ser protegidas por algum mecanismo de sincronização. Por exemplo, interrupções podem ser desabilitadas pela tarefa somente enquanto ela estiver acessando uma estrutura de dados usada pelo tratador de interrupções.

A figura 3.2 ilustra as 3 situações. Na situação (A) o *"kernel"* executa com interrupções desabilitadas e o tratador da interrupção somente é ativado quando a tarefa em execução deixa o *"kernel"*. Na situação (B) o tratador da interrupção é acionado assim que a tarefa chega no próximo ponto de interrupção. Após a execução do tratador da interrupção a tarefa retoma a sua execução. Na situação (C) o tratador é ativado o mais cedo possível, depois do que a tarefa retoma a sua execução dentro de um *"kernel"* que pode ser interrompido a qualquer momento. Obviamente, a situação (C) oferece a menor latência até o início do tratador de interrupção.

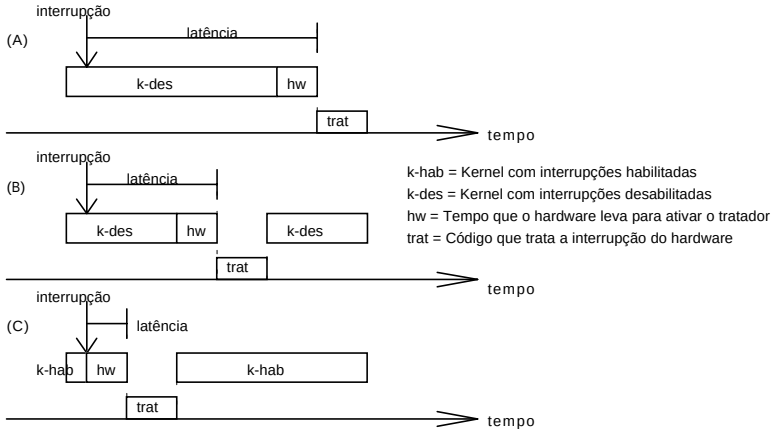

Figura 3.2 - Formas do *"kernel"* lidar com interrupções de hardware.

3.3.3 Relação entre Métricas e Tempo de Resposta

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, conhecendo-se o programa fonte do *"kernel"*. Além disto, estes valores são necessários no momento de aplicar os testes de escalonabilidade descritos no capítulo anterior. 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. Po 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. Todos estes fatores devem ser computados e somente a aplicação correta de um teste de escalonabilidade será capaz de determinar se as restrições temporais serão ou não cumpridas.

Uma questão importante a ser analisada é a relação entre a latência até o início do tratador de interrupção e o tempo total de resposta, ou seja, o tempo entre a sinalização do evento através de uma interrupção de hardware e a respectiva resposta do sistema ao ambiente. Esta relação depende do nível de complexidade do tratador de interrupções. Em linhas gerais podemos classificar os tratadores de interrupções em quatro tipos:

- Tratador simples, executa código da aplicação e não precisa de qualquer serviço ou estrutura de dados do *"kernel"*;
- Tratador complexo, executa código da aplicação mas necessita serviço ou estrutura de dados do *"kernel"*;
- Tratador apenas libera tarefa simples da aplicação, que não utiliza o *"kernel"*;

- Tratador libera tarefa complexa da aplicação, que utiliza o *"kernel"*.

Como o tratador simples não acessa rotinas ou estruturas de dados do *"kernel"*, não é necessário qualquer cuidado com o estado do *"kernel"* no momento que o tratador é ativado. Como o próprio nome indica, esta é a situação mais simples. A figura 3.2 e o texto da seção anterior foram construídos supondo este tipo de tratador.

No caso de um tratador complexo, é necessário assegurar que as estruturas de dados do *"kernel"* estão consistentes no momento que ele é ativado. No caso do *"kernel"* que não pode ser interrompido, não existe problema. Quando o *"kernel"* pode ser interrompido apenas em pontos previamente definidos, é necessário tomar o cuidado de posicionar tais pontos em locais do código do *"kernel"* onde as estruturas de dados estão consistentes e podem ser acessadas pelo tratador da interrupção. Finalmente, no caso de um *"kernel"* que pode ser interrompido a qualquer momento, todas as estruturas de dados passíveis de acesso pelo tratador devem ser protegidas de alguma forma, como por exemplo desabilitar interrupções.

Ainda considerando o tratador complexo em *"kernel"* totalmente interrompível, para que o tratador da interrupção possa ficar bloqueado a espera da liberação de uma estrutura de dados, é possível associar ao tratador de interrupções uma semântica de *"thread"*. Por isto, em sistemas onde o *"kernel"* pode ser interrompido a qualquer momento e o tratador da interrupção necessita acessar estruturas de dados do *"kernel"*, a solução mais elegante é fazer com que o tratador de interrupções apenas libere uma *"thread"* que, ao ser escalonada, fará o acesso ao *"kernel"*. Embora o ato de liberar uma *"thread"* para execução necessite, por si só, acesso a estruturas de dados do *"kernel"*, este acesso é simples e rápido e possíveis inconsistências podem ser evitadas desabilitando-se interrupções por rápidos instantes, quando estas estruturas forem manipuladas. Aliás, este é o único momento no qual as interrupções precisam realmente ficar desabilitadas.

Quando o tratador apenas libera uma tarefa da aplicação, simples ou complexa, seu comportamento é similar àquele que libera uma *"thread"* para executar código do *"kernel"*. Cabe ao código da tarefa da aplicação realmente tratar o evento sinalizado pela interrupção.

Uma vez definidos os quatro tipos de tratadores de interrupção, podemos analisar a relação entre a latência de interrupção e o tempo de resposta do sistema. No caso de um tratador simples, o tempo de resposta é dado pela latência de interrupção somada ao tempo de execução do tratador, pois o seu comportamento equivale ao de uma tarefa com a prioridade mais alta do sistema.

Em alguns sistemas as interrupções de hardware possuem níveis de prioridade. Neste caso, quando o tratador da interrupção X está executando, interrupções de prioridade igual ou inferior estão automaticamente desabilitadas, ao passo que interrupções com prioridade superior continuam habilitadas. Neste caso, o tratador da interrupção X poderá sofrer interferência dos tratadores das interrupções mais importantes, o que aumenta o seu tempo de resposta. A figura 3.3 ilustra as duas

situações.

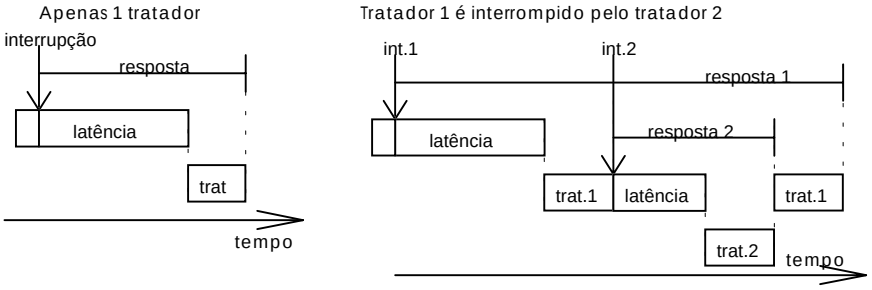

Figura 3.3 - Tempo de resposta de um tratador simples.

O comportamento de um tratador complexo depende da forma como o *"kernel"* foi programado. No caso de *"kernel"* que não admite interrupções ou admite em pontos previamente definidos, o tratador pode acessar livremente o *"kernel"* e, portanto, seu tempo de resposta será dado pela latência de interrupção somada com o seu tempo de execução. No caso de tratador complexo em *"kernel"* que pode sofrer interrupções, o trabalho é realmente feito pela *"thread"* liberada pelo tratador. Do ponto de vista do sistema, o tempo de resposta é definido pela conclusão desta *"thread"*. Além de eventuais interferências de outras *"threads"* com prioridade mais elevada, a *"thread"* liberada poderá enfrentar bloqueios no momento de acessar o *"kernel"*. Isto ocorre quando ela necessita de um recurso que estava alocado para outra *"thread"* no momento que a interrupção de hardware ocorreu. A figura 3.4 ilustra esta situação. O código "trat" inclui a identificação da interrupção que ocorreu, a inclusão da "*thread"* na fila do processador e a execução do escalonador, o qual seleciona a *"thread"* recém liberada para execução e carrega o seu contexto de execução.

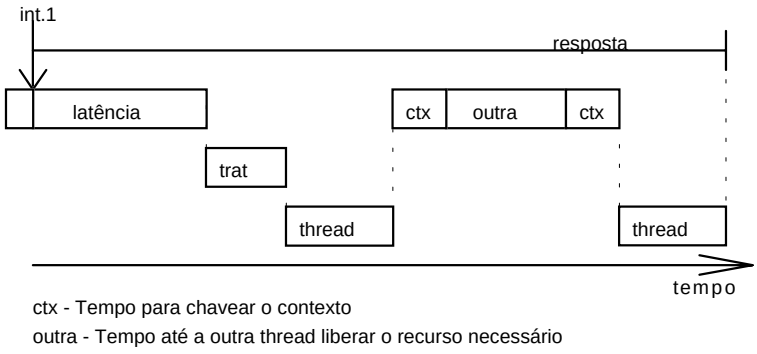

Figura 3.4 - Comportamento de tratador complexo em *"kernel"* que sofre interrupções.

No caso de tratadores que liberam tarefas da aplicação, a situação não é diferente. No caso de tarefa simples, o seu tempo de resposta vai depender das interferências de tarefas com prioridade mais alta, mas não haverá bloqueio devido a conflitos no *"kernel"*, pois a mesma não precisa do *"kernel"* para executar. No caso de tarefas

complexas existe a possibilidade de conflito no *"kernel"* e, portanto, de bloqueios. Inclusive, a tarefa liberada poderá encontrar estruturas de dados do *"kernel"* bloqueadas (*"locked"*) por outras tarefas de mais baixa prioridade. Mas isto ocorre apenas se houver coincidência entre as necessidades da *"thread"* suspensa e da *"thread"* liberada. Observe que se mecanismos tradicionais para sincronização entre *"threads"* forem utilizados, existe a possibilidade de inversão de prioridades (ver capítulo 2).

Como mostrado nesta seção, o tempo de chaveamento e o tempo de latência são importantes mas não contam toda a estória. Um dos maiores problemas atualmente para implementar os algoritmos descritos no capítulo 2 é exatamente o desconhecimento do projetista da aplicação a respeito dos conflitos decorrentes de recursos compartilhados, principalmente a nível de *"kernel"*. Entretanto, fica claro que a teoria de escalonamento apresentada no capítulo anterior pode ser aplicada, desde que conhecidas todas as fontes de interferência e de bloqueio presentes no sistema, tanto a nível de aplicação quanto a nível de *"kernel"*.

3.3.4 Tempo de Execução das Chamadas de Sistema

Uma terceira métrica importante para qualquer SOTR é o tempo de execução de cada uma das chamadas de sistema suportadas. Infelizmente, estes tempos de execução dependem muitas vezes do estado do *"kernel"* quando a chamada é feita. O manual de um SOTR pode informar quanto tempo a chamada de sistema para enviar uma mensagem entre duas tarefas (*"send"*) demora, apenas para a execução do *"send"* na perspectiva da tarefa remetente, em função do cenário encontrado. Por exemplo:

- 5 microsegundos quando não existe tarefa bloqueada esperando por ter executado um *"receive"*;
- 7 microsegundos quando existe tarefa a ser liberada;
- 16 microsegundos quando existe tarefa mais prioritária bloqueada esperando pela mensagem e a manipulação de filas é maior.

Para calcular o tempo de resposta do sistema no pior caso é necessário ser pessimista e considerar que o *"kernel"* estará no estado que resulta no maior tempo de execução possível para a chamada em questão. No exemplo, a análise de escalonabilidade vai assumir que um *"send"* demora 16 microsegundos para a tarefa remetente. Quanto mais complexo for o SOTR, mas difícil será calcular estes tempos. Além disto, eles dependem da arquitetura onde o sistema é executado. Na prática tais valores raramente estão disponíveis para o projetista da aplicação.

3.3.5 Outras Métricas

As seções 3.2 e 3.3 deste capítulo procuram destacar o que é normalmente

considerado requisito para um sistema operacional ser considerado de tempo real. Entretanto, esta é uma área na qual não existe consenso absoluto. Na verdade a definição de SOTR depende do tipo de aplicação em questão. Desta forma, os SOTRs herdam das aplicações um enorme conjunto de possibilidades. Um problema adicional está na terminologia usada pelos fornecedores de SOTR. A descrição de cada sistema feita pelo próprio fornecedor é orientada mais pelo marketing do que pelo técnico. Desta forma, é comum encontrar o mesmo termo sendo usado por diferentes fornecedores para conceitos diferentes, ou o mesmo conceito associado com termos diferentes por fornecedores diferentes.

Em [TBU98] é apresentado um programa de avaliação de SOTR independente de fornecedor que define como requisitos mínimos:

- Multi-tarefas ou *"multi-threads"* com escalonamento baseado em prioridade preemptiva.
- Prioridades são associadas com a execução das tarefas ou *"threads"*, e deve existir um número suficiente de níveis de prioridades diferentes para atender a aplicação alvo.
- Deve incluir um mecanismo de sincronização entre *"threads"* com comportamento previsível.
- Deve existir algum mecanismo para prevenir a inversão de prioridades.
- O comportamento do sistema operacional em termos de métricas deve ser conhecido e previsível, para todos os possíveis cenários de carga.

Em função do exposto nas seções anteriores deste capítulo podemos também listar uma série de propriedades e métricas importantes no momento de selecionar um sistema operacional que deverá suportar aplicações de tempo real. As mais importantes são:

- É possível desativar todos aqueles mecanismos que tornam o comportamento temporal menos previsível, sendo memória virtual o exemplo típico ?
- Os tratadores de dispositivo atendem as requisições conforme as prioridades da aplicação ou simplesmente pela ordem de chegada ?
- A mesma questão pode ser feita com respeito aos módulos do sistema responsáveis pela alocação de memória e pelo sistema de arquivos.
- O *"kernel"* do sistema pode ser interrompido a qualquer momento para a execução de um tratador de interrupção ?
- Uma *"thread"* executando código do *"kernel"* pode ser preemptada por outra *"thread"* de prioridade mais alta, quando esta outra deseja executar código da aplicação ?
- Uma *"thread"* executando código do *"kernel"* pode ser preemptada por outra *"thread"* de prioridade mais alta, quando esta outra deseja fazer uma chamada de sistema ?

- Qual o tempo necessário para chavear o contexto entre duas *"threads"* da mesma tarefa ?
- Qual o tempo necessário para chavear o contexto entre duas *"threads"* de tarefas diferentes ?
- Qual a latência até o início da execução de um tratador de interrupções ?
- Qual o tempo de execução de cada chamada de sistema ?
- Qual o maior intervalo de tempo contínuo no qual as interrupções permanecem desabilitadas ?
- No caso do sistema permitir várias *"threads"* executarem simultaneamente código do *"kernel"*, qual o pior caso de bloqueio associado com as estruturas de dados ?

Um problema sempre encontrado no momento de medir tempos em um SOTR é a necessidade de uma referência temporal confiável. Em geral um SOTR dispõe de temporizadores. Entretanto, muitas vezes a resolução e/ou a precisão não são suficientes para o propósito pretendido. Além disto, a inclusão de código dentro do próprio SOTR para fazer as medições acaba por alterar o comportamento temporal do sistema. As melhores medições são realizadas através de hardware externo ao sistema sendo medido. Este hardware externo fica responsável por observar eventos, medir intervalos de tempo e armazenar a informação para posterior análise. A única alteração necessária no sistema sendo medido é a externalização dos eventos de interesse. Isto pode ser feito através da alteração de valor em um dado pino na porta paralela ou através de um acesso a certo endereço de entrada e saída, quando o barramento estiver sendo monitorado através de um *"bus analyser"*.

No caso de métricas obtidas a partir da instrumentalização do software, é importante observar que na maioria das vezes os tempos medidos variam muito em função da carga e da própria sequência de eventos no momento da medição. Por exemplo, considere a latência associada com o atendimento de uma interrupção de hardware. A figura 3.5 mostra o formato típico de um gráfico onde são colocadas muitas medidas consecutivas desta latência. A diferença entre o menor e o maior valor medido pode alcançar ordens de grandeza. Isto fica claro na figura 3.6, onde aparece a distribuição das medidas ao longo de vários intervalos. Neste caso é possível observar que os intervalos [21,30] e [81,90] dominam o gráfico, indicando que a estrutura interna do SOTR é tal que existem dois cenários típicos que definem o tempo de latência na maioria das vezes. Estes valores em geral dependem da carga no sistema. A distribuição das medidas de latência pode ser afetada pelo aumento da carga. Neste caso, um efeito possível ma figura 3.6 é o deslocamento das barras para a direita, ou seja, em direção a tempos de latência maiores, a medida que a carga aumenta.

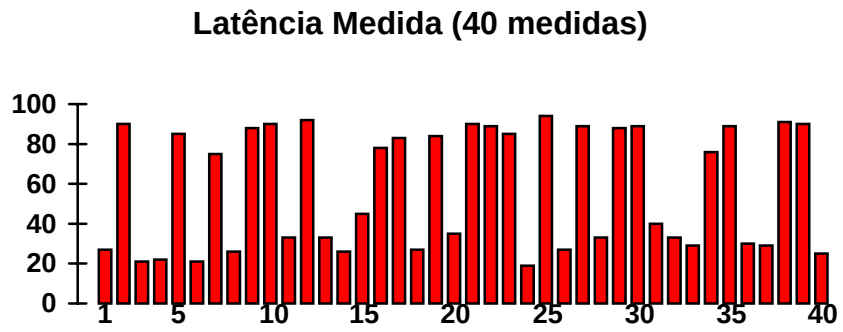

Figura 3.5 - Latência medida em 40 oportunidades consecutivas.

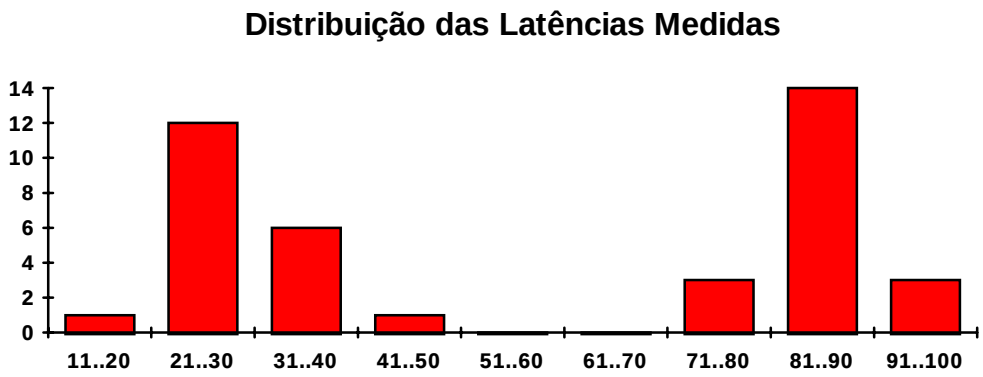

Figura 3.6 - Distribuição das latências medidas conforme sua duração.

A mensagem das figuras 3.5 e 3.6 é clara: simplesmente valores médios e variância não bastam no caso de sistemas de tempo real. Embora o valor máximo possa ser utilizado para garantir um comportamento de pior caso, o conhecimento da distribuição de probabilidade pode abrir caminho para otimizações do sistema.

Existe ainda a questão do escalonamento do processador, que obviamente deve ser analisado no sentido de determinar se ele é apropriado ou não para a aplicação em questão. Embora a teoria de escalonamento tempo real tenha evoluído muito durante a década passada, apenas agora os sistemas operacionais começam a adaptar-se no sentido de suportar a teoria desenvolvida. Prova disto é o fato da maioria dos sistemas operacionais de tempo real disponíveis no mercado não responderem a maioria das perguntas listadas acima.

3.3.6 Abordagens de Escalonamento e o Sistema Operacional

A abordagem de escalonamento a ser utilizada é determinada pela natureza da aplicação: crítica ou não, carga estática ou não, etc. A questão fundamental para quem vai usar um SOTR é determinar sua capacidade de suportar a abordagem de escalonamento escolhida para o projeto.

A leitura do capítulo 2 deste livro deixa claro que escalonamento baseado em prioridades preemptivas é suficiente, desde que os atrasos e bloqueios associados com o SOTR sejam conhecidos.

Talvez o maior obstáculo à aplicação da teoria de escalonamento seja a dificuldade em determinar os tempos máximos de execução de cada tarefa, pois eles dependem de vários fatores como o fluxo de controle, a arquitetura do computador (*"cache"*, *"pipeline"*, etc), a velocidade de barramento e do processador, etc. Embora existam ferramentas experimentais nesta área, ainda não existem ferramentas com qualidade comercial que automaticamente informem o tempo máximo de execução de determinada tarefa em determinado computador.

Existem alguns caminhos para contornar este problema. Em tempo de projeto, quando o código ainda não existe, é possível estimar os tempos de execução das rotinas (métodos) dos diferentes módulos (objetos) e usar estas estimativas como dados de entrada nas equações do capítulo 2. Obviamente os resultados são, também eles, estimativas. Entretanto, como em tempo de projeto o código ainda não existe, é o melhor que pode ser feito. Esta estimativa possui uma grande utilidade. Ela permite detectar, ainda durante o projeto, problemas futuros com respeito aos tempos de resposta das tarefas, ou seja, detectar que não será possível atender aos requisitos temporais especificados com a arquitetura de hardware e software escolhida. A solução poderá ser substituir o processador por outro mais rápido, mudar a linguagem de programação por uma que gere código mais eficiente, alterar a especificação para aumentar períodos e *"deadlines"* ou ainda eliminar funcionalidades inicialmente previstas. Detectar a necessidade de alterações desta magnitude antes de iniciar a programação é certamente muito melhor do que fazer alterações deste tipo depois que toda a aplicação já estiver programada.

Uma vez que a aplicação esteja programada, é possível analisar se as estimativas usadas em tempo de projeto foram adequadas. Embora existam ferramentas que fazem isto automaticamente, são ainda projetos acadêmicos, sem a qualidade necessária para utilização em projetos comerciais. Uma alternativa é medir os tempos de execução. Neste caso, o tempo de execução de cada tarefa é medido um grande número de vezes, através de testes em condições controladas. Os resultados tem tipicamente o formato das figuras 3.5 e 3.6. Embora não exista a garantia de que o pior caso tenha sido observado nas medições, elas fornecem uma boa idéia para o tempo de execução da tarefa. Dependendo do tipo de aplicação, uma margem de segurança pode ser associada aos tempos medidos, isto é, será considerado como tempo máximo de execução o maior tempo de execução observado multiplicado por um fator de segurança. A partir da

estimativa do tempo máximo de execução de cada tarefa, é possível aplicar as equações do capítulo 2 e analisar se a aplicação é ou não escalonável.

Outro grande obstáculo à aplicação da teoria de escalonamento é obter os atrasos e bloqueios associados com o SOTR, em função de chamadas de sistema, interrupções de hardware, acesso a periféricos, etc. Como destacado na seção 3.3, a análise de escalonabilidade requer o conhecimento de detalhes do SOTR que na maioria das vezes não são disponibilizados pelo fornecedor. Este quadro está mudando lentamente na medida que os fornecedores percebem a importância desta informação para os desenvolvedores de aplicação. Enquanto isto, o projetista de uma aplicação tempo real fica com três alternativas:

- Desenvolver um suporte proprietário, que ele então conhecerá em detalhe;
- Selecionar um suporte que forneça as informações necessárias;
- Selecionar um suporte que não fornece as informações necessárias para a análise de escalonabilidade e adotar uma abordagem de escalonamento do tipo melhor esforço.

Quando a aplicação é do tipo tempo real brando (*"soft real-time"*), a preocupação do projetista resume-se a escolher um sistema operacional com boas propriedades temporais. Muitas vezes a teoria de escalonamento sequer é empregada, no sentido que não é feita uma análise matemática de escalonabilidade. Características como escalonamento baseado em prioridades preemptivas, *"kernel"* que executa com interrupções habilitadas e chaveamento de contexto rápido melhoram o desempenho de qualquer sistema, independentemente da análise matemática. Entretanto, somente a aplicação dos testes de escalonabilidade descritos no capítulo 2 é capaz de garantir o atendimento de requisitos temporais do tipo *"hard"*.

3.4 Tipos de Suportes para Tempo Real

A diversidade de aplicações 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 *"kernel"* 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 é um *"kernel"* que implementa a funcionalidade Unix, mas foi projetado para fornecer uma boa resposta

temporal.

Obviamente a solução de escalonamento oferecida em cada suporte também varia, e depende do mercado alvo. Todas as abordagens apresentadas no capítulo 2 encontram utilidade em algum cenário de aplicação e acabam sendo incorporadas em algum tipo de suporte. As soluções mais populares são o executivo cíclico para NTR dedicados e escalonamento baseado em prioridades, mas sem garantias, para SOTR. Uma lista de suportes para tempo real com cerca de 100 referências pode ser encontrada no anexo 2.

A figura 3.7 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.

	Funcionalidade mínima	Funcionalidade completa
Previsibilidade maior	Núcleo de Tempo Real	Futuro...
Previsibilidade menor	Qualquer Núcleo Simples	Sistema Operacional Adaptado

Figura 3.7 - Tipos de suportes para aplicações de tempo real.

3.4.1 Suporte na Linguagem

É importante observar que muitas vezes o suporte de tempo real existe, mas fica escondido do programador da aplicação. Duas situações são freqüentes, especialmente com NTR. É possível esconder o suporte dentro da própria linguagem (como em ADA ou Java) ou do ambiente de programação. Neste caso, os serviços normalmente associados com o sistema operacional não são obtidos através de chamadas de sistemas, mas sim através de construções da própria linguagem de programação. Por exemplo, não existe alocação e liberação de memória explícita em Java. Muitos ambientes integrados de desenvolvimento (IDE – *"Integrated Development Environment"*) incluem extensas bibliotecas e pacotes que também são usados pelo programador no lugar das chamadas de sistema.

Outra possibilidade aparece em propostas onde não existe a figura de um sistema operacional separado da aplicação, mas sim um conjunto de módulos que implementam serviços típicos de sistemas operacionais e são ligados junto com o código da aplicação. Neste caso, o projetista da aplicação identifica os serviços típicos de sistemas operacionais que serão necessários, seleciona os módulos que implementam tais serviços e os liga, através de um ligador (*"link-editor"*), aos módulos da aplicação, gerando um único código executável. De qualquer forma, o suporte para tempo real estará presente, embora "disfarçado".

3.4.2 *"Microkernel"*

Em função da dificuldade de prover todos os serviços com bom comportamento temporal, sistemas podem ser organizados em camadas. Na medida em que subimos na estrutura de camadas, os serviços tornam-se mais sofisticados e o comportamento temporal menos previsível. A figura 3.8 ilustra um sistema com apenas 3 camadas além da aplicação. Além do hardware existe um *"microkernel"* que oferece serviços básicos tais como alocação e liberação de memória física, instalação de novos tratadores de dispositivos e mecanismo para sincronização de tarefas. O *"kernel"* do sistema oferece serviços tais como sistema de arquivos e protocolos de comunicação. Assim, a aplicação tem a sua disposição uma gama completa de serviços. Entretanto, quando os requisitos temporais da aplicação exigem um comportamento melhor, ela pode acessar diretamente o *"microkernel"* e até mesmo o hardware. Obviamente este tipo de solução é mais apropriado para sistemas onde apenas uma aplicação é executada, ou todas as aplicações estão associadas com o mesmo usuário. Ao permitir que a aplicação acesse diretamente as camadas inferiores do sistema e até mesmo o hardware, o sistema operacional abre mão do completo controle sobre o que a aplicação pode e não pode fazer.

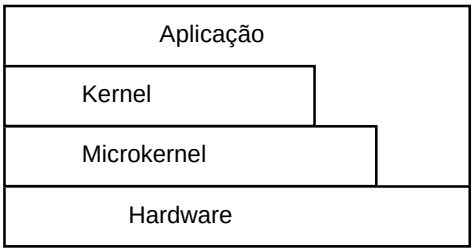

Figura 3.8 – Estratificação de serviços em um sistema.

3.4.3 Escolha de um Suporte de Tempo Real

Na verdade é muito difícil comparar diferentes SOTR com a finalidade de escolher

aquele mais apropriado para um dado projeto. A escolha de um SOTR não é trabalho simples. Por exemplo, este capítulo 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 necessárias para uma análise detalhada. Por exemplo, detalhes internos sobre o *"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 por cada SOTR varia.
- O conjunto de plataformas de hardware suportados varia em função do SOTR.
- Cada SOTR possui um esquema próprio para a incorporação de novos tratadores de dispositivos (*"device-drivers"*) e o esforço necessário para desenvolver novos tratadores varia de sistema para sistema.
- 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 vendida).

3.5 Exemplos de Suportes para Tempo Real

Esta seção descreve várias soluções disponíveis no mercado ou na Internet para o suporte a aplicações de tempo real. Como citado antes, existem dezenas senão centenas de soluções, nos mais variados níveis de preço, robustez, funcionalidade e previsibilidade temporal. O conjunto de soluções escolhido para ser apresentado aqui procura ilustrar os tópicos apresentados nas seções anteriores, sem ter a pretenção de ser um levantamento preciso do mercado. Inicialmente o padrão Posix é decrito com maior profundidade, por ser um padrão que procura definir características de portabilidade para aplicações, sem dependências de fornecedores de sistemas operacionais, e por ser seguido em maior ou menor grau pela maioria dos sistemas operacionais (Solaris, QNX, etc). Em seguida é discutido o escalonamento nos sistemas

Unix SVR4 e Solaris, para mostrar a evolução do mesmo em direção a um melhor comportamento temporal. Os sistemas ChorusOS e QNX são apresentados como soluções comerciais específicas para tempo real. Finalmente, o sistema operacional RT-Linux é descrito, por ser uma alternativa viável dentro do contexto do software livre. O anexo C deste livro contém uma lista com cerca de 100 sistemas operacionais de tempo real, incluindo o respectivo endereço na Internet.

3.5.1 Posix para Tempo Real

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 Programming Interface"*). Na verdade a quase totalidade 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. Como Posix na verdade procurou padronizar o Unix, as abstrações usadas já eram conhecidas da maioria dos programadores.

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. Posix herdou do Unix o termo "processo", com o mesmo significado do termo "tarefa" empregado neste texto. Nesta seção vamos continuar usando "tarefa" para manter a coerência com o resto do capítulo, mesmo nos momentos que a literatura Unix usaria a palavra "processo".

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.

Usando um SOTR que é compatível com Posix, a aplicação fica amarrada aos conceitos e às primitivas do Posix. Além disto, o porte da aplicação entre sistemas diferentes, mas compatíveis com o Posix, fica muito facilitado. 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.

A documentação do Posix aparece dividida em diversos documentos. Por exemplo, o padrão 1003.1 descreve a funcionalidade básica de um sistema operacional Unix e

inclui chamadas de sistema para gerência de tarefas, dispositivos, sistemas de arquivo e comunicação entre tarefas básica (IPC – *"inter-process communication"*). O padrão 1003.1b estende o 1003.1 para tempo real, incluindo semáforos, escalonamento com prioridades, temporizadores, primitivas para IPC melhoradas e arquivos para tempo real. O padrão 1003.1c permite múltiplas *"threads"* no mesmo espaço de endereçamento. O padrão 1003.1d estende ainda mais o 1003.1b para tempo real, incluindo facilidades para instalar tratadores de interrupção. Como este livro trata de sistemas de tempo real, qualquer referência ao "padrão Posix" inclui automaticamente o básico e também todas as extensões definidas para tempo real.

O padrão 1003.13 define perfis (*"profiles"*), isto é, subconjuntos de facilidades que aparecem nos diversos padrões e juntas formam uma solução apropriada para uma determinada classe de aplicações. Uma vez que o padrão Posix é extenso, implementar todos os recursos previstos implica em muito software. A idéia é usar Posix mesmo em pequenos sistemas embutidos em equipamentos industriais e domésticos. Muitas empresas irão preferir desenvolver implementações parciais, e os perfis ajudam a diminuir as variações possíveis dentro do universo de soluções Posix. Para tempo real foram definidos vários perfis:

- Pequeno sistema controlando um ou mais dispositivos, nenhuma interação com operador, não tem sistema de arquivos, apenas uma tarefa com múltiplas *"threads"*;
- Inclui sistema de arquivos e entrada e saída assíncrona;
- Inclui suporte para MMU (*"Memory Management Unit"*), várias tarefas com múltiplas *"threads"*, mas sem sistema de arquivos;
- Capaz de executar uma mistura de tarefas tempo real com tarefas convencionais, inclui MMU, dispositivos de armazenamento (discos, fitas, etc), suporte para rede, etc.

Um aspecto central para as aplicações de tempo real é o escalonamento. Posix suporta escalonamento baseado em prioridades, as quais podem ser definidas em tempo de execução. No mínimo 32 níveis de prioridades devem ser suportados, embora o número exato seja definido por cada implementação. Em conjunto com o esquema de prioridades existem políticas de escalonamento que definem como são escalonadas *"threads"* com a mesma prioridade. Isto pode ser feito via FIFO (*"First-In First-Out"*), fatias de tempo ou outra política qualquer definida pela implementação. Desta forma o programador sabe que pode contar com um escalonamento básico em todos os SOTR tipo Posix, ao mesmo tempo que permite inovações neste campo.

A princípio as *"threads"* são criadas para competir com todas as outras *"threads"* do sistema pelo tempo do processador. Entretanto, a competição também pode acontecer por tarefa, sendo então necessário definir um critério para dividir o tempo do processador entre tarefas. Uma implementação Posix em particular pode suportar apenas uma destas opções ou ambas.

Posix inclui as primitivas clássicas *"fork"* para criação de uma tarefa filha e *"wait"* para esperar pelo término de outra tarefa. Além disto, cada tarefa pode conter várias *"threads"*. As *"threads"* de uma mesma tarefa compartilham o seu espaço de endereçamento mas possuem alguns atributos particulares, como o tamanho da sua pilha individual.

Com respeito a mecanismos de sincronização, o padrão oferece uma coleção deles. Posix inclui semáforos que podem ser usados para sincronizar *"threads"* de diferentes tarefas. Embora semáforos possam ser usados para sincronizar *"threads"* da mesma tarefa, o custo envolvido é alto e existem alternativas mais eficientes. Além das tradicionais operações P e V, existe uma operação P não bloqueante e uma primitiva para determinar o valor atual do semáforo.

Quando o objetivo da sincronização é garantir o acesso exclusivo a uma seção crítica, Posix oferece a construção *"mutex"*. Uma variável tipo *"mutex"* suporta as operações *"lock"* e *"unlock"* e sua implementação é mais eficiente do que semáforos. Variáveis *"mutex"* também suportam os protocolos de herança de prioridade e uma variação do *"prioriry ceiling"*.

A sincronização baseada em condições pode ser construída através de variáveis condição. Uma *"thread"* fica bloqueada junto a uma variável condição através das operações *"wait"* ou *"timedwait"*. Ela é acordada quando outra *"thread"* executar uma operação *"signal"* (acorda uma *"thread"*) ou *"broadcast"* (acorda todas as *"threads"*) sobre a variável condição em questão.

Cada variável condição é associada com uma variável *"mutex"* quando é criada. Para ficar bloqueado em uma variável condição uma *"thread"* deve ter antes executado um *"lock"* sobre o *"mutex"* associado. No momento que a *"thread"* ficar bloqueada na variável condição o *"mutex"* é automaticamente liberado. Da mesma forma, quando uma *"thread"* é acordada da variável condição, é automaticamente solicitado um *"lock"* sobre o *"mutex"* associado. A *"thread"* somente retoma sua execução após ter obtido novamente o *"lock"* sobre o *"mutex"*. Embora o Posix não especifique qual *"thread"* ganha o *"mutex"* quando uma *"thread"* acorda outra, o escalonamento baseado em prioridades resolve a questão executando a *"thread"* com maior prioridade. Esta operação conjugada de *"mutex"* e variável condição permite facilmente a implementação de construções do tipo monitores, como descrito em [BuW97]. Isto pode ser feito através de disciplina de programação e chamadas diretas ao sistema operacional ou então pode ser usado por um compilador para implementar monitores a nível da linguagem de programação.

Posix também define um mecanismo chamado fila de mensagens, o qual é semelhante a caixas postais, pois a comunicação é assíncrona e o endereçamento indireto (endereço de fila e não de tarefa). Cada fila de mensagens pode ter vários leitores e vários escritores. Mensagens podem ter prioridades, usada então para ordenar as mensagens na fila. Filas de mensagens podem ser usadas para a comunicação entre *"threads"* da mesma tarefa mas, em função do custo associado, faz mais sentido usa-las para a comunicação entre *"threads"* de tarefas diferentes. *"Mutexes"* e variáveis

compartilhadas é o mecanismo mais apropriado para *"threads"* da mesma tarefa.

Uma fila de mensagem recebe um nome ao ser criada. Para ser acessada ela deve ser aberta, de maneira semelhante a um arquivo. As operações são também semelhantes às operações utilizadas para acessar arquivos. Uma mensagem é enviada através da operação *"send"*. Ela é recebida através da operação *"receive"*, a qual pode ser bloqueante ou não bloqueante. Quando várias *"threads"* estão esperando para retirar uma mensagem de uma fila vazia, o escalonamento baseado em prioridades define qual será atendida antes. O mesmo pode acontecer com várias *"threads"* esperando para colocar uma mensagem na fila, pois a mesma pode ter uma capacidade máxima definida no momento de sua criação.

Posix suporta a existência de vários relógios simultaneamente, cada um possui um identificador próprio. O padrão exige que cada implementação ofereça no mínimo um relógio, chamado CLOCK_REALTIME, com resolução mínima de 20 ns. Esta resolução diz respeito aos parâmetros mas não garante que a passagem do tempo no sistema seja medida em termos de intervalos de 20 ns cada. Uma espera relativa é obtida através da chamada *sleep* (múltiplo de segundo) ou da chamada *"nanosleep"* (resolução maior). A *"thread"* será bloqueada no mínimo pelo tempo solicitado.

Um aspecto importante do Posix, como de resto de qualquer sistema operacional tipo Unix, são os sinais. Durante a execução de uma tarefa erros do tipo "acesso ilegal a memória" (SIGSEGV), "instrução ilegal" (SIGILL) e "divisão por zero" (SIGFPE) são detectados pelo *"kernel"* que gera um sinal para informar a aplicação. O programador da aplicação pode tratar este tipo de erro associando uma rotina da aplicação que será executada sempre que este sinal for gerado. Após a execução do tratador do sinal, a tarefa é retomada a partir do ponto onde foi interrompida. O mecanismo de sinais é importante para aplicações de tempo real pois permite a implementação de técnicas de tolerância a faltas. Entretanto, por ser um mecanismo assíncrono (se e quando vai ocorrer é desconhecido do programador), ele dificulta a análise de escalonabilidade do sistema.

Uma utilização clássica de sinais no contexto de tempo real aparece associada com temporizadores. Uma tarefa pode solicitar uma determinada temporização ao *"kernel"* ou a uma biblioteca. A tarefa é avisada que a temporização terminou pelo sinal SIGALRM. Sinais também podem ser utilizados para sinalizar a chegada de uma mensagem em uma fila até então vazia, quando não existem *"threads"* bloqueadas esperando.

Acontecimentos importantes na vida da aplicação também podem ser indicados por sinais. Por exemplo, a perda de um *"deadline"*, uma falha no hardware, uma mudança no modo de operação (*"mode change"*) podem ser programados na forma da emissão de um ou mais sinais por parte da *"thread"* que detecta o evento. As demais tarefas da aplicação recebem o sinal e mudam o seu comportamento em função do evento que ele sinaliza. Este estilo de programação é complexo e sujeito a "*bugs*". Efeito semelhante pode ser obtido através de uma *"thread"* que espera pela ocorrência do evento e então aborta ou suspende as *"threads"* envolvidas e toma as providências necessárias para que

a aplicação responda de forma adequada ao evento sinalizado. Embora as duas formas de programação sejam equivalentes em poder de expressão, o uso de *"threads"* e mecanismos de sincronização explícitos gera um código mais legível do que sinais assíncronos.

Para aplicações convencionais o Posix define os sinais SIGUSR1 e SIGUSR2, os quais não carregam qualquer informação adicional além do próprio sinal. Para tempo real é definido um conjunto de sinais entre SIGRTMIN e SIGRTMAX, com no mínimo 8 sinais. Estes sinais podem carregar dados adicionais além da sinalização em si e são enfileirados, ao passo que os outros tipos de sinais não são enfileirados mas sobrescritos. Além disto, caso vários sinais de tempo real estejam enfileirados para uma dada tarefa, aquele com menor valor é sempre entregue antes.

Muitos sinais são gerados pelo *"kernel"*, por tratar-se de uma situação de erro ou por estarem associados com uma ação de teclado (Control-C normalmente gera SIGINT). O código da aplicação pode gerar sinais através da chamada de sistema *"kill"* (sinais convencionais) e da chamada de sistema *"sigqueue"* (sinais de tempo real). São necessárias chamadas diferentes em função dos parâmetros adicionais passados com os sinais de tempo real.

O conceito de sinais foi introduzido já nas primeiras versões do Unix, por volta de 1970, quando o conceito de *"threads"* não era comum e estas não eram suportadas pelo Unix. O conceito de sinais é coerente com um fluxo de execução por tarefa, mas foi necessário adapta-lo para o contexto de múltiplas *"threads"*. Alguns sinais são enviados para uma *"thread"* específica, outros são enviados para uma *"thread"* qualquer da tarefa, novas primitivas foram criadas especialmente para lidar com a existência de *"threads"*. Uma análise detalhada do serviço de sinais foge do escopo deste livro. Na verdade sinais foram utilizados originalmente, em grande parte, para contornar a falta de *"threads"*. Com a disponibilidade de múltiplas *"threads"* por tarefa a utilidade dos sinais diminuiu muito. Entretanto, existe uma enorme quantidade de código para Unix que utiliza sinais (sem falar nos próprios programadores). Esta situação deverá perdurar por muito tempo.

Uma descrição completa do Posix é capaz de ocupar um livro inteiro, como em [Gal95]. Pelo resumo apresentado nesta seção é possível perceber que, em termos funcionais, o Posix cobre as necessidades expostas no início do capítulo. Na verdade ele oferece muito mais do que o necessário para sistemas tempo real pequenos. O mecanismo de perfis (*"profiles"*) permite a um implementador de SOTR fornecer apenas as facilidades Posix que são importantes para o mercado pretendido. Mesmo existindo esta flexibilidade sobre quais facilidades estarão presentes em uma dada implementação Posix, as facilidades presentes devem implementar as abstrações e apresentar as interfaces previstas no padrão. Desta forma, programadores sempre encontrarão um ambiente de programação familiar. Cabe apenas ao projetista escolher a implementação Posix com as facilidades apropriadas.

É importante notar que, como o padrão é extenso, a compatibilidade com Posix quase sempre é completa com alguns perfis e parcial com outros. Muitas vezes o

fornecedor de um SOTR anuncia vagamente que "segue o Posix", sem especificar o perfil em questão ou se é algo parcial e não corresponde exatamente a um perfil padronizado. Parece ser algo bom para o marketing do SOTR mas obriga o projetista da aplicação fazer um estudo detalhado do sistema operacional. Existe um processo formal de homologação (ver http://www.computer.org), mas a maioria dos SOTR no mercado não passaram por ele.

Muitos SOTR suportam duas interfaces, uma Posix e uma proprietária. Isto acontece porque a interface proprietária já existia e sobre ela foi implementada uma biblioteca com interface Posix, ou porque na interface proprietária podem ser feitas otimizações que resultam em melhor desempenho.

É importante observar que o padrão Posix preocupa-se com as interfaces e a funcionalidade. Com respeito aos aspectos temporais a questão é mais complexa. Na verdade o padrão não especifica (propositadamente) como o *"kernel"* deve ser implementado, qual deve ser o tempo de execução de cada chamada de sistema, qual deve ser o tempo de processador gasto para executar funções do sistema. Sendo assim, estes elementos dependem totalmente da implementação Posix sendo usada. Pode-se contar com aquilo que está no padrão, como escalonamento baseado em prioridades e variáveis *"mutex"* suportanto uma variação do *"priority ceiling protocol"*. Entretanto, como discutido antes neste capítulo, uma boa lista de perguntas sobre o comportamento temporal ainda fica para ser respondida por quem implementa o sistema operacional.

3.5.2 Escalonamento no Unix SVR4

Embora o Unix SVR4 não implemente exatamente a solução de escalonamento do Posix, elas são semelhantes. Seja como for, a solução do SVR4 é muito melhor do que aquela apresentada antes como a do Unix tradicional. Uma descrição detalhada deste sistema operacional, assim como de várias outras versões do Unix, pode ser encontrada em [Vah96]. Novamente vamos utilizar a palavra "tarefa" para denotar o conceito associado com o termo "processo" no mundo Unix.

No SVR4 classes de escalonamento definem a política de escalonamento para as tarefas. Como *"default"*, SVR4 fornece duas classes: *"time-sharing"* e tempo real. A tarefa com prioridade maior sempre executa, com exceção de partes não interrompíveis do *"kernel"*. Existem 160 níveis de prioridades, onde um número maior representa uma prioridade mais elevada. Existe uma divisão prévia entre as classes de escalonamento: 0 a 59 para a classe *time-sharing*, 60 a 99 são prioridades usadas pelo código do sistema e os números entre 100 e 159 são utilizados pelas tarefas da classe de tempo real.

Uma tarefa executando código do *"kernel"* pode ser preemptada apenas nos pontos de interrupção (*"preemption points"*). A atribuição e atualização das prioridades de cada tarefa é feita pela classe em questão. Novas classes de escalonamento podem ser escritas e instaladas no sistema. O código de uma classe de escalonamento é organizado de forma semelhante a um tratador de dispositivo (*"device-driver"*), pois deve fornecer

código para suportar uma interface padrão definida pelo SVR4, cujas rotinas serão chamadas sempre que uma decisão de escalonamento envolvendo tarefas daquela classe forem necessárias. Desta forma, além das duas classes providas sempre, o projetista é livre para criar sua própria solução de escalonamento. Obviamente será muito mais fácil portar a aplicação para outras instalações se apenas as classes de escalonamento básicas forem utilizadas.

A classe *"time-sharing"* é a classe *"default"* das tarefas. Tarefas possuem prioridades variáveis, definidas em tempo de execução. Fatias de tempo são usadas para dividir o tempo do processador entre tarefas de mesma prioridade. A duração da fatia de tempo depende da prioridade da tarefa (menor prioridade, maior fatia). Em vez de recalcular a prioridade de cada tarefa a cada segundo (como no Unix SVR3), o SVR4 recalcula a prioridade de uma tarefa somente na ocorrência de eventos associados com a tarefa. Desta forma o custo computacional deste recálculo é drasticamente reduzido.

A prioridade é reduzida sempre que a fatia de tempo é esgotada pela tarefa. A prioridade é elevada sempre que a tarefa fica bloqueada por alguma razão ou fica muito tempo sem esgotar sua fatia de tempo (provavelmente porque outras tarefas de maior prioridade estão sempre tomando o processador). O recálculo é rápido, pois o evento em geral afeta uma única tarefa de cada vez.

O descritor de uma tarefa contém, entre outros campos:

- ts_timeleft - tempo restante na fatia;
- ts_cpupri - parte da prioridade definida pelo sistema;
- ts_upri - parte da prioridade definida pelo usuário (nice, entre -20 e +19);
- ts_umdpri - prioridade em modo usuário, ts_cpupri + ts_upri, valor máximo de 59;
- ts_dispwait - número de segundos desde o início da fatia.

Quando uma tarefa acorda de um bloqueio dentro do *"kernel"* a sua nova prioridade dentro do *"kernel"* é determinada pela condição de bloqueio da qual acorda. Ao voltar para modo usuário a prioridade é definida pelo campo *ts_umdpri*. Uma tabela de parâmetros determina como *ts_cpupri* é calculada. Ela contém uma entrada para cada nível de prioridade possível no sistema. Os campos desta tabela determinam o funcionamento de um mecanismo de envelhecimento (*"aging"*).

A classe tempo real utiliza prioridades entre 100 e 159, maiores que qualquer tarefa *time-sharing*. Quando uma tarefa tempo real é liberada, ela obtém o processador imediatamente no caso de estar executando uma tarefa *time-sharing* em modo usuário. No caso de uma tarefa *"time-sharing"* estar executando código do *"kernel"*, a tarefa tempo real será obrigada a esperar que a tarefa *"time-sharing"* execute até atingir o próximo ponto de interrupção do *"kernel"*. Esta espera pode ser relativamente grande e prejudica bastante o tempo de resposta das tarefas de tempo real no Unix SVR4.

Cada tarefa tempo real é caracterizada pela sua prioridade e pela sua fatia de tempo.

A tabela de parâmetros empregada por esta classe contém apenas a fatia de tempo *"default"* para cada nível de prioridade. Tipicamente são fatias maiores para prioridades menores, pois espera-se que as tarefas de prioridade maior sejam muito curtas. Entretanto, como fatias de tempo são usadas apenas para tarefas de mesma prioridade, este mecanismo pode ser completamente ignorado em análises de escalonabilidade.

Uma figura de mérito importante é a latência desde uma interrupção de hardware até o disparo da tarefa que trata este evento. Quando o evento ocorre ele é sinalizado por uma interrupção. Existe o processamento da interrupção, a qual libera a tarefa de tempo real. Se houver uma tarefa *"time-sharing"* executando código do *"kernel"*, será necessário esperar que ela chegue até o próximo ponto de interrupção. Neste momento ocorre o chaveamento de contexto e a tarefa tempo real inicia a sua execução. O código da aplicação é executado, respondendo ao evento. Observe que, para o tempo de resposta da tarefa tempo real, ainda seria necessário incluir as interferências e bloqueios causados por outras tarefas tempo real da própria aplicação.

Certamente a solução de escalonamento do SVR4 é melhor do que a solução Unix tradicional. Entretanto, ainda não é satisfatória para muitas aplicações de tempo real, pois em geral melhora o desempenho mas não resolve completamente o problema da previsibilidade. Além do *"kernel"* não ser interrompível em qualquer ponto, é difícil ajustar o sistema para um conjunto misto de aplicações. Por exemplo, experiências envolvendo um editor simples, uma tarefa em *"background"*, e uma sessão de vídeo usando X-server, mostraram que nenhuma atribuição de classes e prioridades resolve completamente o problema de compartilhamento do processador neste sistema operacional. Além disto, os tempos das chamadas de sistema e os tempos máximos de bloqueio dentro do *"kernel"* não são conhecidos.

3.5.3 Escalonamento no Solaris 2.x

O Solaris é uma variação do Unix fornecido pela Sun Microsystems. A solução de escalonamento do sistema operacional Solaris melhora o escalonamento do Unix SVR4 em vários aspectos. O *"kernel"* do Solaris é completamente preemptável, isto é, interrupções podem acontecer mesmo quando uma *"thread"* está executando código do *"kernel"*. Na verdade as interrupções são desabilitadas apenas muito rapidamente em alguns poucos pontos. Isto diminui a latência no atendimento das interrupções mas exige que as estruturas de dados do *"kernel"* sejam protegidas por mecanismos de sincronização. Interrupções na verdade ativam *"threads"* especiais do *"kernel"*, que podem ficar bloqueadas se necessário. Isto acontece quando a *"thread"* especial necessita acessar uma estrutura de dados em uso por uma *"thread"* normal. As *"threads"* associadas com interrupções possuem a prioridade mais alta no sistema.

Da mesma forma que o Unix SVR4, são empregadas classes de escalonamento. Além das classes fornecidas, novas classes podem ser carregadas dinamicamente. O escalonador suporta multiprocessamento, usando uma fila única de *"threads"* para todos

os processadores. A exceção são as *"threads"* de interrupção, associadas necessariamente com o processador onde a interrupção aconteceu. Isto é necessário pois o tratamento da interrupção de hardware pode exigir o acesso a um controlador de dispositivo conectado com aquele processador em particular.

Embora o modelo básico seja elegante, existe também o que poderia ser chamado de "escalonamento escondido". Por exemplo, quando uma tarefa vai voltar para modo usuário depois de executar código do *"kernel"*, é verificado se existe alguma pendência nos módulos do *"kernel"* que implementam *"streams"* (*"streams"* são tipicamente associadas com protocolos de comunicação). Caso exista, esta tarefa executa a pendência, "prestando um favor" para as tarefas que estão usando os serviços dos *"streams"*. Se estas tarefas possuírem prioridade menor do que aquela deixando o *"kernel"*, temos uma situação de inversão de prioridades. Este problema não é tão grande pois o Solaris utiliza nas *"threads"* do *"kernel"* uma prioridade menor do que as *"threads"* de tempo real. Mas se forem as *"threads"* de tempo real que estão usando os *"streams"*, então elas serão prejudicadas.

Com respeito a inversão de prioridades devido ao compartilhamento de estruturas de dados, o *"kernel"* do Solaris suporta parcialmente herança de prioridades. Ocorre que dentro do *"kernel"* são usados quatro tipos de mecanismos de sincronização. *"Mutexes"* suportam corretamente herança de prioridades, mas no caso dos semáforos e das variáveis condição o dono do *"lock"* (*"thread"* que bloqueou o recurso) não é conhecido e a herança de prioridade não pode ser usada. Existe ainda um mecanismo do tipo "bloqueio leitor/escritor" onde a herança de prioridades funciona apenas para os escritores e o primeiro leitor.

Uma análise completa do *"kernel"* Solaris foge do escopo deste livro (ver [Vah96]). Entretanto, os problemas apresentados nos parágrafos anteriores ilustra o nível de complexidade presente na análise temporal do *"kernel"* de um sistema operacional completo como o Solaris. Não é sem razão que sistemas operacionais com funcionalidade completa podem até ser rápidos, mas não apresentam um comportamento determinista.

3.5.4 ChorusOS

O sistema operacional ChorusOS (http://www.sun.com/chorusos/), fornecido pela Sun Microsystems, é a base para tempo real da solução de software proposta pela Sun para o setor de telecomunicações. O ChorusOS pode ser considerado como o sistema operacional de tempo real que complementa o sistema operacional Solaris. Com estes dois sistemas operacionais a Sun procura prover uma solução completa para o setor de telecomunicações, no que se refere a sistemas operacionais.

O mercado visado pelo sistema operacional ChorusOS é principalmente o dos equipamentos de telecomunicações. O ChorusOS é usado em centrais públicas e privadas de telefonia, assim como em sistemas de comunicação de dados, *"switches"*,

sistemas de mensagens faladas, estações de telefonia celular, *"web-phones"*, telefones celulares e sistemas de transmissão via satélite.

Além do sistema operacional propriamente dito, a Sun comercializa o "Sun Embedded Workshop", um ambiente integrado de desenvolvimento (IDE – *"Integrated Development Environment"*) que inclui as ferramentas e todos os componentes necessários para construir instâncias executáveis do ChorusOS. Esta IDE é sempre adaptada para uma determinada plataforma alvo e uma plataforma de desenvolvimento. Presentemente as seguintes plataformas alvo são suportadas:

- ix86 (desenvolvimento no Solaris ou no Windows NT);
- UltraSPARC™ IIi (desenvolvimento no Solaris);
- PPC603, 604, 750, 821, 823, 860, 8260 (desenvolvimento no Solaris);
- PPC603, 604, 750, 860 (desenvolvimento no Windows NT).

O ChorusOS também é o sistema operacional subjacente ao "JavaOS for Consumers". O "JavaOS for Consumers" aproveita a capacidade do *"microkernel"* do ChorusOS em suportar várias plataformas alvo e prover um ambiente de tempo real para o desenvolvimento de aplicações em Java visando a eletrônica de consumo.

O ChorusOS emprega uma arquitetura baseada em componentes que permite a inclusão de diferentes serviços no executável do sistema operacional. Isto permite um ajuste fino do *"kernel"*, de acordo com o plataforma alvo, a memória disponível e as necessidades da aplicação. O *"kernel"* pode iniciar com apenas 10 Kbytes e crescer a medida que componentes são acrescentados. Múltiplas personalidades e APIs podem executar simultaneamente sobre a mesma plataforma de hardware. Isto facilita a integração de aplicações já existentes em novos sistemas que executam sobre o ChorusOS. As seguintes APIs são suportadas:

- API nativa do CHORUS;
- RT-POSIX (1003.1b/lc);
- Java runtime and services;
- Sistemas operacionais legados.

O ChorusOS inclui um *"microkernel"* de 10 Kbytes, que suporta uma única aplicação composta por várias *"threads"*. A gerência de interrupções de hardware e de software deve ser provida pela aplicação. Para aplicações maiores pode ser usado um *"kernel"* que suporta múltiplos usuários independentes além de programas de sistema. Aplicações podem executar no espaço de endereçamento do sistema ou de usuário. Existe um escalonador de tempo real com prioridades preemptivas e FIFO para *"threads"* de mesma prioridade. Alternativamente, pode ser usado um escalonador que suporta várias classes de tarefas, incluindo "prioridade preemptiva e FIFO", "prioridade com fatias de tempo", "estilo Unix" ou então uma política definida pela aplicação.

A gerência de memória inclui 4 possibilidades: Gerência da memória física, sem

proteção; Múltiplos espaços de endereçamento protegidos; Espaços de endereçamento paginados protegidos; Memória virtual com paginação sob demanda.

A comunicação entre tarefas inclui troca de mensagens com transparência de distribuição, caixas postais e filas de mensagens tempo real. Para sincronização existem semáforos, *"mutexes"*, *"mutexes"* de tempo real (reduzem a inversão de prioridade) e sinalização de eventos.

Existe amplo suporte para serviços de tempo e gerência de interrupções. O mesmo acontece com sistemas de arquivos, os quais incluem: UFS e FFS do Unix com apontadores de 64 bits; *"Flash File System"* baseado no sistema de arquivos do MS-DOS mas com nomes longos; compartilhamento de arquivos através do NFS (cliente e servidor); etc.

Com respeito a redes de comunicação, o ChorusOS oferece uma pilha TCP/UDP/IP completa, incluindo suporte para múltiplas interfaces de rede, roteamento entre múltiplas interfaces, IP *"forwarding"*, IP *"multicast"*. Suporta *"sockets"*. Inclui SLIP e PPP, DHCP, servidores de nomes, rsh, Xclients (Xlib, Xt, Xmu, Xext, Xaw) além do Sun RPC. A nível de hardware suporta ethernet, linha serial, VME-backplane e cPCI.

Como pode ser visto, o ChorusOS é um sistema operacional flexível com respeito ao seu tamanho. Todas as características listadas antes (sistemas de arquivos, protocolos de rede, etc) são configuráveis. Assim, o projetista da aplicação inclui na sua versão do ChorusOS apenas as facilidades realmente necessárias. Com respeito ao comportamento temporal, a documentação da Sun é bastante restrita, situação na verdade típica. O dados disponíveis informam apenas o tempo típico do chaveamento de tarefas e a máxima latência até o disparo de um tratador de interrupção. Um SOTR com amplos recursos pode tornar-se um problema, pois quanto mais funcionalidade do sistema é utilizada, mais difícil fica prever o seu comportamento temporal.

3.5.5 Neutrino e QNX

O sistema operacional QNX consiste de um *"microkernel"* (http://www.qnx.com/products/os/qnxrtos.html) e uma coleção de módulos opcionais para os serviços como sistemas de arquivos, redes, interfaces gráficas de usuário, etc. O *"microkernel"* é responsável por criar tarefas, gerênciar a memória e controlar temporizadores. O *"microkernel"* também inclui API POSIX.1 certificada e muitos serviços tempo real do POSIX.1b. Esta divisão em *"microkernel"* e módulos permite ao QNX ser pequeno o bastante para ser colocado em ROM, mas ser capaz de crescer através da adição de módulos até transformar-se em um sistema operacional distribuído com funcionalidade completa. O software de rede do QNX é chamado FLEET e cria um conjunto homogêneo de recursos que podem ser acessados de forma transparente.

O QNX inclui relógios e temporizadores estilo Posix, interrupções aninhadas, instalação e desinstalação dinâmica de tratadores de interrupções e compartilhamento

de memória. O QNX suporta 32 níveis de prioridades preemptivas e oferece a escolha do algoritmo de escalonamento para tarefas de mesma prioridade. Uma característica interessante, servidores podem ter a sua prioridade definida pelas mensagens que eles recebem dos clientes.

Diversos sistemas de arquivos podem ser executados simultaneamente, entre eles: Sistema de arquivos Posix, com semântica POSIX.1 e Unix completa; Sistema de arquivos embutido (*"embedded"*), em diversas versões; Protocolo SMB (*"Server Message Block"*) para compartilhamento de arquivos; NFS; Sistema de arquivos DOS; Sistema de arquivos para CD-ROM; etc.

Algumas métricas são apresentadas na documentação do QNX. Estes valores foram medidos em um processador Pentium/133 com um Adaptec 2940 Wide SCSI controller, um Barracuda SCSI-Wide disk drive, um 100 Mbit PCI-bus Digital 21040 Ethernet card, e um 10 Mbit ISA-bus NE2000 Ethernet card:

- Chaveamento de contexto: 1.95 µsec (completo, a nível de usuário);
- Latência de interrupção: 4.3 µsec;
- Latência de escalonamento:7.8 µsec;
- Disk I/O (baseado em registros de 16384 bytes): 4 Mbytes/s (leitura) e 5,3 Mbytes/s (escrita);
- Network throughput: 1.1 Mbytes/s (10 Mbit Ethernet) e 7.5 Mbytes/s (100 Mbit Ethernet).

Abaixo é apresentada a latência de interrupções e tarefas. Todos os tempos são dados em microsegundos:

Processor	Chaveamento de contexto	Latência de interrupção
Pentium/133	1.95	4.3
Pentium/100	2.6	4.4
486DX4	6.75	7
386/33	22.6	15

Com interrupções aninhadas, estas latências de interrupção representam a latência no pior caso para a interrupção de maior prioridade. A prioridade de uma interrupção pode ser definida pela aplicação e a latência de interrupções associadas com prioridades menores é definida em parte pelos tempos de execução dos tratadores de interrupção específicos da aplicação.

O QNX suporta diversos processadores, tais como AMD ÉlanSC300/310/400/410, Am386 DE/SE, Cyrix MediaGX, Intel386 EX, Intel486, ULP Intel486, Pentium (com/sem MMX), Pentium Pro, Pentium II, STPC da STMicroelectonics, e todos os processadores genéricos baseados em x86 (386 e depois). Além disto, existe suporte para uma enorme variedade de barramentos e periféricos.

Outro sistema operacional de tempo real comercializado pela mesma empresa é o QNX Neutrino (http://www.qnx.com/products/os/neutrino.html). O *"microkernel"* do Neutrino fornece serviços de tempo real essenciais para aplicações embutidas (*"embedded"*), incluindo troca de mensagens, serviços de *"threads"* do Posix, *"mutexes"*, variáveis condição, semáforos, sinais e escalonamento. Ele pode ser estendido para suportar as filas de mensagens do Posix, sistemas de arquivos, redes de computadores, e outras facilidades a nível de sistema operacional através de módulos de serviço que são plugados ao *"microkernel"*.

A arquitetura do Neutrino é baseada em troca de mensagens e forma um barramento de software que permite a aplicação plugar e desplugar módulos do sistema operacional sem necessidade de reinicializar o sistema. O resultado é um sistema operacional bastante flexível.

Por exemplo, é possível ligar (*"to link"*) o código da aplicação diretamente com o *"microkernel"* para criar uma imagem de memória única, com múltiplas *"threads"*, para sistemas embutidos pequenos (solução muitas vezes empregada também por executivos de tempo real mais simples). Alternativamente, é possível executar o módulo "gerente de tarefas" e obter todos os serviços de um modelo tradicional de tarefas, como proteção de memória e múltiplas aplicações, com APIs baseadas no Posix. Entre os módulos disponíveis existe:

- Sistema de janelas para plataformas com recursos limitados;
- Módulos de inicialização que são descartados após a sua execução;
- Gerência de tarefas, com *"threads"*, proteção de memória, etc;
- Diversos sistemas de arquivos, incluindo o *"Embeddable QNX Filesystem Manager"* compatível com Posix 1003.1, sistemas de arquivos para memória Flash e *"CD-ROM Filesystem Manager"*;
- Protocolos TCP/IP, incluindo ftp, ftpd, telnet, telnetd e outros, além de PPP e rede local;
- Gerência de linhas seriais.

A lista de processadores suportados pelo Neutrino inclui: x86 - 386, i386 EX, Am386SE/DE, AMD ÉlanSC400/410, 486, Cyrix MediaGX, Pentium, Pentium Pro, Pentium II, PowerPC - 401, 403, 603e, 604e, 750, MPC860, MPC821, MPC823, MIPS - R4000, R5000, NEC VR4300/4102/4111, VR5000, IDT R4700, QED RM5260/5270/5261/5271.

Na verdade o QNX e o QNX Neutrino ocupam faixas sobrepostas do mercado de sistemas operacionais de tempo real. A princípio o Neutrino é voltado para aplicações menores, embutidas, ao passo que o QNX tradicional visa aplicações maiores, possivelmente distribuídas. Mas existe uma certa fatia do mercado que pode ser atendida tanto por um como pelo outro.

3.5.6 Linux para Tempo Real

Linux é um sistema operacional com 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 *"free software"* no sentido que pode ser copiado, modificado, usado de qualquer forma e redistribuído sem nenhuma restrição. Entretanto, ninguém usando Linux ou criando uma adaptação do Linux pode tornar o produto resultante proprietário. Desenvolvido sob o *"GNU General Public License"*, o código fonte do Linux está disponível de graça para todos.

O sistema operacional Linux é uma implementação independente do Posix e 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 livro. 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 convencional segue o estilo de um *"kernel"* Unix tradicional, não baseado em *"microkernel"*, e portanto não apropriado para aplicações de tempo real. Por exemplo, em [ENS99] é descrita uma aplicação onde uma aplicação de controle de aproximação de aeronaves em aeroportos 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 *"soft"*. 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, tarefas *"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, as tarefas de tempo real perdem a vantagem que tem com respeito a política de escalonamento e passam a ter o mesmo atendimento que qualquer outra tarefa.

Entretanto, o *"kernel"* do Linux possui um recurso que facilita sua adaptação para o contexto de tempo real. Embora o *"kernel"* seja monolítico e 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. Estes módulos executam em modo privilegiado e são usados normalmente na implementação de tratadores de dispositivos (*"device-drivers"*), sistemas de arquivos e protocolos de rede. 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 *"kernel"* 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"*. No momento que este livro está sendo escrito (início de 2000), existem três projetos envolvendo adaptações do Linux para tempo real citados nesta página, os quais serão descritos a seguir. Vários outros projetos e experiências também foram apresentados no "Real Time Linux Workshop" em Vienna-Austria, em 1999, cujos anais podem ser obtidos em http://www.thinkingnerds.com/projects/rtl-ws/rtl-ws.html. Um dos projetos apresentados foi o LINUX-SMART, desenvolvido no Brasil (IME-USP). Em [Vie99] pode ser encontrada uma descrição do LINUX-SMART, juntamente com uma excelente revisão da problemática relacionada com Linux para tempo real.

Real-Time Linux

O RT-Linux (http://luz.cs.nmt.edu/~rtlinux/) é uma extensão do Linux que se propõe a suportar tarefas com restrições temporais críticas. O seu desenvolvimento iniciou no "Department of Computer Science" do "New Mexico Institute of Technology". Atualmente o sistema é mantido principalmente pela empresa FSMLabs e já está em desenvolvimento a sua segunda versão.

O RT-Linux é um sistema operacional no qual um *"microkernel"* de tempo real co-existe com o *"kernel"* do Linux. O objetivo deste 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"* convencional como sua tarefa de mais baixa prioridade (Tarefa Linux), usando o conceito de máquina virtual para tornar o *"kernel"* convencional e todas as suas aplicações completamente interrompíveis (*"pre-emptable"*).

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"* convencional, o hardware que controla interrupções é emulado. Assim, quando o *"kernel"* convencional "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 tarefas convencionais podem comunicar-se através de filas sem bloqueio 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 tarefas Linux convencionais, as 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"* convencional. Esta solução não suporta requisitos temporais durante a inicialização do sistema, o que na verdade acontece também com todos os sistemas operacionais de tempo real.

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 um escalonador EDF e implementação de 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 estes serviços mais difícil será prever o comportamento 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 no *"microkernel"* tornará este trabalho bem mais fácil.

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 das tarefas 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 tarefas de tempo real que foram registradas com o Alocador. Tarefas convencionais são escalonadas pelo escalonador original do Linux quando nenhuma tarefa de tempo real estiver pronta para executar ou durante o tempo reservado para

tarefas 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 tarefa. O artigo [WaL99] descreve este mecanismo.

O Alocador é utilizado para definir os parâmetros de escalonamento de cada nova tarefa tempo real. Na maioria das aplicações ele pode executar fora do *"kernel"*, como tarefa 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. Tarefas de tempo real inicialmente registram-se com o Alocador que, por sua vez, informa os seus parâmetros para o Disparador. O Alocador executa como a tarefa 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.

O RED-Linux está sendo desenvolvido basicamente na Universidade da California em Irvine, e encontra-se ainda em um estágio inicial. Detalhes sobre o RED-Linux em geral ou sobre o mecanismo de escalonamento proposto em particular podem ser encontrados em [WaL99].

KU Real Time Linux

O KURT-Linux (http://hegel.ittc.ukans.edu/projects/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"* convencional, onde destaca-se o aumento na precisão da marcação da passagem do tempo real e, como consequência, na maior precisão de qualquer ação associada com a passagem do tempo. Uma descrição do mecanismo usado pode ser encontrada em [SPH98]. 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 [HSP98]. Embora este tipo de escalonamento seja menos flexível do que o baseado em prioridades, ele ainda 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.

Segundo os autores do KURT-Linux, é possível integrar as soluções do KURT-Linux com as do RT-Linux e que foram feitas experiências com sucesso neste sentido. Desta forma, seria possível combinar a melhor marcação de tempo e disparo de tarefas do KURT-Linux com a baixa interferência experimentada pelas tarefas de tempo real no RT-Linux.

3.6 Conclusão

Este capítulo procurou discutir aspectos de sistemas operacionais cujo propósito é suportar aplicações de tempo real. Além dos aspectos temporais, obviamente importantes neste contexto, foram discutidos aspectos funcionais importantes, como tarefas e *"threads"*, a comunicação entre elas, instalação de tratadores de dispositivos e interrupções e a disponibilidade de temporizadores.

O texto procurou mostrar as limitações dos sistemas operacionais de propósito geral, no que diz respeito a atender requisitos temporais. As métricas usualmente empregadas para avaliar um SOTR e as dificuldades associadas com a medição foram apresentadas.

A seção 3.4 apresentou uma classificação dos suportes de tempo real, com respeito ao determinismo temporal e a complexidade dos serviços oferecidos. Finalmente, foram descritos alguns sistemas existentes, com especial atenção ao padrão Posix e as propostas de uma versão do Linux para tempo real.

A área de sistemas operacionais de tempo real é muito dinâmica. Novos sistemas ou novas versões dos sistemas existentes são apresentadas a todo momento. O anexo 2 contém uma lista não exaustiva de soluções disponíveis na Internet, com as mais variadas características e preços.

A mensagem central que este capítulo procura transmitir é que o comportamento temporal da aplicação tempo real depende tanto da aplicação em si quanto do sistema operacional. Desta forma, a seleção do SOTR a ser usado depende fundamentalmente dos requisitos temporais da aplicação em questão. Não existe um SOTR melhor ou pior para todas as aplicações. A diversidade de aplicações tempo real existente gera uma equivalente diversidade de sistemas operacionais de tempo real.