Sistemas de Tempo Real

Jean-Marie Farines

Joni da Silva Fraga

Rômulo Silva de Oliveira

Departamento de Automação e Sistemas

Universidade Federal de Santa Catarina

Florianópolis, julho de 2000.

Prefácio

A preocupação com o tempo, entre filósofos e matemáticos, vem desde a antigüidade. A contribuição mais conhecida desse período foi a de Zenon cujo o paradoxo sobre o tempo e o espaço[1] levanta questões de continuidade e de atomicidade no tempo e no espaço.

No mundo atual, a rapidez nas decisões, nas comunicações e nas atividades em geral, se tornou um dos paradigmas dominantes na Sociedade da Informação. Utiliza-se cada vez mais o termo *Tempo Real* em diversas situações, as vezes com propriedade, outras apenas com objetivo comercial. De fato, o tempo está sempre presente em todas as atividades mesmo que não seja de forma explícita; as atividades computacionais seguem também essa regra.

Um número crescente de aplicações de importância na sociedade atual apresentam comportamentos definidos segundo restrições temporais. Alguns exemplos dessas aplicações se encontram no controle de plantas industriais, de tráfego aéreo ou ferroviário, nas telecomunicações, na eletrônica embarcada em carros e aviões, na robótica, em sistemas de multimídia, etc. Essas aplicações que apresentam a característica adicional de estarem sujeitas a restrições temporais, são agrupados no que é normalmente identificado como *Sistemas de Tempo Real*.

A maior parte dos sistemas de tempo real são projetados e implementados com ferramentas convencionais de verificação e de implementação. Por exemplo, na prática corrente, são usadas linguagens de alto nível com construções não deterministas ou mesmo linguagens de baixo nível, mas sem a preocupação de tratar o tempo de uma forma mais explícita o que torna difícil a garantia da implementação das restrições temporais. Os sistemas operacionais e suportes de tempo de execução geralmente utilizados apresentam mecanismos para implementar escalonamentos dirigidos a prioridades; estas nunca refletem as restrições temporais definidas para essas aplicações. Na prática usual, a importância em termos das funcionalidades presentes nessas aplicações são determinantes nas definições dessas prioridades; o que pode ser contestado, pois os possíveis graus de importância de funções em uma aplicação nem sempre se mantém na mesma ordem relativa durante todo o tempo de execução desta. Essas práticas têm permitido resolver de forma aceitável e durante muito tempo certas classes de problemas de tempo real nas quais as exigências de garantia sobre as restrições temporais não são tão estritas.

Entretanto, as necessidades de segurança num número cada vez maior de aplicações e a ligação dessa com a correção temporal desses sistemas colocam em xeque as metodologias e ferramentas convencionais, sob pena de perdas em termos financeiros,

[1]Conhecido como paradoxo de *Aquiles e da tartaruga* que, na essência, mostra uma máquina - Aquiles - que pode realizar cálculos infinitos num tempo finito.

ambiental ou humano. Essas aplicações exigem toda uma demanda de algoritmos, de suportes computacionais e de metodologias que ultrapassa as ferramentas até então utilizadas e lançam de certa forma novos desafios para os projetistas desse tipo de sistemas.

Apesar da evolução nos últimos anos, em termos de conceitos e métodos, para tratar a problemática de sistemas de tempo real, a adoção no setor produtivo, desses novos algoritmos, suportes e metodologias não se dá no mesmo ritmo. Na prática, o uso de meios mais convencionais continua, mesmo no caso de aplicações críticas, o que pode ser a causa de muitas situações desastrosas. Essa lacuna foi constatada em diversas oportunidades de contato com profissionais que atuam diretamente no mercado.

A primeira motivação desse livro se encontra nessa dicotomia entre os avanços teóricos na área de Sistemas de Tempo Real e a prática no mundo real. Esse livro visa introduzir os conceitos básicos de programação tempo real, apresentar duas abordagens metodológicas para a construção das aplicações de tempo real e mostrar a aplicabilidade dos avanços mais recentes da área contidos nessas abordagens a partir da programação de exemplos de aplicações de tempo real.

O livro é escrito com a motivação de servir de guia e de apoio nas atividades de formação na área de Sistemas de Tempo Real e é o resultado da reunião das respectivas experiências de ensino dos autores, envolvendo cursos de graduação, pós-graduação e de reciclagem ministrados nesses últimos anos. Os leitores não necessitam ter um conhecimento prévio do assunto, sendo que os conceitos necessários ao seu entendimento serão introduzidos no inicio dos diversos capítulos do livro e que o conteúdo de cunho mais teórico será sempre acompanhado por figuras e pequenos exemplos ilustrativos. Uma experiência ou um conhecimento básico sobre os mecanismos tradicionais de escalonamento de processos e gerenciamento de recursos, encontrados nos cursos de sistemas operacionais ou de programação concorrente é interessante mas não indispensável para o bom entendimento do conteúdo desse livro.

Apresentação do Conteúdo do Livro

O objetivo desse livro é tratar os principais aspectos relacionados a Sistemas de Tempo Real, dentro de uma visão fundamentada em conceitos e novas técnicas presentes na literatura relacionada, destacando a aplicabilidade concreta das mesmas. Dois tipos de abordagens envolvendo a programação de aplicações de tempo real são introduzidas nesse livro: a abordagem assíncrona baseada no escalonamento de tarefas e a abordagem baseada na hipótese síncrona.

A organização desse livro leva em conta o caráter desse objetivo e as duas abordagens citadas. Após um primeiro capítulo de definições e conceitos, apresenta-se um segundo capítulo que trata do escalonamento de tarefas, um terceiro capítulo que trata de suportes de tempo real e de suas utilizações na abordagem assíncrona, um quarto capítulo que apresenta a abordagem síncrona, um quinto capítulo que discute

essas duas abordagens ao visto de exemplos ilustrativos e um último capítulo de conclusões e perspectivas.

O primeiro capítulo apresenta uma introdução geral ao problema da programação em tempo real. Nesse, é estabelecido o entendimento da noção de tempo real, destacando a terminologia e os conceitos a serem usados em particular os de correção temporal e de previsibilidade. Em seguida, é discutida a problemática de tempo real considerando as aplicações de tempo real mais comuns.

No livro são caracterizados os segundo e terceiro capítulos como definindo uma linha metodológica para a programação de aplicações de tempo real, na qual é priorizado a determinação de uma escala de execução de tarefas que possa atender as restrições de tempo da aplicação (abordagem assíncrona).

O segundo capítulo concentra sua atenção sobre a conceituação e os algoritmos de escalonamento de tempo real, essenciais na garantia da correção temporal dos sistemas de tempo real. Inicialmente conceitos, objetivos, hipóteses e métricas são claramente apresentados no sentido de introduzir o problema de escalonamento. A seguir, diferentes classes de problemas aplicáveis em diferentes contextos de aplicação são tratadas em suas soluções algorítmicas.

O terceiro capítulo discute principalmente aspectos de sistemas operacionais e de núcleos cujo propósito é suportar aplicações de tempo real. Na abordagem apresentada no capítulo anterior, os requisitos temporais são atendidos com algoritmos de escalonamento adequados mas deve levar em conta também as funcionalidades de suportes de tempo de execução no sentido de permitir a previsibilidade ou pelo menos um desempenho satisfatório da aplicação. Entre os aspectos desenvolvidos está a definição da funcionalidade mínima que vai caracterizar os núcleos de tempo real a partir de demandas específicas da aplicação. Finalmente, são apresentados e discutidos padrões, soluções comerciais existentes, protótipos notórios presentes na literatura.

No quarto capítulo, é apresentada a abordagem alternativa na programação de sistemas de tempo real fundamentada na chamada hipótese síncrona e num conjunto de ferramentas integradas de especificação, verificação e implementação. Essa abordagem é particularmente adaptada para sistemas de tempo real concebidos dentro de uma visão de sistemas reativos e que se baseia na hipótese da instantaneidade da reação do sistema a eventos externos. Os conceitos, ferramentas e metodologias relacionadas com essa abordagem são apresentados nesse capítulo, particularizando nas descrições os mesmos para o modelo de programação usado na linguagem síncrona Esterel.

O quinto capítulo confronta as abordagens apresentadas anteriormente (a assíncrona e a síncrona). Essas duas abordagens são utilizadas em exemplos de aplicação do mundo real. Inicialmente é dado destaque para as linhas metodológicas que permitem projetar e implementar essas aplicações nesses dois casos. A seguir são discutidos vantagens e limitações das duas abordagens e a adequação dessas em diferentes situações. No ultimo capítulo, conclusões, desafios e perspectivas complementam esse livro, apontando para os caminhos futuros de evolução da área de Sistemas de Tempo Real.

Índice

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução sobre o Tempo Real

Esse capítulo visa esclarecer o entendimento de tempo real dos autores, definir conceitualmente os Sistemas de Tempo Real e apresentar os problemas e desafios que lhes são relacionados.

1.1 Os Sistemas de Tempo Real

Na medida em que o uso de sistemas computacionais prolifera na sociedade atual, aplicações com requisitos de tempo real tornam-se cada vez mais comuns. Essas aplicações variam muito em relação à complexidade e às necessidades de garantia no atendimento de restrições temporais. Entre os sistemas mais simples, estão os controladores inteligentes embutidos em utilidades domésticas, tais como lavadoras de roupa e videocassetes. Na outra extremidade do espectro de complexidade estão os sistemas militares de defesa, os sistemas de controle de plantas industriais (químicas e nucleares) e o controle de tráfego aéreo e ferroviário. Algumas aplicações de tempo real apresentam restrições de tempo mais rigorosas do que outras; entre esses, encontram-se os sistemas responsáveis pelo monitoramento de pacientes em hospitais, sistemas de supervisão e controle em plantas industriais e os sistemas embarcados em robôs e veículos (de automóveis até aviões e sondas espaciais). Entre aplicações que não apresentam restrições tão críticas, normalmente, são citados os vídeogames, as teleconferências através da Internet e as aplicações de multimídia em geral. Todas essas aplicações que apresentam a característica adicional de estarem sujeitas a restrições temporais, são agrupados no que é usualmente identificado como Sistemas de Tempo Real.

Metodologias e ferramentas convencionais são usadas, em uma prática corrente, no projeto e implementação de sistemas de tempo real. A programação dessas aplicações é feita com o uso de linguagens de alto nível, em geral eficientes, mas com construções não deterministas ou ainda, com linguagens de baixo nível. Em ambos os casos, sem a preocupação de tratar o tempo de uma forma mais explícita, o que torna difícil a garantia de implementação das restrições temporais. Os sistemas operacionais ou núcleos de tempo real, que gerenciam interrupções e tarefas e permitem a programação de temporizadores e de *"timeout"*, são para muitos projetistas as ferramentas suficientes para a construção de sistemas de tempo real. Embora esses suportes apresentem mecanismos para implementar escalonamentos dirigidos a prioridades, essas prioridades nunca refletem as restrições temporais definidas para essas aplicações.

Essas prioridades são determinadas usualmente a partir da importância das funcionalidades presentes nessas aplicações; o que não leva em conta, por exemplo, que o grau de importância relativa de uma função da aplicação nem sempre se mantém igual durante todo o tempo de execução desta.

Essas práticas correntes têm permitido resolver de forma aceitável e durante muito tempo certas classes de problemas de tempo real nas quais as exigências de garantia sobre as restrições temporais não são tão rigorosas. Entretanto essas técnicas e ferramentas convencionais apresentam limitações. Por exemplo, a programação em linguagem *"Assembly"* produz programas com pouca legibilidade e de manutenção complexa e cuja a eficiência está intimamente ligada à experiência do programador. Acrescenta-se a essas limitações, as dificuldades advindas do uso de ferramentas e metodologias que permitem a verificação apenas da correção lógica nessas aplicações. Em conseqüência, o software obtido dessa forma é considerado altamente imprevisível e sem garantia de um comportamento correto durante todo seu tempo de uso; situações perigosas podem resultar da utilização de software assim produzido.

Além do aspecto dessas ferramentas convencionais não tratarem da correção temporal, um outro fato que influi sobre o que se pode considerar como prática corrente é o desconhecimento do que seria tempo real. A grande maioria dos sistemas de tempo real vem sendo projetada e implementada até hoje a partir de uma visão errada de que tempo real é simplesmente reduzido a uma questão de melhoria no desempenho [Sta88].

As exigências de segurança num número cada vez maior de aplicações e a ligação dessa com a correção temporal desses sistemas colocam em xeque as metodologias e ferramentas convencionais, sob pena de perdas em termos financeiro, ambiental ou humano. Essas aplicações necessitam de algoritmos, de suportes computacionais e de metodologias que ultrapassam as características das ferramentas até então utilizadas e lançam novos desafios para os projetistas desse tipo de sistemas. Nas aplicações mais críticas, são assumidos situações extremas e pessimistas com hipóteses de carga de pico e cenários de falhas, sendo que mesmo nestas situações extremas, de pior caso, o sistema de tempo real deve manter as restrições temporais impostas pelo seu ambiente. Há quem considera que a famosa lei de Murphy é o paradigma pessimista ideal a ser considerado para as análises de Sistemas de Tempo Real com restrições temporais críticas [But97].

Antes de apresentarmos abordagens para que se garanta a correção temporal em sistemas de tempo real é necessário que se entenda o conceito de tempo.

1.2 O Tempo: Diferentes Interpretações

A noção de tempo em sistemas informáticos é difícil de ser expressa, podendo assumir diferentes enfoques. Na seqüência abaixo, são apresentados alguns desses enfoques com as suas respectivas interpretações que serão utilizadas nesse livro

[Ray91], [Mot92], [Kop92a], [Jos91]. Alguns desses enfoques são colocados em contraposição:

- *Tempo na Execução* considerado como um recurso a ser gasto durante a execução de um programa como outros recursos físicos ou lógicos e o *Tempo na Programação* visto como uma grandeza a ser manipulada pelo programa como outros tipos de variáveis; na execução o tempo intervém de forma implícita, enquanto na programação, ele pode ser visto de forma implícita ou explícita;
- *Tempo Lógico* que é definido a partir de relações de precedência entre eventos, o que permite o estabelecimento de ordens causais sobre um conjunto de eventos e o *Tempo Físico* que é um tempo métrico que permite expressar quantitativamente a distância entre eventos e estabelecer ordens totais entre eventos;
- *Tempo Denso* que segue a natureza uniforme e continua do tempo físico e é isomorfo ao conjunto dos reais e o *Tempo Discreto*, uma simplificação geralmente aceita do anterior, porém isomorfo em relação ao conjunto dos naturais positivos;
- *Tempo Global*, noção abstrata que permite ao usuário de um sistema distribuído ter acesso a um instante de referência único em qualquer parte do sistema e o *Tempo Local* observável localmente nos diferentes nós de um sistema distribuído; tanto o tempo global quanto o tempo local podem ser físicos ou lógicos;
- *Tempo Absoluto* com referência estabelecida a partir de um evento global e o *Tempo Relativo* tendo como referência um evento local; o tempo absoluto é sempre global, enquanto o tempo relativo é sempre local.

Dependendo dos tipos de sistemas e das abordagens usadas na descrição de seus comportamentos, alguns desses enfoques podem ser assumidos para representar a noção de tempo. O que é relevante num sistema de tempo real é que o tempo, de forma implícita ou explícita, é um componente essencial e intrínseco no comportamento deste.

1.3 Conceituação Básica e Caracterização de um Sistema de Tempo Real

Excetuando sistemas computacionais – normalmente identificados como *Sistemas Transformacionais* que calculam valores de saída a partir de valores de entrada e depois terminam seus processamentos como ocorre, por exemplo, com compiladores, programas de engenharia econômica e programas de cálculo numérico –, uma grande parte dos sistemas computacionais atuais interagem permanentemente com os seus ambientes. Entre esses, distingue-se os chamados *Sistemas Reativos* que reagem enviando respostas continuamente à estímulos de entrada vindos de seus ambientes. Sistemas de tempo real de uma forma geral se encaixam neste conceito de sistemas reativos:

- Um *Sistema de Tempo Real (STR)* é um sistema computacional que deve reagir a estímulos oriundos do seu ambiente em prazos específicos.

O atendimento desses prazos resulta em requisitos de natureza temporal sobre o comportamento desses sistemas. Em conseqüência, em cada reação, o sistema de tempo real deve entregar um resultado correto dentro de um prazo específico, sob pena de ocorrer uma falha temporal. O comportamento correto de um sistema de tempo real, portanto, não depende só da integridade dos resultados obtidos (correção lógica ou *"correctness"*) mas também dos valores de tempo em que são produzidos (correção temporal ou "*timeliness"*). Uma reação que ocorra além do prazo especificado pode ser sem utilidade ou até representar uma ameaça. Descrições semelhantes de sistemas de tempo real são encontradas na literatura da área ([Aud93], [You82], [StR88], [StR90], [Jos91], [Kop92b], [LeL90] e [But97]).

A maior parte das aplicações tempo real se comportam então como sistemas reativos com restrições temporais. A reação dos sistemas de tempo real aos eventos vindo do ambiente externo ocorre em tempos compatíveis com as exigências do ambiente e mensuráveis na mesma escala de tempo. A concepção do sistema de tempo real é diretamente relacionada com o ambiente no qual está relacionado e com o comportamento temporal do mesmo. Na classe de Sistema de Tempo Real na qual se encontram os sistemas embutidos (*"Embedded Systems"*) e os sistemas de supervisão e controle, distingue-se entre o Sistema a Controlar, o Sistema Computacional de Controle e o Operador. O Sistema a Controlar e o Operador são considerados como o Ambiente do Sistema Computacional. A interação entre os mesmos ocorre através de interfaces de instrumentação (compostas de sensores e atuadores) e da interface do operador. A figura 1.1 representa esse tipo de Sistema de Tempo Real.

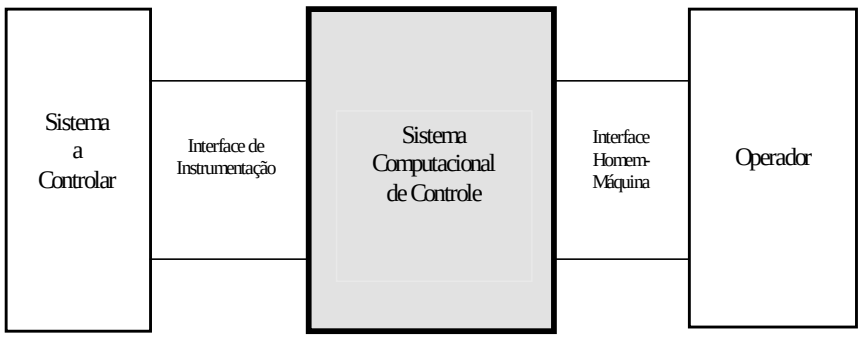


Figura 1.1: Sistema de Tempo Real

Existem também situações nas quais as restrições temporais não são impostas pelo comportamento dinâmico de um eventual Sistema a Controlar mas pelas exigências dos serviços a serem oferecidos a um usuário humano ou computacional (p.ex. no caso do

manuseio ou da apresentação de vídeos em sistemas multimídias). Nesses casos utiliza-se a noção de *Serviço de Tempo Real* como sendo um serviço que deve ser oferecido dentro de restrições de tempo impostas por exigências externas ao próprio Sistema Computacional [DEL91]. Então, numa generalização podemos assumir:

- Um *Sistema de Tempo Real* deve ser então capaz de oferecer garantias de *correção temporal* para o fornecimento de todos os seus serviços que apresentem restrições temporais.

1.4 A Previsibilidade nos Sistemas de Tempo Real

Uma das crenças mais comuns é que o problema de tempo real se resolve pelo aumento da velocidade computacional. A rapidez de cálculo visa melhorar o desempenho de um sistema computacional, minimizando o tempo de resposta médio de um conjunto de tarefas, enquanto o objetivo de um cálculo em tempo real é o atendimento dos requisitos temporais de cada uma das atividades de processamento caracterizadas nesses sistemas [Sta88]. Ter um tempo de resposta curto, não dá nenhuma garantia que os requisitos temporais de cada processamento no sistema serão atendidos. O desempenho médio não tem importância para o comportamento de um sistema composto de diversas atividades com restrições temporais. Mais do que a rapidez de cálculo, para os sistemas de tempo real, importa o conceito de *previsibilidade*.

Um sistema de tempo real é dito ser *previsível* (*"predictable*") no domínio lógico e no domínio temporal quando, independentemente de variações ocorrendo à nível de hardware (i.e. desvios do relógio), da carga e de falhas, o comportamento do sistema pode ser antecipado, antes de sua execução. Para se poder prever a evolução de um sistema de tempo real e garantir dentro de certos limites as suas restrições temporais, é necessário definir um conjunto de hipóteses sobre o comportamento do ambiente externo no que diz respeito à carga e as falhas:

- *A hipótese de carga:* ela determina o que corresponde a carga computacional de pico (carga máxima) gerada pelo ambiente em um intervalo mínimo de tempo, entre cada reação do sistema de tempo real. Mesmo eventos que ocorrem esporadicamente como os que levam a situações críticas (p.ex. alarme em planta nuclear) devem ser levados em conta para determinar essa carga computacional;
- *A hipótese de falhas*: ela descreve os tipos e freqüências de falhas com os quais o sistema deve conviver em tempo de execução, continuando a atender os seus requisitos funcionais e temporais.

Consequentemente, de um ponto de visto rigoroso, para se assumir a previsibilidade de um sistema (ou de um serviço) de tempo real, precisa-se conhecer *a priori* o comportamento de um sistema, levando-se em conta a pior situação de carga ocorrendo, simultaneamente, com as hipóteses de falhas. Como se vê é necessário nesse caso que

as hipóteses de carga e de falha que descrevem o comportamento do ambiente sejam definidas de forma realista, o que nem sempre é uma tarefa simples.

Mas a garantia de previsibilidade não depende só da carga computacional ativada pelo ambiente e das hipóteses de falhas. Um conjunto de fatores ligados a arquitetura de hardware, ao sistema operacional e as linguagens de programação são também importantes. Ou seja, os tempos gastos, no pior caso, na execução de códigos da aplicação (tempo de computação) e em funções de suportes devem ser perfeitamente conhecidos ou determinados previamente. Esses tempos limites são necessários nas análises e verificações do comportamento temporal do sistema de tempo real.

Se consideramos os tempos de computação dos códigos de aplicação envolvidos, muitas das construções nas linguagens de programação de caracter geral são fontes de não determinismo, tornando os sistemas não previsíveis. Como exemplo, podemos citar construções de recorrência não limitadas (laços não limitados) ou ainda, primitivas da linguagem Ada como o *"delay"* que estipula um limite mínimo mas não garante um limite máximo na suspensão de uma tarefa e o *"select"* que na ativação de alternativas de fluxos de instruções faz uma escolha aleatória entre aquelas com guardas abertos. Os mecanismos e protocolos de acesso a recursos compartilhados disponíveis nestas linguagens, geralmente não limitados, podem também tornar os códigos programados não previsíveis. Entretanto, melhorias são obtidas em linguagens destinadas a programação de tempo real que eliminam chamadas recursivas e estruturas dinâmicas de dados e onde são permitidos apenas laços limitados no tempo. Além disso, estas linguagens permitem expressar comportamentos temporais através de construções apropriadas. Real-Time Euclid [KSt86] e Real-Time Concurrent C [GeR91] são exemplos dessas linguagens.

O hardware também influi sobre a previsibilidade de comportamento dos sistemas de tempo real. O processador com suas instruções de *"prefetch"*, os dispositivos de acesso direto à memória (DMA), e mecanismos de memória *"cache"* são outras fontes de não determinismo; o uso destes mecanismos ou dispositivos dificulta a determinação dos tempos de computação no pior caso e consequentemente a análise da previsibilidade. As características de sistemas operacionais e suportes são também determinantes para que se conheça os tempos limites envolvidos em seus processamentos.

Diante do exposto acima, é importante enfatizar que em cada etapa do ciclo de desenvolvimento de um Sistema de Tempo Real, torna-se necessário o uso de ferramentas e metodologias apropriadas que permitem verificar o comportamento do sistema e sua implementação como previsíveis. A previsibilidade é o requisito necessário que deve ser introduzido em paradigmas de computação clássico para se poder atender aplicações de tempo real, em particular quando estas apresentam requisitos temporais rígidos.

Nesta seção, a noção de previsibilidade que foi introduzida está associada a uma antecipação determinista do comportamento temporal do sistema. Ou seja, o sistema é previsível quando podemos antecipar que todos os prazos colocados a partir das

interações com o seu ambiente serão atendidos. Mas na literatura, alguns autores como em [JLT85] também associam o conceito de previsibilidade a uma antecipação probabilista do comportamento do sistema, baseada em estimativas ou simulações que estipulam probabilidades dos prazos a serem atendidos. A previsibilidade probabilista é útil em sistemas onde a carga computacional não pode ser conhecida a priori e portanto, não se consegue uma garantia em tempo de projeto do atendimento de todos os prazos.

As discussões apresentadas acima sobre a previsibilidade, no que se refere a aspectos de implementação, estão ligadas a um tipo de abordagem na construção de sistemas de tempo real. Existem outras abordagens onde fatores ligados à implementação não são levados em conta e por hipótese, válida num grande número de aplicações, o sistema de tempo real é assumido com tendo um comportamento determinista e por conseqüência a sua previsibilidade garantida. Essas abordagens que apresentam formas distintas de conseguir a previsibilidade de sistemas de tempo real são introduzidas no item 1.6 e apresentadas em detalhe no transcorrer desse livro.

1.5 Classificação dos Sistemas de Tempo Real

Os sistemas de tempo real podem ser classificados a partir do ponto de vista da Segurança (*"Safety"*) em: Sistemas Não Críticos de Tempo Real (ou STRs brandos, *"Soft Real Time Systems"*) quando as conseqüências de uma falha devida ao tempo é da mesma ordem de grandeza que os benefícios do sistema em operação normal (ex. sistema de comutação telefônico, sistema de processamento bancário); e Sistemas Críticos de Tempo Real (ou STRs duros, *"Hard Real Time Systems"*), quando as conseqüências de pelo menos uma falha temporal excedam em muito os benefícios normais do sistema (ex. sistema de controle de vôo, ou de sinalização em ferrovias, sistema de controle de planta nuclear). Nesse caso, essa falha é dita catastrófica. Previsibilidades probabilista e determinista são associadas aos dois grupos distinguidos acima, respectivamente.

O grupo de sistemas críticos de tempo real pode ser subdividido em: Sistemas de Tempo Real Crítico Seguros em Caso de Falha (*"fail safe"*) onde um ou vários estados seguros podem ser atingidos em caso de falha (por exemplo, parada obrigatória de trens no caso de falha do sistema de sinalização ferroviário); e Sistemas de Tempo Real Crítico Operacionais em Caso de Falha (*"fail operational"*) que, na presença de falhas parciais, podem se degradar fornecendo alguma forma de serviço mínimo (por exemplo, sistema de controle de vôo com comportamento degradado mas ainda seguro).

Alguns autores apresentam a sua visão de sistemas de tempo real baseada no ponto de vista da implementação, distinguindo então: Sistemas de Resposta Garantida ("*guaranteed response system*") onde existem recursos suficientes para suportar a carga de pico e o cenário de falhas definido; e Sistemas de Melhor Esforço (*"best effort system"*) quando a estratégia de alocação dinâmica de recursos se baseia em estudos

probabilistas sobre a carga esperada e os cenários de falhas aceitáveis. Essa classificação é similar a anterior. A maior parte dos sistemas de tempo real vem sendo projetados de acordo com o paradigma de melhor esforço, satisfatório de fato apenas para os sistemas não críticos de tempo real e levando a situações danosas para os sistemas críticos de tempo real para os quais é aconselhado seguir o paradigma de resposta garantida.

1.6 O Problema Tempo Real e Abordagens para a sua Solução

O Problema Tempo Real consiste então em especificar, verificar e implementar sistemas ou programas que, mesmo com recursos limitados, apresentam comportamentos previsíveis, atendendo as restrições temporais impostas pelo ambiente ou pelo usuário [Jos91]. Se consideramos esses aspectos de construção, tempo real pode ser visto inicialmente como um problema intrínseco de programação concorrente. Baseado então na maneira de tratar a concorrência surgiram duas abordagens diferentes amplamente discutidas na literatura nesses vinte últimos anos: a Abordagem Assíncrona1 e a Abordagem Síncrona.

A abordagem assíncrona cujo o entendimento mais usual foi introduzido por R. Milner [Mil80] trata a ocorrência e a percepção de eventos independentes numa ordem arbitrária mas não simultânea. As linguagens CSP, Ada e Real-Time Concurrent C seguem o paradigma definido nessa abordagem e podem ser consideradas como linguagens assíncronas. Essa abordagem visa uma descrição a mais exata possível de um sistema; para tal, baseia-se na observação durante a execução de todas as combinações de ocorrência de eventos (de forma não simultânea), conhecida como entrelaçamento de eventos (*"interleaving"*). Essa abordagem é considerada como orientada à implementação. Consequentemente, se torna necessário levar em conta na especificação e no decorrer do projeto, algumas características do suporte de software e de hardware das aplicações, o que fere eventuais exigências em termos de portabilidade. Por outro lado, a procura de uma descrição completa do comportamento e a introdução de considerações de implementação torna complexa a análise das propriedades do sistema por causa da necessidade de tratar com grande número de estados e do possível não determinismo introduzido. A abordagem assíncrona apresentada nesse livro é implementada usando linguagens e sistemas operacionais e está fundamentada no tratamento explícito da concorrência e do tempo de uma aplicação em tempo de execução; a questão do escalonamento de tempo real é o ponto principal do estudo da previsibilidade dos sistemas de tempo real.

A abordagem síncrona na forma introduzida em [Ber89] e [BeB91] tem o seu

[1] O termo *Abordagem Assíncrona* utilizado neste livro não é consenso na literatura de tempo real.

princípio básico na consideração que os cálculos e as comunicações não levam tempo. A abordagem síncrona se coloca num nível de abstração dos aspectos de implementação tal que o tempo é visto com granularidade suficientemente grossa para que essa hipótese seja verdadeira. A descrição do comportamento do sistema é nessa abordagem menos dependente das questões de implementação, o que é satisfatório do ponto de vista da portabilidade. A observação dos eventos nessa abordagem é cronológica, permitindo uma eventual simultaneidade entre eles. Nesta abordagem, a partir das suas premissas, a concorrência é resolvida sem o entrelaçamento de tarefas (*"interleaving"*) e o tempo não é tratado de maneira explícita. A abordagem síncrona é considerada como orientada ao comportamento de uma aplicação e a sua verificação. Por se situar num nível de abstração maior que no caso da abordagem anterior, a abordagem síncrona facilita a especificação e a análise das propriedades de sistemas de tempo real. Diversas linguagens como Esterel, Statecharts, Signal, Lustre – chamadas de linguagens síncronas -- seguem o paradigma definido nessa abordagem.

Nesse livro, apresentamos soluções que seguem essas duas abordagens para tratar o problema de tempo real. A abordagem assíncrona, dependente das ferramentas de implementação e que trata explicitamente o tempo e a concorrência têm introduzidos os conhecimentos necessários para o seu uso nos capítulos 2 e 3. A abordagem síncrona baseada na hipótese de sincronismo é discutida no capítulo 4. Exemplos ilustrativos permitindo entender essas duas abordagens, mostrando suas vantagens e limitações são mostrados no capítulo 5. É objetivo nesse livro apresentar os tipos de aplicações nas quais cada uma dessas abordagens é mais adaptada.

Capítulo 2

O Escalonamento de Tempo Real

Em sistemas de tempo real que seguem a abordagem assíncrona os aspectos de implementação estão presentes mesmo na fase de projeto. Na implementação de restrições temporais, é de fundamental importância o conhecimento das propriedades temporais do suporte de tempo de execução usado e da escolha de uma abordagem de escalonamento de tempo real adequada à classe de problemas que o sistema deve tratar. Neste sentido, este capítulo e o próximo apresentam aspectos da teoria de escalonamento e de sistemas operacionais sob a ótica de tempo real.

Este capítulo trata sobre escalonamento de tempo real de um modo geral. Conceitos, objetivos, hipóteses e métricas são claramente apresentados no sentido de introduzir o que chamamos de um *problema de escalonamento*. Posteriormente, diferentes classes de problemas de escalonamento são examinadas em suas soluções algorítmicas.

2.1 Introdução

Em sistemas onde as noções de tempo e de concorrência são tratadas explicitamente, conceitos e técnicas de escalonamento formam o ponto central na previsibilidade do comportamento de sistemas de tempo real. Nos últimos anos, uma quantidade significativa de novos algoritmos e de abordagens foi introduzida na literatura tratando de escalonamento de tempo real. Infelizmente muitos desses trabalhos definem técnicas restritas e conseqüentemente de uso limitado em aplicações reais. Esse capítulo se concentra em algumas técnicas gerais e em suas extensões, visando também à perspectiva de um uso mais prático.

O foco desse capítulo é sobre técnicas para escalonamentos dirigidos a prioridades. Essa escolha é devido à importância da literatura disponível e, porque cobre diversos aspectos de possíveis comportamentos temporais em aplicações de tempo real. A grande difusão de suportes (núcleos, sistemas operacionais), na forma de produtos, que baseiam seus escalonamentos em mecanismos dirigidos a prioridade é sem dúvida outra justificativa bastante forte para a escolha do enfoque dado nesse capítulo. Essa difusão é tão significativa que organismos de padronização têm adotado essa abordagem de escalonamento de tempo real. Os padrões POSIX [Gal95] - referência para produtos comerciais - enfatizam em suas especificações para tempo real escalonamentos dirigidos a prioridades. Alguns dos algoritmos apresentados nesse capítulo são recomendados pelas especificações POSIX.

Na seqüência é introduzido um conjunto de conceitos que permitem a caracterização de um *problema de escalonamento*.

2.2 Modelo de Tarefas

O conceito de *tarefa* é uma das abstrações básicas que fazem parte do que chamamos um problema de escalonamento. Tarefas ou processos formam as unidades de processamento seqüencial que concorrem sobre um ou mais recursos computacionais de um sistema. Uma simples aplicação de tempo real é constituída tipicamente de várias tarefas. Uma tarefa de tempo real, além da correção lógica (*"correctness"*), deve satisfazer seus prazos e restrições temporais ou seja, apresentar também uma correção temporal (*"timeliness"*).

As restrições temporais, as relações de precedência e de exclusão usualmente impostas sobre tarefas são determinantes na definição de um *modelo de tarefas* que é parte integrante de um problema de escalonamento. Nas seções subseqüentes descrevemos essas formas de restrições que normalmente estão presentes em processamentos de tempo real.

2.2.1 Restrições Temporais

Aplicações de tempo real são caracterizadas por restrições temporais que devem ser respeitadas para que se tenha o comportamento temporal desejado ou necessário. Todas as tarefas de tempo real tipicamente estão sujeitas a prazos: os seus *"deadlines"*. A princípio, uma tarefa deve ser concluída antes de seu *"deadline"*. As conseqüências de uma tarefa ser concluída após o seu *"deadline"* define dois tipos de tarefas de tempo real:

- *Tarefas Críticas* (tarefas *"hard"*) : Uma tarefa é dita crítica quando ao ser completada depois de seu *"deadline"* pode causar falhas catastróficas no sistema de tempo real e em seu ambiente. Essas falhas podem representar em danos irreversíveis em equipamentos ou ainda, em perda de vidas humanas.

- *Tarefas Brandas ou Não Críticas* (tarefas *"soft"*): Essas tarefas quando se completam depois de seus *"deadlines"* no máximo implicam numa diminuição de desempenho do sistema. As falhas temporais nesse caso são identificadas como benignas onde a conseqüência do desvio do comportamento normal não representa um custo muito significativo.

Outra característica temporal de tarefas em sistemas de tempo real está baseada na regularidade de suas ativações. Os modelos de tarefa comportam dois tipos de tarefas segundo suas freqüências de ativações:

- *Tarefas Periódicas*: Quando as ativações do processamento de uma tarefa ocorrem, numa seqüência infinita, uma só ativação por intervalo regular chamado de *Período*, essa tarefa é identificada como periódica. As ativações de uma tarefa periódica formam o conjunto de diferentes *instâncias* da tarefa. Nesse texto assumimos a primeira ativação de uma tarefa periódica ocorrendo na origem dos tempos considerados na aplicação (em *t=0*).

- *Tarefas Aperiódicas ou Tarefas Assíncronas*: Quando a ativação do processamento de uma tarefa responde a eventos internos ou externos definindo uma característica aleatória nessas ativações, a tarefa é dita aperiódica.

As tarefas periódicas pela regularidade e portanto pela previsibilidade, usualmente são associadas a *"deadlines hard"*, ou seja, são tarefas críticas. As tarefas aperiódicas pela falta de previsibilidade em suas ativações, normalmente, tem *"deadlines soft"* associados a suas execuções, compondo portanto as tarefas brandas de um sistema de tempo real. *Tarefas esporádicas* que correspondem a um subconjunto das tarefas aperiódicas, apresentam como característica central a restrição de um intervalo mínimo conhecido entre duas ativações consecutivas e por isso, podem ter atributos de tarefas críticas. As tarefas esporádicas portanto são também associadas a *"deadlines hard"*. As figuras 2.1 e 2.2 apresentam características temporais de tarefas periódicas e aperiódicas, respectivamente.

Outras restrições temporais são importantes na definição do comportamento temporal de uma tarefa:

- *Tempo de computação* (*"Computation Time"*): O tempo de computação de uma tarefa é o tempo necessário para a execução completa da tarefa.

- *Tempo de início* ("*Start Time"*): Esse tempo corresponde ao instante de início do processamento da tarefa em uma ativação.

- *Tempo de término* (*"Completion Time"*): É o instante de tempo em que se completa a execução da tarefa na ativação.

- *Tempo de chegada* (*"Arrival Time"*): O tempo de chegada de uma tarefa é o instante em que o escalonador toma conhecimento de uma ativação dessa tarefa. Em tarefas periódicas, o tempo de chegada coincide sempre com o início do período da ativação. As tarefas aperiódicas apresentam o tempo de chegada coincidindo com o tempo da requisição do processamento aperiódico.

- *Tempo de liberação* (*"Release Time"*): O tempo de liberação de uma tarefa coincide com o instante de sua inclusão na fila de Pronto (fila de tarefas prontas) para executar.

Dependendo do modelo de tarefas assumido o tempo de liberação pode ou não

coincidir com o tempo de chegada da tarefa. Em geral é assumido que tão logo uma instância de uma tarefa chegue, a mesma é liberada na fila de Pronto. Mas, nem sempre esse é o caso; uma tarefa pode ser retardada na sua liberação pelo *"polling"* de um escalonador ativado por tempo (*"tick scheduler"*) ou talvez pelo bloqueio na recepção de uma mensagem (tarefas ativadas por mensagem). Essa não coincidência dos tempos de chegada com as liberações da tarefa conduz ao que é identificado como *"Release Jitter"*, que representa a máxima variação dos tempos de liberação das instâncias da tarefa.

Diante das restrições temporais citadas temos então o comportamento temporal de uma tarefa periódica T_i descrito pela quádrupla (J_i, C_i, P_i, D_i) onde C_i representa o tempo de computação da tarefa, P_i é o período da tarefa, D_i é o *"deadline"* e J_i é o *"Release Jitter"* da tarefa que, de certa maneira, corresponde a pior situação de liberação da tarefa. Nessa representação de tarefas periódicas, J_i e D_i são grandezas relativas (intervalos), medidas a partir do início do período P_i. O *"deadline"* absoluto e o tempo de liberação da $k^{ésima}$ ativação da tarefa periódica T_i são determinados a partir dos períodos anteriores:

$$d_{ik} = (k\text{-}1)P_i + D_i$$
$$r_i = (k\text{-}1)P_i + J_i \qquad \text{(pior situação de liberação).}$$

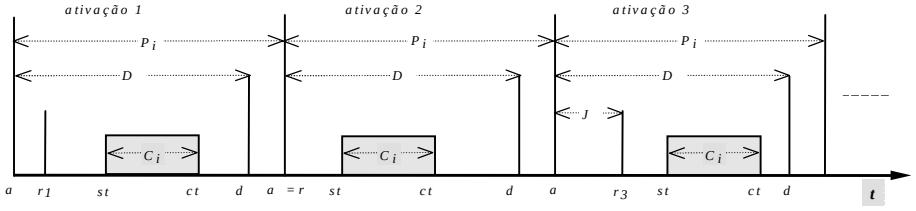


Figura 2.1: Ativações de uma tarefa periódica

Na figura 2.1 é ilustrado alguns dos parâmetros descritos acima. Porém, cada ativação da tarefa periódica (J_i, C_i, P_i, D_i) é definida a partir de tempos absolutos: os tempos de chegada (a_i), os tempos de liberação (r_i), os tempos de início (st_i), os tempos de término (ct_i) e os *"deadlines"* absolutos (d_i).

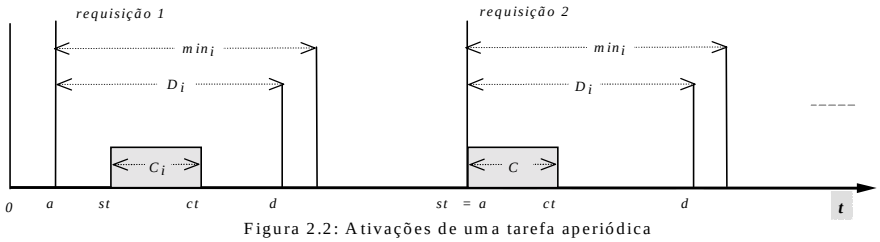


Figura 2.2: Ativações de uma tarefa aperiódica

Uma tarefa esporádica é descrita pela tripla (C_i, D_i, min_i) onde C_i é o tempo de computação, D_i é o *"deadline"* relativo medido a partir do instante da requisição do processamento aperiódico (chegada da tarefa esporádica) e min_i corresponde ao mínimo intervalo entre duas requisições consecutivas da tarefa esporádica. A descrição de uma tarefa aperiódica pura se limita apenas às restrições C_i e D_i. Na figura 2.2, a tarefa aperiódica esporádica (C_i, D_i, min_i) é apresentada com duas requisições. Tomando o tempo de chegada da requisição esporádica 2 como a_2, o *"deadline"* absoluto desta ativação assume o valor dado por: $d_2=a_2+D_i$.

2.2.2 Relações de Precedência e de Exclusão

Em aplicações de tempo real, muitas vezes, os processamentos não podem executar em ordem arbitrária. Implicações semânticas definem *relações de precedência* entre as tarefas da aplicação determinando portanto, ordens parciais entre as mesmas. Uma tarefa T_j é precedida por uma outra T_i ($T_i \rightarrow T_j$), se T_j pode iniciar sua execução somente após o término da execução de T_i. Relações de precedência podem também expressar a dependência que tarefas possuem de informações (ou mesmo sinais de sincronização) produzidas em outras tarefas. As relações de precedência em um conjunto de tarefas usualmente são representadas na forma de um grafo acíclico orientado, onde os nós correspondem às tarefas do conjunto e os arcos descrevem as relações de precedência existentes entre as tarefas.

O compartilhamento de recursos em exclusão mútua define outra forma de relações entre tarefas também significativas em escalonamentos de tempo real: as *relações de exclusão*. Uma tarefa T_i exclui T_j quando a execução de uma seção crítica de T_j que manipula o recurso compartilhado não pode executar porque T_i já ocupa o recurso. Relações de exclusão em escalonamentos dirigidos a prioridade podem levar a *inversões de prioridades* onde tarefas mais prioritárias são bloqueadas por tarefas menos prioritárias.

As relações de precedência e de exclusão serão retomadas no item que trata sobre os algoritmos de escalonamento dirigidos a prioridades.

2.3 Escalonamento de Tempo Real

2.3.1 Principais Conceitos

O termo *escalonamento* (*"scheduling"*) identifica o procedimento de ordenar tarefas na fila de Pronto. Uma escala de execução (*"schedule"*) é então uma ordenação ou lista que indica a ordem de ocupação do processador por um conjunto de tarefas disponíveis

na fila de Pronto. O *escalonador* (*"scheduler"*) é o componente do sistema responsável em tempo de execução pela gestão do processador. É o escalonador que implementa uma *política de escalonamento* ao ordenar para execução sobre o processador um conjunto de tarefas.

Políticas de escalonamento definem critérios ou regras para a ordenação das tarefas de tempo real. Os escalonadores utilizando então essas políticas produzem escalas que se forem *realizáveis* (*"feasible"*), garantem o cumprimento das restrições temporais impostas às tarefas de tempo real. Uma escala é dita *ótima* se a ordenação do conjunto de tarefas, de acordo com os critérios pré-estabelecidos pela política de escalonamento, é a melhor possível no atendimento das restrições temporais.

Tendo como base a forma de cálculo da escala (ordenação das tarefas), algumas classificações são encontradas para a grande variedade de algoritmos de escalonamento de tempo real encontrados na literatura [AuB90, But97, CSR88]. Os algoritmos são ditos *preemptivos* ou *não preemptivos* quando em qualquer momento tarefas se executando podem ou não, respectivamente, ser interrompidas por outras mais prioritárias. Algoritmos de escalonamento são identificados como *estáticos* quando o cálculo da escala é feito tomando como base parâmetros atribuídos às tarefas do conjunto em tempo de projeto (parâmetros fixos). Os dinâmicos, ao contrário, são baseados em parâmetros que mudam em tempo de execução com a evolução do sistema. Os algoritmos de escalonamento que produzem a escala em tempo de projeto são identificados como algoritmos *"off-line"*. Se a escala é produzida em tempo de execução o algoritmo de escalonamento é dito de *"on-line"*. A partir dessas classificações podemos ter algoritmos off-line estáticos, on-line estáticos e on-line dinâmicos. Essas classificações serão revistas na apresentação de algoritmos de escalonamento.

Um *problema de escalonamento*, na sua forma geral, envolve um conjunto de processadores, um conjunto de recursos compartilhados e um conjunto de tarefas especificadas segundo um modelo de tarefas definindo restrições temporais, de precedência e de exclusão. O escalonamento de tempo real, na sua forma geral, é identificado como um problema intratável (NP-completo [GaJ79], [AuB90]). Muito freqüentemente os algoritmos existentes representam uma solução polinomial para um problema de escalonamento particular, onde um conjunto de hipóteses podem expressar simplificações no modelo de tarefas ou ainda na arquitetura do sistema, no sentido de diminuir a complexidade do problema. Quando nenhuma simplificação é usada para abrandar a complexidade no escalonamento, uma heurística é usada para encontrar uma escala realizável ainda que não sendo ótima mas que garanta as restrições do problema.

Os algoritmos de escalonamento estão então ligados a classes de problemas de escalonamento de tempo real. Um algoritmo é identificado como *ótimo* se minimiza algum custo ou métrica definida sobre a sua classe de problema. Quando nenhum custo ou métrica é definido, a única preocupação é então encontrar uma escala realizável. Nesse caso, o algoritmo é dito *ótimo* quando consegue encontrar uma escala realizável para um conjunto de tarefas sempre que houver um algoritmo da mesma classe que também chega a uma escala realizável para esse mesmo conjunto; se o algoritmo ótimo

falha em um conjunto de tarefas na determinação de uma escala realizável então todos os algoritmos da mesma classe de problema também falharão.

2.3.2 Abordagens de Escalonamento

Uma aplicação de tempo real é expressa na forma de um conjunto de tarefas e, para efeito de escalonamento, o somatório dos tempos de computação dessas tarefas na fila de Pronto determina a carga computacional (*"task load"*) que a aplicação constitui para os recursos computacionais em um determinado instante. Uma carga toma características de *carga estática ou limitada* quando todas as suas tarefas são bem conhecidas em tempo de projeto na forma de suas restrições temporais, ou seja, são conhecidas nas suas condições de chegada (*"arrival times"* das tarefas). O fato de conhecer *a priori* os tempos de chegada torna possível a determinação dos prazos a que uma carga está sujeita. As situações de pico (ou de pior caso) nestas cargas são também conhecidas em tempo de projeto. Cargas estáticas são modeladas através de tarefas periódicas e esporádicas.

Cargas dinâmicas ou ilimitadas ocorrem em situações onde as características de chegada das tarefas não podem ser antecipadas. Essas cargas são modeladas usando tarefas aperiódicas, ou seja, basta que se tenha uma tarefa aperiódica no conjunto cujo intervalo mínimo entre requisições seja nulo e teremos as condições de pico dessa carga desconhecida em tempo de projeto.

O escalonamento é muitas vezes dividido em duas etapas. Um *teste de escalonabilidade* é inicialmente executado, no sentido de determinar se as restrições temporais de um conjunto de tarefas são atendidas considerando os critérios de ordenação definidos no algoritmo de escalonamento. A segunda etapa envolve então o cálculo da escala de execução.

Diferentes abordagens de escalonamento são identificadas na literatura, tomando como base o tipo de carga tratado e as etapas no escalonamento identificados acima. Em [RaS94] é definida uma taxonomia que identifica três grupos principais de abordagens de escalonamento de tempo real: as abordagens com garantia em tempo de projeto (*"off-line guarantee"*), com garantia em tempo de execução (*"on-line guarantee"*) e abordagens de melhor esforço (*"best-effort"*).

O grupo *garantia em tempo de projeto* é formado por abordagens que tem como finalidade a previsibilidade determinista. A garantia em tempo de projeto é conseguida a partir de um conjunto de premissas:

- a carga computacional do sistema é conhecida em tempo de projeto (carga estática);
- no sistema existe uma reserva de recursos suficientes para a execução das tarefas, atendendo suas restrições temporais, na condição de pior caso.

O fato de se conhecer *a priori* a carga (conhecer os tempos de chegada das tarefas), permite que se possa fazer testes de escalonabilidade, determinando se o conjunto de tarefas considerado é escalonável e, portanto, garantindo suas restrições temporais ainda em tempo de projeto. O conhecimento prévio da situação de pior caso (situação de pico) permite até que se possa redimensionar o sistema de modo a garantir as restrições temporais mesmo nessa situação de pico. O grupo de abordagens com *garantia em tempo de projeto* é próprio para aplicações deterministas ou críticas onde a carga é conhecida previamente, como em aplicações embarcadas, controle de tráfego ferroviário, controle de processos em geral, etc.

São basicamente dois os tipos de abordagens com garantia em tempo de projeto: *o executivo cíclico* e *os escalonamentos dirigidos a prioridades*. No *executivo cíclico* ([Kop97] [XuP93]) ambos, o teste de escalonabilidade e a produção da escala, são realizados em tempo de projeto. Essa escala (ou grade), definida em tempo de projeto, é de tamanho finito e determina a ocupação dos *slots* do processador em tempo de execução. O teste de escalonabilidade fica implícito no processo de montagem da escala. A complexidade dos algoritmos ou heurísticas usadas no cálculo da escala depende da abrangência do modelo de tarefas usado. Modelos mais complexos envolvem o uso de funções heurísticas para dirigir técnicas de *"branch and bound*" na procura de escalas realizáveis [Pin95].

Essa abordagem é chamada de *executivo cíclico* porque o escalonador em tempo de execução se resume a um simples dispachante, ativando tarefas segundo a grade, ciclicamente. A escala calculada nessa abordagem, em tempo de projeto, reflete o pior caso[1]. Como o pior caso é distante do caso médio e nem sempre ocorre, a ativação das tarefas segundo essas grades, embora satisfaça às restrições temporais, implica em desperdício de recursos que são sempre reservados para o pior caso.

Abordagens com garantia em tempo de projeto baseadas em escalonadores *dirigidos a prioridades* são mais flexíveis. O teste de escalonamento é realizado em tempo de projeto enquanto a escala é produzida *"on-line"* por um escalonador dirigido a prioridades. As tarefas têm suas prioridades definidas segundo políticas de escalonamento que envolvem atribuições estáticas (prioridades fixas) ou dinâmicas (prioridades variáveis), caracterizando escalonadores on-line estáticos ou dinâmicos. Nessa abordagem, o pior caso se reflete apenas no teste de escalonabilidade que é executado em tempo de projeto, decidindo se o conjunto de tarefas é escalonável ou não. Por não existir uma reserva de recursos em tempo de execução como no *executivo cíclico*, os *escalonamentos dirigidos a prioridades* definem soluções mais flexíveis, porém, mantendo também garantias de recursos para o pior caso.

Os grupos de abordagens *garantia dinâmica* (*"on-line guarantee"*) e *de melhor*

[1] No cálculo da escala o pior caso é considerado, levando em conta a pior situação de chegada das tarefas, as tarefas se executando com os seus piores tempos de computação, as tarefas esporádicas ocorrendo na máxima freqüência possível definida por seus intervalos mínimos entre ativações (min_i), etc.

esforço (*"best-effort"*), ao contrário do grupo anterior, não tratam com uma carga computacional previsível; na verdade, a carga tratada por essas abordagens é dinâmica: os tempos de chegada das tarefas não são conhecidos previamente. Quando o pior caso não pode ser antecipado em tempo de projeto, não se consegue prever recursos para todas as situações de carga. Não existindo portanto, a possibilidade de se antecipar que todas as tarefas, em qualquer situação de carga, terão sempre seus *"deadlines"* respeitados. Essas abordagens que tratam com carga dinâmica devem então lidar com situações, em tempo de execução, onde os recursos computacionais são insuficientes para os cenários de tarefas que se apresentam (situações de sobrecarga). Tanto a escala como os testes de escalonabilidade são realizados em tempo de execução nessas abordagens (escalonadores *"on-line"* e dinâmicos).

O grupo de abordagens *com garantia dinâmica* ([RaS94]) se utilizam de um teste para verificar a escalonabilidade do conjunto formado por uma nova tarefa que chega no sistema e das tarefas que já existiam previamente na fila de pronto. Esses testes são chamados de *testes de aceitação* e estão baseados em análises realizadas com hipóteses de pior caso sobre alguns parâmetros temporais. Se o teste indica o conjunto como não escalonável, a nova tarefa que chegou é então descartada. Esse mecanismo de garantia dinâmica em qualquer situação preserva as tarefas já previamente garantidas como escalonáveis. Essas abordagens que oferecem *garantia dinâmica* são próprias para aplicações que possuam restrições críticas mas que operam em ambientes não deterministas. Sistemas militares, sistemas de radar e controle aéreo exemplificam alguns desses sistemas que estão sujeitos a cargas dinâmicas e necessitam atender restrições temporais.

As abordagens de melhor esforço são constituídas por algoritmos que tentam encontrar uma escala realizável em tempo de execução sem realizar testes ou ainda, realizando testes mais fracos. Não existe a garantia de execuções de tarefas atendendo suas restrições temporais. Essas abordagens são adequadas para aplicações não críticas, envolvendo tarefas *"soft"* onde a perda de *"deadlines"* não representa custos além da diminuição do desempenho nessas aplicações. As abordagens de melhor esforço são adequadas para aplicações de tempo real brandas como sistemas de multimídia.

O desempenho de esquemas de melhor esforço, na ocorrência de casos médios, é melhor que os baseados em garantia. As hipóteses pessimistas feitas em abordagens com garantia dinâmica podem desnecessariamente descartar tarefas. Nas abordagens de melhor esforço tarefas são abortadas somente em condições reais de sobrecarga, na ocorrência de falhas temporais (*"deadline"*s não podem ser atendidos).

Algumas técnicas foram introduzidas no sentido de tratar com sobrecargas. A *Computação Imprecisa* [LSL94], o *"Deadline (m,k) Firm"* [HaR95], *Tarefas com Duplo "Deadlines"* [GNM97] são exemplos dessas técnicas adaptativas. Essas técnicas de escalonamento adaptativo são usadas em abordagens de melhor esforço para tratar com sobrecargas em cargas dinâmicas. A figura 2.3 sintetiza as abordagens descritas nesse item.

Nesse capítulo, o estudo de escalonamento de tempo real é concentrado sobre a

abordagem de *escalonamento dirigido a prioridades.*

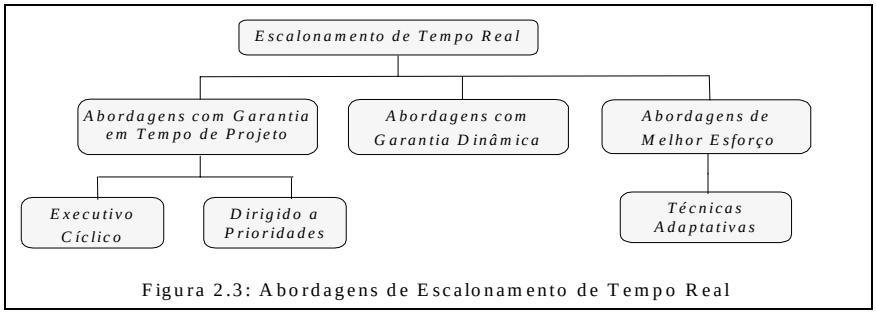


Figura 2.3: Abordagens de Escalonamento de Tempo Real

2.3.3 Teste de Escalonabilidade

Testes de escalonabilidade são importantes no processo de escalonamento de tarefas de tempo real no sentido de determinar se um conjunto de tarefas é escalonável, ou seja, se existe para esse conjunto de tarefas uma escala realizável. Esses testes variam conforme os modelos de tarefas e políticas definidas em um problema de escalonamento. Normalmente, correspondem a análises de pior caso que em certas classes de problemas, apresentam complexidade NP.

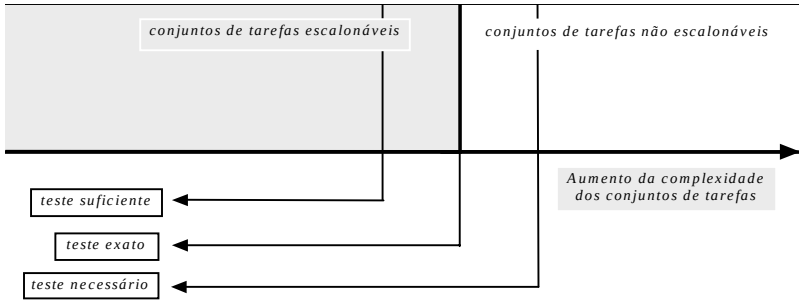


Figura 2.4 : Tipos de testes de escalonabilidade

Na literatura são identificados alguns tipos de testes [Kop92c]:

- *Testes exatos* são análises que não afastam conjuntos de tarefas que apresentam escalas realizáveis. São precisos na medida em que identificam também conjuntos não escalonáveis. Em muitos problemas são impraticáveis os testes exatos.

- *Testes suficientes* são mais simples na execução porém apresentam o custo do descarte de conjuntos de tarefas escalonáveis. É um teste mais restritivo onde

conjuntos de tarefas aceitos nesses testes certamente são escalonáveis; porém entre os descartados podem existir conjuntos escalonáveis.

- *Testes necessários* correspondem também a análises simples porém não tão restritivas. O fato de um conjunto ter passado por um teste necessário não implica que o mesmo seja escalonável. A única garantia que esse tipo de teste pode fornecer é que os conjuntos descartados de tarefas certamente são não escalonáveis.

A figura 2.4 ilustra os três tipos de testes descritos acima.

A *utilização de uma tarefa* T_i que serve como uma medida da ocupação do processador pela mesma, é dado por:

$U_i = C_i / P_i$ se a tarefa T_i é periódica, ou
$U_i = C_i / Min_i$ se a tarefa T_i é esporádica,

onde C_i, P_i e Min_i são respectivamente o tempo máximo de computação, o período e o intervalo mínimo entre requisições da tarefa T_i. A utilização de um processador (U) dá a medida da ocupação do mesmo por um conjunto de tarefas { T_1, T_2, T_3, ..., T_n}:

$$U = \sum_{i}^{n} U_i . \qquad [1]$$

O conceito de *utilização* serve de base para testes simples e bastante usados. A idéia central nesses testes é que a soma dos fatores de utilização (U_i) das tarefas do conjunto não ultrapasse o número de processadores disponíveis para suas execuções:

$$U = \sum_{i}^{n} U_i \leq m$$

onde m é o número de processadores. Testes baseados na utilização podem ser exatos, necessários ou suficientes, dependendo da política usada e do modelo de tarefas assumido [Kop92c].

2.4 Escalonamento de Tarefas Periódicas

Nesse item é tratado o problema do escalonamento de tarefas periódicas. Se considerarmos aplicações de tempo real de uma maneira geral, sejam controle de processos ou mesmo aplicações de multimídia, as atividades envolvidas nessas aplicações se caracterizam basicamente pelo comportamento periódico de suas ações. As características de tarefas periódicas que determinam o conhecimento *a priori* dos tempos de chegada e, por conseqüência, da carga computacional do sistema, permitem que se obtenha garantias em tempo de projeto sobre a escalonabilidade de um conjunto de tarefas periódicas.

O escalonamento de tarefas periódicas, nesse texto, é discutido em esquemas

dirigidos a prioridades. Nesses esquemas de escalonamento, as prioridades atribuídas às tarefas do conjunto são derivadas de suas restrições temporais, e não de atributos outros como a importância ou o grau de confiabilidade das tarefas. Escalonamentos baseados em prioridades, além de apresentarem melhor desempenho e flexibilidade, se comparados a abordagens como o executivo cíclico, são objeto de uma literatura relativamente recente e abrangente que estende os trabalhos iniciais introduzidos em [LiL73].

Neste capítulo, é feita uma revisão dos algoritmos de prioridade fixa Taxa Monotônica ("*Rate Monotonic*") [LiL73] e *"Deadline"* Monotônico ("*Deadline Monotonic*") [LeW82] e do algoritmo *"Earliest Deadline First"* [LiL73] que apresenta atribuição dinâmica de prioridades. Esses algoritmos clássicos são tidos como ótimos para suas respectivas classes de problemas, sendo caracterizados por modelos de tarefas simplificados. O conceito de utilização e os testes de escalonabilidade nesses algoritmos são apresentados como forma de análise *a priori* para determinar se existe uma escala realizável que garanta as restrições temporais impostas sobre um conjunto de tarefas periódicas. Na seqüência, após o estudo destes modelos clássicos, são aprofundados os esquemas de prioridade fixa, nas suas extensões aos trabalhos de Liu [LiL73]. Um estudo similar para esquemas de prioridade dinâmica é apresentado no *Anexo A*.

2.4.1 Escalonamento Taxa Monotônica [LiL73]

O escalonamento Taxa Monotônica ("*Rate Monotonic*") produz escalas em tempo de execução através de escalonadores preemptivos, dirigidos a prioridades. É um esquema de prioridade fixa; o que define então, o RM como escalonamento estático e *on-line* segundo os conceitos apresentados no item 2.3.1. O RM é dito ótimo entre os escalonamentos de prioridade fixa na sua classe de problema, ou seja, nenhum outro algoritmo da mesma classe pode escalonar um conjunto de tarefas que não seja escalonável pelo RM.

As premissas do RM que facilitam as análises de escalonabilidade, definem um modelo de tarefas bastante simples :

a. As tarefas são periódicas e independentes.
b. O *"deadline"* de cada tarefa coincide com o seu período ($D_i=P_i$).
c. O tempo de computação (C_i) de cada tarefa é conhecido e constante ("*Worst Case Computation Time*").
d. O tempo de chaveamento entre tarefas é assumido como nulo.

As premissas *a* e *b* são muito restritivas para o uso desse modelo na prática, contudo essas simplificações são importantes para que se tenha o entendimento sobre o escalonamento de tarefas periódicas.

A política que define a atribuição de prioridades usando o RM, determina uma

ordenação baseada nos valores de períodos das tarefas do conjunto: as prioridades decrescem em função do aumento dos períodos, ou seja, quanto mais freqüente a tarefa maior a sua prioridade no conjunto. Como os períodos das tarefas não mudam, o RM define uma atribuição estática de prioridades (prioridade fixa).

A análise de escalonabilidade no RM, feita em tempo de projeto, é baseada no cálculo da utilização. Para que *n* tarefas tenham o atendimento de suas restrições temporais quando escalonadas pelo RM, deve ser satisfeito o teste abaixo que define uma condição *suficiente*:

$$U = \sum_{i}^{n} C_i / P_i \leq n \left(2^{1/n} - 1\right). \qquad [2]$$

A medida que *n* cresce, nesse teste, a utilização do processador converge para 0,69. Uma utilização de aproximadamente 70% define uma baixa ocupação do processador que, certamente, implica no descarte de muitos conjuntos de tarefas com utilização maior e que, mesmo assim, apresentam escalas realizáveis ("*feasible*").

Essa condição suficiente pode ser relaxada quando as tarefas do conjunto apresentam períodos múltiplos do período da tarefa mais prioritária. Nesse caso a utilização alcançada sob o RM se aproxima do máximo teórico, coincidindo o teste abaixo com uma condição *necessária e suficiente* [Kop92c]:

$$U = \sum_{i}^{n} C_i / P_i \leq 1.$$

Tarefas Periódicas	***Periodo*** (P_i)	***Tempo de Computação*** (C_i)	***Prioridade RM*** (p_i)	***Utilização*** (U_i)
tarefa A	100	20	1	0,2
tarefa B	150	40	2	0,267
tarefa C	350	100	3	0,286

tabela 2.1: Utilização de tarefas periódicas

A tabela 2.1 mostra um exemplo de conjunto de tarefas periódicas onde o objetivo é verificar a escalonabilidade desse conjunto sob a política Taxa Monotônica. A utilização do processador por esse conjunto de tarefas corresponde a 0,753. Aplicando a equação [2] é concluído que esse conjunto é escalonável sob o RM:

$$0{,}753 \leq n \left(2^{1/n} - 1\right) = 0{,}779$$

A figura 2.1 apresenta – na forma de um *diagrama de Gantt* – a escala RM correspondente à tabela 2.1. As tarefas *A*, *B* e *C* chegam em *t=0*. Por ser mais freqüente (maior prioridade segundo o RM), *A* assume o processador. Em *t=20*, a tarefa *A* conclui e *B* toma posse do processador por ser a mais prioritária na fila de Pronto. A tarefa *C*

assume em *t=60* e é interrompida quando da chegada da nova ativação de *A* em *t=100* que, por sua vez, passa a se executar (preempção de *C* por *A*). *C* sofre mais interrupções de novas ativações das tarefas *B* e *A* em *t=150* e *t=200*, respectivamente.

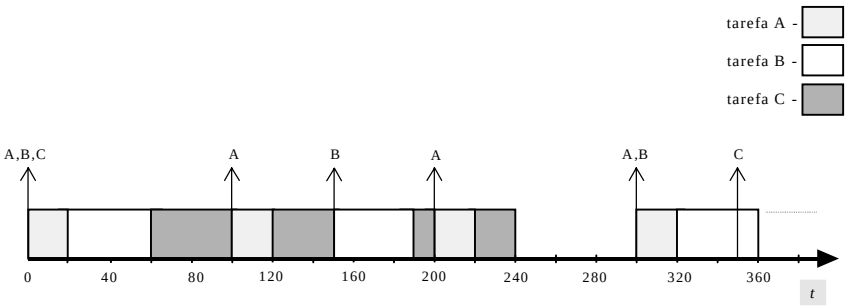


Figura 2.5: Escala RM produzida a partir da Tabela 2.1

2.4.2 Escalonamento *"Earliest Deadline First"* (EDF) [LiL73]

O *"Earliest Deadline First"* (EDF) define um escalonamento baseado em prioridades: a escala é produzida em tempo de execução por um escalonador preemptivo dirigido a prioridades. É um esquema de prioridades dinâmicas com um escalonamento *"on-line"* e dinâmico (item 2.3.1.). O EDF é um algoritmo ótimo na classe dos escalonamentos de prioridade dinâmica. As premissas que determinam o modelo de tarefas no EDF são idênticas às do RM:

a. As tarefas são periódicas e independentes.
b. O *"deadline"* de cada tarefa coincide com o seu período ($D_i=P_i$).
c. O tempo de computação (C_i) de cada tarefa é conhecido e constante ("*Worst Case Computation Time*").
d. O tempo de chaveamento entre tarefas é assumido como nulo.

A política de escalonamento no EDF corresponde a uma atribuição dinâmica de prioridades que define a ordenação das tarefas segundo os seus *"deadlines"* absolutos (d_i). A tarefa mais prioritária é a que tem o *"deadline"* d_i mais próximo do tempo atual. A cada chegada de tarefa a fila de prontos é reordenada, considerando a nova distribuição de prioridades. A cada ativação de uma tarefa T_i, seguindo o modelo de tarefas periódicas introduzido no item 2.2., um novo valor de *"deadline"* absoluto é determinado considerando o número de períodos que antecede a atual ativação (k): $d_{ik}=kP_i$.

No EDF, a escalonabilidade é também verificada em tempo de projeto, tomando como base a utilização do processador. Um conjunto de tarefas periódicas satisfazendo as premissas acima é escalonável com o EDF se e somente se :

$$U = \sum_{i=1}^{n} C_i / P_i \leq 1. \qquad [3]$$

Esse teste é *suficiente* e *necessário* na classe de problema definida para o EDF pelas premissas *a*, *b*, *c* e *d*. Se qualquer uma dessas premissas é relaxada (por exemplo, assumindo $D_i \neq P_i$), a condição [3] continua a ser *necessária* porém não é mais *suficiente*.

No exemplo mostrado na figura 2.6, o mesmo conjunto de tarefas é submetido a escalonamentos EDF e RM. A utilização do conjunto de tarefas é de 100%. Com isto pela equação [2] o conjunto não é escalonável pelo RM. Entretanto, a equação [3] garante a produção de uma escala pelo EDF. No caso (b) da figura 2.6, no tempo *t = 50* a tarefa B perde seu *"deadline"*. Além da melhor utilização, uma outra diferença, normalmente citada na literatura, é que o EDF produz menos preempções que o RM. A favor do RM está a sua simplicidade que facilita a sua implementação.

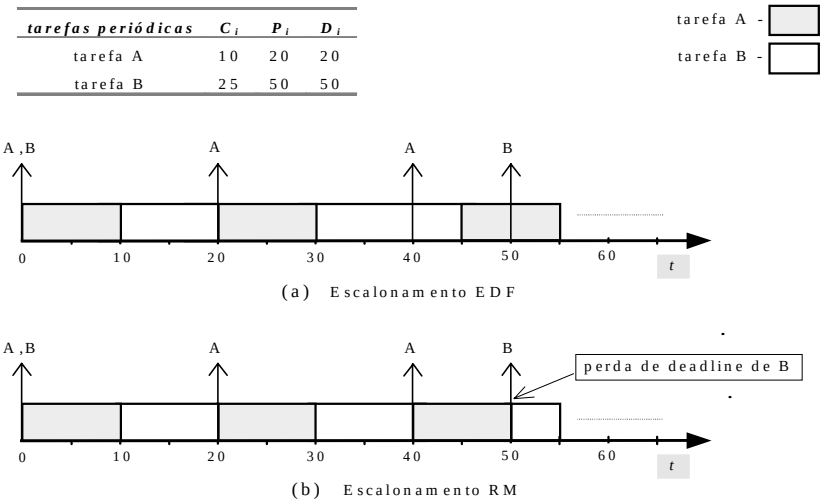


Figura 2.6: Escalas produzidas pelo (a) EDF e (b) RM

2.4.3 Escalonamento *"Deadline"* Monotônico [LeW82]

O *"Deadline Monotonic"* introduzido em [LeW82] estende o modelo de tarefas do Taxa Monotônica. A premissa do RM que limitava os valores de *"deadlines"* relativos aos valores dos períodos das tarefas ($D_i = P_i$), em muitas aplicações, pode ser considerada bastante restritiva. O modelo de tarefas do DM também define tarefas periódicas independentes, o pior caso de tempo de processamento das tarefas (C_i) e o

chaveamento entre tarefas de duração nula. Porém assume *"deadlines"* relativos menores ou iguais aos períodos das tarefas ($D_i \leq P_i$).

A política do DM define uma atribuição estática de prioridades, baseada nos *"deadlines"* relativos das tarefas (D_i). As prioridades são atribuídas na ordem inversa dos valores de seus *"deadlines"* relativos. A produção da escala, portanto, é feita em tempo de execução por escalonador preemptivo dirigido a prioridades. O esquema de prioridades fixas do DM também define um escalonamento estático e *"on-line"*. O *"Deadline"* Monotônico é também um algoritmo ótimo na sua classe de problema. A figura 2.7 mostra a escala produzida pelo DM para um conjunto de tarefas periódicas que apresentam *"deadlines"* menores que seus respectivos períodos.

tarefas periódicas	C_i	P_i	D_i	p_i
tarefa A	2	10	6	1
tarefa B	2	10	8	2
tarefa C	8	20	16	3

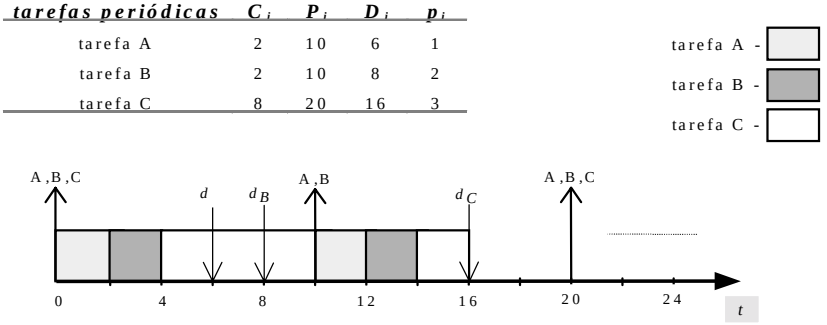


Figura 2.7: Escala produzida pelo DM

Na introdução desse algoritmo os autores não apresentaram em [LeW82] um teste correspondente. Em [ABR91] é definido um teste suficiente e necessário para o DM, baseado no conceito de *tempo de resposta* de uma tarefa. Uma evolução deste teste é descrito na seqüência, no item 2.5.2.

2.5 Testes de Escalonabilidade em Modelos Estendidos

Os testes de escalonabilidade apresentados até aqui são baseados em limites na utilização do processador. Os modelos de tarefas nesses algoritmos preemptivos, baseados em prioridades, são mantidos simples o que facilita as análises nesses testes. Qualquer extensão nos modelos permitindo, por exemplo, que tarefas possam ter relações de precedência entre si ou que os *"deadlines"* assumam valores arbitrários, define modelos mais próximos de aplicações concretas de tempo real. Porém, com essas extensões, os testes fundamentados na noção de utilização, como apresentados anteriormente, passam a ser *condições necessárias* e novos testes mais restritivos são desejáveis.

São muitas as propostas na literatura de extensões dos modelos de [LiL73]. Estes modelos apresentados no item 2.4 podem ser estendidos, por exemplo, para incluírem também tarefas esporádicas. Nos testes que se seguem podemos interpretar o valor P_i também como o mínimo intervalo entre requisições (min_i) de uma tarefa esporádica T_i [ABR91], [TBW94]. Garantir uma tarefa esporádica na sua maior freqüência implica em ter ativações menos freqüentes também respeitando suas restrições temporais.

Como visto anteriormente, a análise (ou teste) de escalonabilidade tenta responder as questões sobre o atendimento dos requisitos de correção temporal de um conjunto de tarefas tempo real quando uma determinada política de escalonamento, e portanto uma atribuição de prioridades é feita. Nessas análises são considerados cenários de pior caso. Além de considerarem as tarefas executando com os seus piores casos de tempo de computação (C_i) e com as suas maiores freqüências de ativação (no caso de tarefas esporádicas), esses testes são construídos levando em conta o *Instante Crítico*. O pior caso de ocorrência de tarefas na fila de Pronto do processador é identificado como instante crítico, ou seja, é o instante onde todas as tarefas do sistema estão prontas para a ocupação do processador. Se nesses cenários de pior caso um conjunto de tarefas consegue recursos suficientes para atender suas restrições, em situações mais favoráveis certamente também atenderá.

Na literatura, tanto para escalonamentos com prioridades fixas como para prioridades dinâmicas são descritos testes de escalonabilidade exatos ou suficientes, que exploram modelos de tarefas mais complexos. Esses testes podem ainda ser construídos usando a noção de *utilização* ou estarem baseados em conceitos como *tempo de resposta* e *demanda de processador*. Nesse item exploramos alguns testes para políticas de prioridade fixa. Os testes referentes a políticas de prioridade dinâmica seguem o mesmo estilo de apresentação e, foram colocados no *Anexo A* no sentido de simplificar a apresentação deste capítulo.

2.5.1 *"Deadline"* Igual ao Período

Ainda não estendendo o modelo do algoritmo Taxa Monotônica (*"Rate Monotonic"*), mas tentando apresentar um teste com melhor desempenho do que o original, em [LSD89] é apresentado um teste onde a utilização do processador não tem mais um sentido estático, dependente somente das restrições temporais das tarefas, mas é também função de uma janela de tempo *t* considerada no seu cálculo. As tarefas de prioridade maior ou igual a *i* que estão disponíveis para execução no processador no intervalo *t* constituem a carga cumulativa (*"workload"*) $W_i(t)$ deste intervalo e é dada por:

$$W_i(t) = \sum_{j=1}^{i} \left\lceil \frac{t}{P_j} \right\rceil . C_j , \qquad [4]$$

onde $\lceil t/P_j \rceil$ corresponde ao máximo número de ocorrências de uma tarefa T_j no intervalo t e $\lceil t/P_j \rceil . C_j$ define as necessidades de processador dessa mesma tarefa no intervalo considerado. A divisão do *"workload"* $W_i(t)$ pelo valor de t dá a percentagem de utilização do processador nesse intervalo de tempo t, considerando apenas tarefas de prioridade maior ou igual a Ti :

$$U_i(t) = \frac{W_i(t)}{t}. \qquad [5]$$

A condição necessária e suficiente para que a tarefa Ti seja escalonável é que exista um valor de t em que a sua utilização $U_i(t)$ seja menor ou igual a 1, isto é, que a carga de trabalho $W_i(t)$ não supere o intervalo de tempo t ($W_i(t) \leq t$) [LSD89]. A dificuldade deste teste está em se determinar este valor de t.

Como o modelo define tarefas periódicas com instante crítico ocorrendo em $t = 0$, se cada tarefa T_i respeitar o seu primeiro *"deadline"* então os *"deadlines"* subseqüentes, referentes a suas outras ativações, também serão atendidos. Com isto, os valores de t podem se limitar ao intervalo $(0,P_i]$. Neste caso a procura de t pode se resumir a testar os valores de mínimo de $U_i(t)$ no intervalo considerado:

$$\forall i, \min_{0<t\leq Pi} U_i(t) \leq 1,$$

No lugar de fazer uma procura entre todos os valores de t no intervalo $(0, P_i]$, os cálculos podem se resumir a testar os valores de :

$$S_i = \left\{ kP_j \;\middle|\; 1 \leq j \leq i, \quad k = 1,2,\ldots,\left\lfloor \frac{P_i}{P_j} \right\rfloor \right\}. \qquad [6]$$

Os pontos descritos em S_i correspondem aos tempos de chegada das ativações das tarefas T_j de prioridade maior ou igual a T_i, ocorrendo dentro do período P_i. Estes pontos correspondem aos mínimos de $U_i(t)$, uma vez que, esta função é monotônica decrescente nos intervalos entre chegadas de tarefas T_j. Logo, um conjunto de tarefas periódicas independentes, com instante crítico em $t=0$ e escalonadas segundo o RM, terá seus *"deadlines"* respeitados se :

$$\forall i, \min_{t \in Si} U_i(t) \leq 1. \qquad [7]$$

O teste acima quando comparado com [2], embora mais complexo, pode aprovar conjuntos de tarefas que seriam rejeitados pelo teste suficiente proposto originalmente para o RM. O teste composto a partir de [5] e [7] permite uma maior utilização do processador, formando uma condição suficiente e necessária [LSD89], [Fid98].

Como exemplo, considere o conjunto de tarefas da figura 2.6 (tabela 2.2) em que ocorre na escala do RM, a perda do *"deadline"* da tarefa T_2 em $t=50$. Para a verificação

das condições de escalonabilidade do conjunto de tarefas usamos o teste formado por [5] e [7].

tarefas periódicas	C_i	P_i	D_i
tarefa T_1	10	20	20
tarefa T_2	25	50	50

Tabela 2.2: Exemplo da figura 2.6

Nessa verificação, inicialmente temos que calcular as utilizações $U_i(t)$ para todas as tarefas do conjunto. Começamos então calculando a carga de trabalho correspondente de cada tarefa (equação [4]):

$$W_1(t) = \left\lceil \frac{t}{20} \right\rceil .10 \qquad e \qquad W_2(t) = \left\lceil \frac{t}{50} \right\rceil .25 + \left\lceil \frac{t}{20} \right\rceil .10,$$

determinamos as utilizações (equação [5]):

$$U_1(t) = \frac{W_1(t)}{t} \quad e \quad U_2 = \frac{W_2(t)}{t}.$$

A dificuldade é fazer a inspeção na procura de valores de t que determinam $U_i(t) \bullet 1$. Para a tarefa T_1 esses valores são obtidos a partir de S_1 ([6]): $S_1 = \{20\}$. Então, para o valor de $t=20$, a utilização $U_1(20)$ é determinada como sendo de 0,5. Logo, pela condição [7], a tarefa T_1 é escalonável.

O conjunto referente a T_2, usando também [6], é formado por: $S_2 = \{20, 40, 50\}$. A partir deste conjunto é determinada $U_2(t)$:

$$t = 20 \Rightarrow U_2(20) = 1{,}75$$

$$t = 40 \Rightarrow U_2(40) = 1{,}12$$

$$t = 50 \Rightarrow U_2(50) = 1{,}1.$$

O máximo valor de utilização $U_2(t)$ ocorre em $t = 20$ (o menor valor de t em S_2) e, usando a condição [7], é determinado que o conjunto de tarefas da tabela 2.2 é de fato não escalonável pelo RM.

2.5.2 *"Deadline"* Menor que o Período

O modelo de tarefas, no teste que se segue, relaxa em escalonamentos de prioridade fixa a condição que todas as tarefas periódicas independentes devem ter seus

"deadlines" relativos iguais aos seus respectivos períodos. O modelo assume também tarefas com *"deadlines"* menores aos seus períodos ($D_i \leq P_i$). Esse teste, diferentemente dos anteriores, está fundamentado no conceito de *tempo de resposta*. Em [JoP86], *tempo de resposta máximo* de uma tarefa foi introduzido como sendo o tempo transcorrido entre a chegada e o término de sua execução, considerando a máxima interferência que a tarefa pode sofrer de outras tarefas de maior ou igual prioridade. Uma análise de escalonabilidade que se utilize desse conceito deve representar então o pior caso de interferências (em termos de tempo) sobre cada tarefa do sistema.

Para o cálculo do tempo de resposta máximo de uma tarefa é necessário que se defina uma janela de tempo R_i que corresponda ao intervalo de tempo máximo transcorrido da liberação de uma tarefa T_i até o término de sua execução. A largura R_i corresponde ao tempo necessário, em situação de *instante crítico*, para a execução de T_i e de todas as tarefas com prioridades maiores ou iguais a i (p_i=i). Nestas condições, o tempo de resposta máximo R_i da tarefa T_i é dado por :

$$R_i = C_i + \sum_{j \in hp(i)} I_j$$

onde *hp(i)* é o conjunto de prioridades maior que i e, I_j é a interferência que a tarefa T_i pode sofrer de uma tarefa T_j de prioridade maior, durante a largura R_i. A interferência I_j é calculada por:

$$I_j = \left\lceil \frac{R_i}{P_j} \right\rceil \cdot C_j$$

onde $\lceil R_i / P_j \rceil$ representa o número de liberações de T_j em R_i. A expressão do tempo de resposta R_i pode ser reescrita como:

$$R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{P_j} \right\rceil \cdot C_j \qquad [8]$$

Uma tarefa T_i mantém suas propriedades temporais sempre que $R_i \leq D_i$. Nesta verificação é necessário a determinação de R_i a partir de [8]. Porém a largura R_i aparecendo em ambos os lados dessa equação, implica na necessidade de um método iterativo para essa determinação. Em [JoP86] é apresentado um método para solucionar [8]:

$$R_i^{n+1} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i^n}{P_j} \right\rceil \cdot C_j$$

onde a solução é conseguida a partir do cálculo de R_i^n, a $n^{ésima}$ aproximação de R_i, quando $R_i^{n+1} = R_i^n$. Nesse método iterativo é assumido, como condição de partida, $R_i^0 = C_i$. O método não converge quando a utilização do conjunto de tarefas for maior que 100%.

O teste fundamentado no conceito de tempo de resposta determina então um

conjunto de *n* tarefas de tempo real como escalonável sempre que a condição $\forall i: 1 \leq i \leq n$ $R_i \leq D_i$ for verificada. Os cálculos dos tempos de respostas formam a base para um teste suficiente e necessário.

tarefas periódicas	C_i	P_i	D_i
tarefa A	2	10	6
tarefa B	2	10	8
tarefa C	8	20	16

Tabela 2.3: Exemplo da figura 2.7

Para clarificar o uso desse teste de escalonabilidade, considere o exemplo que foi apresentado na figura 2.7 onde era apresentada uma escala produzida pelo *"Deadline Monotonic"*. As características das tarefas são reproduzidas na tabela 2.3.

A política DM pelos valores da tabela define uma atribuição de prioridades onde $p_A > p_B > p_C$. Os tempos de resposta das tarefas consideradas são determinados a partir da aplicação da equação [8] nos valores da tabela 2.3. A tarefa T_A, por ser a mais prioritária, não sofre interferência das demais e o seu tempo de resposta é dado por $R_A = C_A = 2$. T_A é escalonável porque seu tempo de resposta máximo é menor que seu *"deadline"* relativo ($D_A = 6$) O cálculo de R_B, ao contrário, envolve mais passos devido a interferência que T_B sofre de T_A. Aplicando [8] é obtido:

$$R_B^0 = C_B = 2$$
$$R_B^1 = 2 + \left\lceil \frac{2}{10} \right\rceil .2 = 4$$
$$R_B^2 = 2 + \left\lceil \frac{4}{10} \right\rceil .2 = 4$$

A tarefa T_B que apresenta $R_B = 4$ é também escalonável ($R_B \leq D_B$). O tempo R_C, por sua vez, envolve as interferências de T_A e T_B em T_C. A partir de [8]:

$$R_C^0 = C_C = 8$$
$$R_C^1 = 8 + \left\lceil \frac{8}{10} \right\rceil .2 + \left\lceil \frac{8}{10} \right\rceil .2 = 12$$
$$R_C^2 = 8 + \left\lceil \frac{12}{10} \right\rceil .2 + \left\lceil \frac{12}{10} \right\rceil .2 = 16$$
$$R_C^3 = 8 + \left\lceil \frac{16}{10} \right\rceil .2 + \left\lceil \frac{16}{10} \right\rceil .2 = 16$$

A tarefa T_C é também escalonável apresentando um tempo de resposta (16 unidades de tempo) no limite máximo para o seu *"deadline"* relativo ($R_C = D_C$). Os resultados obtidos confirmaram a escala obtida pelo DM para o conjunto de tarefas da figura 2.7.

Nos modelos apresentados até aqui as tarefas são assumidas como periódicas e liberadas sempre no início de cada período. Contudo isto nem sempre corresponde a uma hipótese realista. Escalonadores ativados por tempo (*"ticket scheduler"* [TBW94]) podem ser fonte do atraso na liberação de tarefas: as tarefas tem as suas chegadas detectadas nos tempos de ativação do escalonador, determinando atrasos nas suas liberações. Esses atrasos podem ser expressados no pior caso como *"release jitters"* (*J*).

Para determinar as modificações necessárias no sentido de representar no cálculo dos tempos de resposta as variações na liberação das tarefas, é necessário que se introduza o conceito de *período ocupado* ("*busy period*"). Um período ocupado *i* corresponde a uma janela de tempo W_i onde ocorre a execução contínua de tarefas com prioridade maior ou igual a *i* ($p_i = i$). O *"i-busy period"* associado com a tarefa T_i começa quando da liberação da instância de T_i e parte do pressuposto que todas as tarefas de prioridades maior que *i* estão na fila de pronto.

Se considerarmos a janela W_i, o limite máximo das ocorrências de T_j nesse intervalo é dado por $\lceil W_i / P_j \rceil$. Porém ao se assumir que uma instância de T_j, anterior ao início de W_i, experimenta um atraso máximo J_j na sua liberação determinando a interferência dessa instância sobre T_i associada com *Wi*, o número de ativações de T_j que interferem com T_i passa a ser $\lceil (W_i + J_j) / P_j \rceil$. Nessas condições o cálculo de W_i é dado por [TBW94] :

$$W_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{W_i + J_j}{P_j} \right\rceil \cdot C_j \qquad [9]$$

onde a solução de [9] é conseguida pelo mesmo método iterativo usado para resolver [8]. *Wi* é o intervalo entre a liberação e o termino de T_i. Para o cálculo do tempo de resposta máximo (R_i), correspondendo ao intervalo de tempo entre a chegada e o término da instância da tarefa T_i, é necessário que se considere também o atraso máximo experimentado por T_i na sua liberação:

$$R_i = W_i + J_i . \qquad [10]$$

O teste de um conjunto de tarefas experimentando atrasos em suas liberações leva então em consideração [9], [10] e a verificação da condição $\forall i: 1 \le i \le n \; R_i \le D_i$.

2.5.3 Deadline Arbitrário

O teste baseado em tempos de resposta calculados a partir de [9] e [10] é exato quando aplicado em modelos com $D_i \le P_i$. Porém em modelos de *"deadlines"* arbitrários onde *"deadlines"* podem assumir valores maiores que os respectivos períodos ($D_i > P_i$), esse teste deixa de ser uma condição suficiente [TBW94]. Tarefas que apresentam seus *"deadlines"* maiores que o seus períodos, sofrem o que se pode chamar de

interferências internas, ou seja, uma vez que se pode ter $D_i > P_i$, ocorrências anteriores de uma mesma tarefa podem interferir numa ativação posterior.

É necessário que se estenda o conceito de *"busy period"* para se determinar o pior caso de interferência em uma tarefa T_i onde suas ativações podem se sobrepor. O *"i-busy period*" não começa mais com a liberação da instância de T_i que se deseja calcular o tempo de resposta. O período ocupado *i* continua sendo a janela de tempo W_i que inclui essa instância de T_i e que corresponde a maior execução contínua de tarefas com prioridade maior ou igual a *i* ($p_i = i$). Porém, o início desse período de execução contínua de tarefas se dá com a liberação de outra ativação anterior de T_i. Nessa extensão é assumido que a execução de qualquer liberação de uma tarefa será atrasada até que liberações anteriores da mesma tarefa sejam executadas completamente.

Considera-se então o *"i-busy period"* incluindo *q+1* ativações de T_i que podem se sobrepor na concorrência ao processador pois $D_i > P_i$. O valor da janela de tempo $W_i(q)$ correspondente é dado por [TBW94]:

$$W_i(q) = (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{W_i(q)}{P_j} \right\rceil C_j . \qquad [11]$$

O valor $q.C_i$ em [11] representa a interferência interna que a última instância de T_i sofre em $W_i(q)$. Por exemplo, se *q=2* então T_i terá três liberações em $W_i(q)$. O tempo de resposta da *(q+1)ésima* ativação de T_i é dada por:

$$R_i(q) = W_i(q) - qP_i \qquad [12]$$

ou seja, o tempo de resposta é dado considerando o desconto do número de períodos de T_i em $W_i(q)$, anteriores a $(q+1)^{ésima}$ ativação de T_i. Para que se obtenha o *tempo de resposta máximo* R_i da tarefa T_i é necessário que se faça a inspeção de todos *"i-busy periods"*, na ordem de valores crescentes de *q* (*q=0, 1, 2, 3,..*), até um valor de janela que verifique a condição:

$$W_i(q) \le (q+1)P_i .$$

Esta janela corresponde ao maior valor de *"i-busy period"* que se consegue, antes da execução de uma tarefa menos prioritária que *i* [TBW94]. Nessas condições, o tempo de resposta máximo R_i é calculado por:

$$R_i = \max_{q=0,1,2,\ldots} R_i(q).$$

Se o efeito de *"deadlines"* arbitrários for incluído em modelos onde as liberações não coincidem com os tempos de chegada das instâncias da tarefa T_i, a janela de tempo $W_i(q)$ e o tempo de resposta R_i passam a ser dados, respectivamente, por:

$$W_i(q) = (q+1)C_i + \sum_{j \in hp(i)} \left\lceil \frac{W_i(q) + J_j}{P_j} \right\rceil C_j \qquad [13]$$

$$e \qquad R_i = \max_{q=0,1,2,\ldots} (J_i + W_i(q) - qP_i). \qquad [14]$$

O teste de um conjunto de tarefas com *"deadlines"* arbitrários e experimentando atrasos em suas liberações leva então em consideração [13], [14] e a verificação da condição $\forall i: 1 \leq i \leq n \; R_i \leq D_i$. Este teste funciona como uma análise *a priori* (em tempo de projeto) válida para qualquer política de prioridade fixa. A solução iterativa da equação [13] tem uma complexidade pseudo-polinomial que para propósitos práticos pode ser assumida como polinomial [TBW94].

tarefas periódicas	J_i	C_i	P_i	D_i
tarefa T_1	1	10	40	40
tarefa T_2	3	10	80	25
tarefa T_3	-	5	20	40

Tabela 2.4: Tarefas com Deadlines Arbitrários

Como um exemplo de aplicação deste teste para modelos com *"deadlines"* arbitrários, considere o conjunto de tarefas dado pela tabela 2.4. Suponha que queremos determinar o tempo de resposta máximo da tarefa T_3. Considerando uma atribuição de prioridades onde $p_1 > p_2 > p_3$. A tarefa T_3 sofre interferência das outras duas e pela equação [13], obtemos:

$$W_3(q) = (q+1).5 + \left\lceil \frac{W_3(q) + 1}{40} \right\rceil .10 + \left\lceil \frac{W_3(q) + 3}{80} \right\rceil .10$$

Para que se obtenha o *tempo de resposta máximo* R_3 da tarefa T_3 é necessário que se faça a inspeção de todos *"3-busy periods"*, na ordem de valores crescentes de q ($q=0, 1, 2, 3,..$), até que o valor de janela seja limitado por: $W_3(q) \bullet (q+1).P_3$. Então, para $q = 0$ e usando a equação [13], é obtido:

$$W_3^0(0) = C_3 = 5$$

$$W_3^1(0) = 5 + \left\lceil \frac{5+1}{40} \right\rceil .10 + \left\lceil \frac{5+3}{80} \right\rceil .10 = 25$$

$$W_3^2(0) = 5 + \left\lceil \frac{25+1}{40} \right\rceil .10 + \left\lceil \frac{25+3}{80} \right\rceil .10 = 25$$

O valor do tempo de resposta $R_3(0)$ é dado por [12] e corresponde a 25. Com $W_3(0)$ superando o período P_3 ($P_3 = 20$) então, essa janela não corresponde ao maior *"busy*

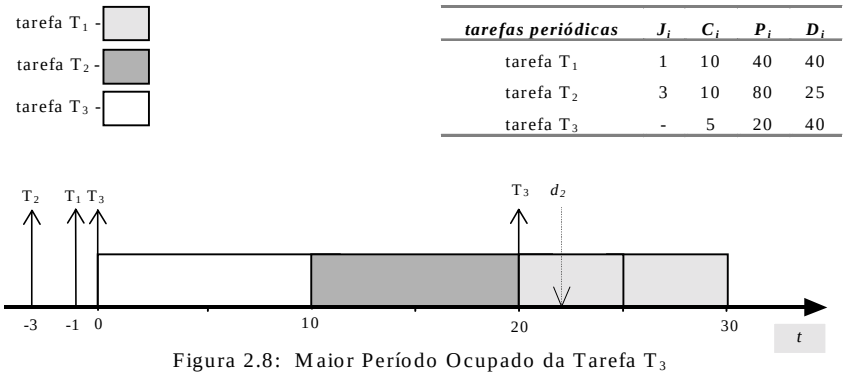
period" de prioridade 3. Assumindo então *q =1*:

$$W_3^0(1) = C_3 = 5$$

$$W_3^1(1) = 10 + \left\lceil \frac{5+1}{40} \right\rceil .10 + \left\lceil \frac{5+3}{80} \right\rceil .10 = 30$$

$$W_3^2(1) = 10 + \left\lceil \frac{30+1}{40} \right\rceil .10 + \left\lceil \frac{30+3}{80} \right\rceil .10 = 30$$

Com isto obtemos pela equação [12] $R_3(1) = W_3(1) - P_3 = 10$. Como $W_3(1) < 2P_3$, temos então o máximo *"3-busy period"* envolvendo duas ativações da tarefa T_3. Logo o tempo de resposta máximo da tarefa T_3 é o maior dos tempos de resposta obtidos, ou seja, é de valor 25, correspondendo a primeira ativação de T_3 no maior *"3-busy period"*. A figura 2.8 mostra esse período ocupado de prioridade 3 onde T_3 é liberada em suas duas ativações em $t = 0$ e $t = 20$. Por sua vez, devido aos seus *"releases jitters"*, as tarefas T_1 e T_2 que chegam em –1 e –3, respectivamente, só são liberadas em $t = 0$. A primeira ativação de T_3 é empurrada para fora de seu período interferindo com a ativação seguinte da mesma tarefa.



tarefas periódicas	J_i	C_i	P_i	D_i
tarefa T_1	1	10	40	40
tarefa T_2	3	10	80	25
tarefa T_3	-	5	20	40

Figura 2.8: Maior Período Ocupado da Tarefa T_3

2.6 Tarefas Dependentes:Compartilhamento de Recursos

Nos modelos discutidos até a presente seção, as tarefas eram apresentadas como independentes o que, se considerarmos a grande maioria de aplicações de tempo real, não corresponde a uma premissa razoável. Em um ambiente multitarefas o compartilhamento de recursos é implícito e determina alguma forma de relação de exclusão entre tarefas. Comunicações entre tarefas residindo no mesmo processador, por exemplo, podem se dar através de variáveis compartilhadas, usando mecanismos como semáforos, monitores ou similares para implementar a exclusão mútua entre as

tarefas comunicantes.

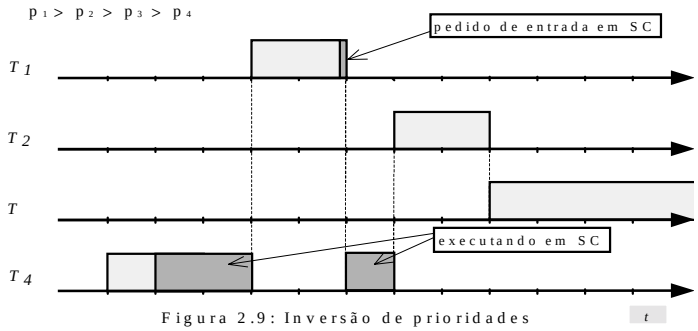


Figura 2.9: Inversão de prioridades

O compartilhamento de recursos e as relações de exclusão decorrentes do mesmo, determinam bloqueios em tarefas mais prioritárias. Esses bloqueios são identificados na literatura de tempo real como *inversões de prioridades*. Considere o cenário da figura 2.9, formado pelas tarefas periódicas T_1, T_2, T_3 e T_4, apresentadas na ordem crescente de seus períodos. Uma escala é construída sobre o conjunto de tarefas baseada no algoritmo *"Rate Monotonic"*. T_1 e T_4 compartilham um recurso guardado por um mecanismo de exclusão mútua. Na escala da figura 2.9, o bloqueio que T_1 sofre pelo acesso anterior de T_4 ao recurso compartilhado caracteriza um exemplo de *inversão de prioridade*: mesmo liberada a tarefa T_1 não consegue evoluir devido ao bloqueio. A tarefa T_1, durante o bloqueio, sofre também interferências de T_2 e de T_3. Esse fato ocorre porque T_4 é a tarefa menos prioritária do conjunto e sofre preempções dessas tarefas intermediárias. As preempções de T_2 e de T_3 sobre a tarefa T_4 podem caracterizar um bloqueio de T_1 com duração de difícil determinação.

Quando as tarefas se apresentam como dependentes, a inversão de prioridades é inevitável em um escalonamento dirigido a prioridades. O que seria desejável é que as inversões de prioridades que eventualmente possam ocorrer nas escalas produzidas sejam limitadas. É nesse contexto que alguns métodos para controlar o acesso em recursos compartilhados foram introduzidos. Esses métodos impõem certas regras no compartilhamento dos recursos de modo que o pior caso de bloqueio experimentado por uma tarefa no acesso a uma variável compartilhada possa sempre ser conhecido *a priori*.

Nesse item examinamos duas destas técnicas: o *Protocolo Herança de Prioridade* e o Protocolo de Prioridade Teto (*"Priority Ceiling Protocol"*) desenvolvidos para esquemas de prioridades fixas [SRL90]. A técnica Política de Pilha ("*Stack Resource Policy*" [Bak91]) própria para escalonamentos de prioridades dinâmicas é apresentada no *anexo A*.

2.6.1 Protocolo Herança de Prioridade

Uma solução simples para o problema de inversões prioridades não limitadas seria ter desabilitada a preempção quando tarefas entrassem em seções críticas [AuB90]. Esse método de escalonamento híbrido (preemptivo e não preemptivo) evita as interferências de tarefas intermediárias, porém, é bastante penalizante com tarefas mais prioritárias que não usam os recursos compartilhados, quando as seções críticas envolvidas não são pequenas.

O método de *Herança de Prioridade* apresentado por [SRL90] corresponde a uma solução eficiente para tratar o problema de inversões de prioridades provocadas pelas relações de exclusão. Nesse protocolo as prioridades deixam de ser estáticas; toda vez que uma tarefa menos prioritária bloqueia uma de mais alta prioridade em um recurso compartilhado, a menos prioritária ascende à prioridade da tarefa bloqueada mais prioritária.

- **Descrição do Protocolo**

O uso do Protocolo Herança de Prioridade (PHP) determina que as tarefas sejam definidas possuindo uma prioridade *nominal ou estática*, atribuída por alguma política de prioridade fixa (RM, DM, etc.) e uma prioridade *dinâmica ou ativa* derivada das ações de bloqueio que ocorrem no sistema. Inicialmente, numa situação sem bloqueio no sistema, todas as tarefas apresentam suas prioridades estáticas coincidindo com suas prioridades ativas. As tarefas são escalonadas tomando como base suas prioridades ativas.

Quando uma tarefa T_i é bloqueada em um semáforo, sua prioridade dinâmica ou ativa é transferida para a tarefa T_j que mantém o recurso bloqueado. Quando reassume, T_j executa o resto de sua seção crítica com a prioridade herdada de T_i ($p_j = p_i$). Uma tarefa *herdada*, ao se executar, sempre a mais alta das prioridades das tarefas que mantenha sob bloqueio. No momento em que T_j ao completar sua seção crítica, libera o semáforo associado, a sua prioridade ativa retorna à prioridade nominal ou assume a mais alta prioridade das tarefas que ainda estejam sob seu bloqueio.

A aplicação do Protocolo Herança de Prioridade no exemplo anterior (da figura 2.9) implica que T_4 não mais sofrerá de interferências intermediárias (preempções de T_2 e T_3) porque herda a prioridade de T_1 em *t=5* (figura 2.10). E, ao sair da sua seção crítica (em *t=7*, figura 2.10), T_4 volta ao nível de sua prioridade original.

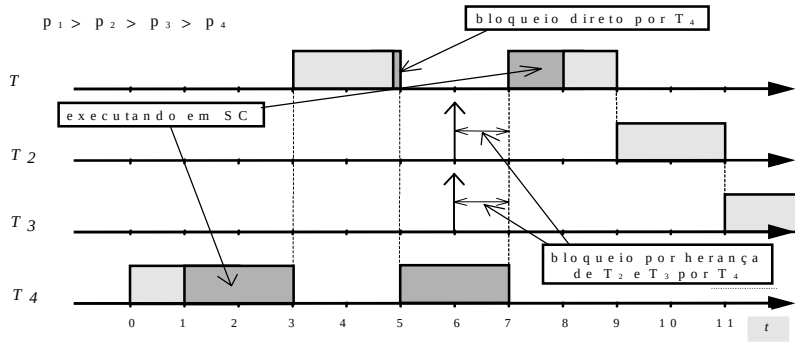


Figura 2.10: Exemplo do uso do PHP

Uma tarefa mais prioritária quando executando sob o PHP pode sofrer dois tipos de bloqueios [But97]:

- *Bloqueio direto*: que ocorre quando a tarefa mais prioritária tenta acessar o recurso compartilhado já bloqueado pela tarefa menos prioritária.

- *Bloqueio por herança*: ocorre quando uma tarefa de prioridade intermediária é impedida de continuar sua execução por uma tarefa que tenha herdado a prioridade de uma tarefa mais prioritária.

No exemplo da figura 2.10, as tarefas T_2 e T_3 sofrem bloqueios por herança em $t = 6$ (T_2 e T_3 chegam em $t = 6$), e T_1 está sujeita a um bloqueio direto em $t = 5$.

O PHP define um limite superior para o número de bloqueios que uma tarefa pode sofrer de outras menos prioritárias. Se uma tarefa T_i pode ser bloqueada por *n* tarefas menos prioritárias, isto significa que, em uma ativação, T_i pode ser bloqueada por *n* seções críticas, uma por cada tarefa menos prioritária. Por outro lado, se houverem *m* distintos semáforos (recursos compartilhados) que podem bloquear diretamente T_i, então essa tarefa pode ser bloqueada no máximo a duração de tempo correspondente às *m* seções críticas, sendo uma por cada semáforo. Em [SRL90] é então assumido que sob o Protocolo Herança de Prioridade, uma tarefa T_i pode ser bloqueada no máximo a duração de *min (n, m)* seções críticas.

A ocorrência de seções críticas aninhadas permite o surgimento de um terceiro tipo de bloqueio: o *transitivo*. A figura 2.11, mostra quatro tarefas (T_1, T_2, T_3 e T_4) onde T_2 e T_3 possuem seções aninhadas. T_1 é mostrada bloqueada por T_2; por sua vez, a tarefa T_2 é bloqueada por T_3 e, por fim, T_3 é bloqueada por T_4. Nessa cadeia de bloqueios, a tarefa T_1 sofre um bloqueio indireto ou transitivo de T_4. A tarefa T_1 só retoma o seu processamento quando houver a liberação, na seqüência, das seções críticas de T_4, T_3 e T_2, respectivamente. Bloqueios transitivos portanto, criam a possibilidade de que se formem cadeias de bloqueios que podem levar até mesmo a situações de *deadlocks*.

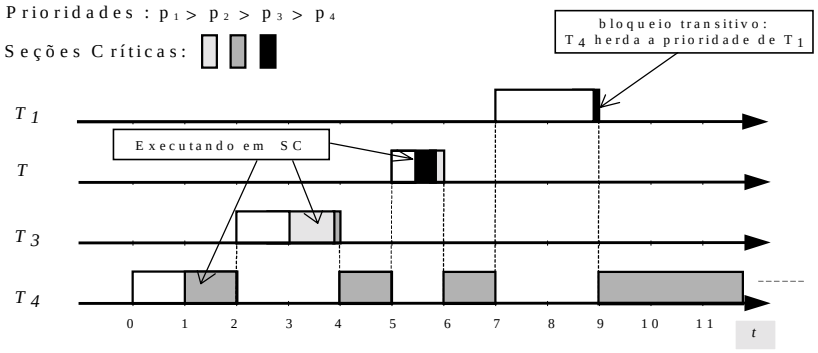


Figura 2.11: Bloqueio transitivo

- **Extensões de Testes de Escalonabilidade Tomando como Base o PHP**

Uma determinação precisa do valor de bloqueio máximo B_i que uma tarefa T_i pode sofrer quando do uso do PHP é certamente bem difícil, uma vez que, seções críticas de tarefas menos prioritárias podem interferir com T_i através de diferentes tipos de bloqueios. Dependendo da complexidade do modelo de tarefas, fica impraticável a determinação precisa de B_i. Alguns autores apresentam métodos para estimativas desse tempo de bloqueio ([BuW97], [But97] e [Raj91]). Um cálculo mais preciso de B_i envolve procuras exaustivas que considerando a complexidade do conjunto de tarefas pode ser impraticável.

O limite imposto pelo PHP no bloqueio máximo (B_i) que uma tarefa T_i pode sofrer de tarefas menos prioritárias, tem que se refletir nas análises de escalonabilidade de esquemas baseados em prioridades fixas. Em [SRL90] e [SSL89] o teste do RM (equação [**2**]) é estendido no sentido de incorporar as relações de exclusão de um conjunto de tarefas:

$$\left(\sum_{j=1}^{i} \frac{C_j}{P_j}\right) + \frac{B_i}{P_i} \leq i\,(2^{1/i} - 1), \quad \forall i. \qquad [15]$$

O somatório do teste acima considera a utilização de tarefas com prioridade maior ou igual a p_i e o termo B_i / P_i corresponde à utilização perdida no bloqueio de T_i por tarefas menos prioritárias. Para que um conjunto de *n* tarefas seja considerado escalonável pelo *"Rate Monotonic"*, é necessário que as *n* condições geradas a partir desse teste sejam verificadas.

tarefas	C_i	P_i	B_i
T_1	6	18	2
T_2	4	20	4
T_3	10	50	0

Tabela 2.5

A tabela 2.5 apresenta um conjunto de tarefas periódicas com seus respectivos bloqueios máximos quando executadas sob o PHP. O uso do teste [15] nesse conjunto de tarefas implica nas relações abaixo:

$$\frac{C_1}{P_1} + \frac{B_1}{P_1} \leq 1$$
$$\frac{C_1}{P_1} + \frac{C_2}{P_2} + \frac{B_2}{P_2} \leq 0{,}82$$
$$\frac{C_1}{P_1} + \frac{C_2}{P_2} + \frac{C_3}{P_3} \leq 0{,}78$$

Todas essas relações acima se verificam para os valores de indicados na tabela 2.5; o que indica que o conjunto é escalonável e todas as tarefas se executarão dentro de seus *"deadlines"*.

Uma outra variante desse teste onde a escalonabilidade pode ser verificada apenas por uma equação só, é também apresentado em [SRL90]:

$$\sum_{i=1}^{n} \frac{C_i}{P_i} + max\left(\frac{B_1}{P_1}, \ldots, \frac{B_n}{P_n}\right) \leq n.\left(2^{\frac{1}{n}} - 1\right). \qquad [16]$$

O novo teste é mais simples que o anterior porém é mais restritivo e menos preciso. Como exemplo, considere o uso do teste [16] na verificação da escalonabilidade do mesmo conjunto de tarefas da tabela 2.5. A equação [16] aplicada às condições desse mesmo conjunto implica em:

$$\frac{C_1}{P_1} + \frac{C_2}{P_2} + \frac{C_3}{P_3} + max\left(\frac{B_1}{P_1}, \frac{B_2}{P_2}\right) \leq 3\left(2^{\frac{1}{3}} - 1\right)$$

onde, se substituirmos os valores da tabela 2.5, chegaremos a conclusão que o conjunto é não escalonável. Como esse teste é mais restritivo, todo o conjunto descartado em relação ao teste [16] deve ser verificado com o teste [15] no sentido de confirmar o descarte. Porém o conjunto que passar pelo teste [16] certamente é escalonável.

Os testes para políticas de prioridades fixas, apresentados na seção 2.5, podem ser facilmente estendidos no sentido de incluir os bloqueios que sofrem cada tarefa no conjunto. O teste proposto em [LSD89] baseado em utilização, na sua versão estendida

toma a seguinte forma, ([Fid98]) :

$$U_i(t) = \frac{\sum_{j=1}^{i} \left\lceil \frac{t}{P_j} \right\rceil C_j + B_i}{t}, \quad \forall i, \ min_{0<t\leq P_i} U_i(t) \leq 1. \qquad [17]$$

O bloqueio B_i nessa equação é também apresentado como utilização perdida no bloqueio de T_i por tarefas menos prioritárias. O teste baseado em tempo de resposta para modelos envolvendo *"deadlines"* arbitrários apresentado em [TBW94] é também facilmente estendido:

$$W_i(q) = (q+1)C_i + B_i + \sum_{j \in hp(i)} \left\lceil \frac{W_i(q) + J_j}{P_j} \right\rceil C_j. \qquad [18]$$

O bloqueio B_i é apresentado na equação [18] como uma interferência sofrida por T_i. Todos os testes apresentados com as extensões referentes ao limite máximo de bloqueio definido sob o PHP, deixam de ser exatos e passam a ser condições suficientes. Isto porque, o cálculo de B_i conforme citado acima, não é exato, refletindo um pessimismo por vezes exagerado.

2.6.2 Protocolo de Prioridade Teto (“*Priority Ceiling Protocol*”)

A idéia central no *"Priority Ceiling Protocol"* (PCP), introduzido em [SRL90], é limitar o número de bloqueios ou inversões de prioridades e evitar a formação de cadeias de bloqueios e *"deadlocks"* em uma ativação de tarefa. O PCP é dirigido para escalonamentos de prioridade fixa, como o *"Rate Monotonic"*. Esse protocolo é uma extensão do Protocolo Herança de Prioridade ao qual se adiciona uma regra de controle sobre os pedidos de entrada em exclusão mútua.

Em essência, o PCP assegura no máximo uma inversão de prioridades por ativação. Ou seja, se uma tarefa menos prioritária T_j tiver uma seção crítica executando em um recurso compartilhado com T_i, então nenhuma outra tarefa menos prioritária que T_i conseguirá entrar em seção crítica que possa também bloquear T_i. Essa regra evita também que uma tarefa possa entrar em uma seção crítica se já houverem semáforos que podem levá-la a bloqueios.

- **Descrição do Protocolo**

Nesse protocolo, todas as tarefas apresentam também uma prioridade *nominal* ou *estática*, definida pelo RM. Uma prioridade *ativa* ou *dinâmica* que incorpora o mecanismo de herança do PHP, é também usada para definir a inclusão da tarefa nas escalas em tempo de execução. Sempre que uma tarefa menos prioritária bloquear uma mais prioritária, sua prioridade ativa assume a prioridade da tarefa mais prioritária. A

herança de prioridades é transitiva, ou seja, se uma tarefa menos prioritária T_3 bloqueia uma tarefa T_2 de prioridade média e, por sua vez, T_2 bloqueia uma tarefa mais prioritária T_1, então T_3 herda a prioridade de T_1.

Todos os recursos acessados em exclusão mútua possuem um valor de *prioridade teto* (“*ceiling*” $C(S_k)$) que corresponde à prioridade da tarefa mais prioritária que acessa o recurso.

A regra que define as entradas ou não em seções críticas é enunciada como se segue: *Uma tarefa só acessa um recurso compartilhado se sua prioridade ativa for maior que a prioridade teto (“ceiling”) de qualquer recurso já previamente bloqueado*. São excluídos dessa comparação recursos bloqueados pela tarefa requerente. Se S_l for o semáforo com maior prioridade teto entre todos os semáforos bloqueados, então uma tarefa T_i só entrará em sua seção crítica se sua prioridade dinâmica p_i for maior que o "ceiling" $C(S_l)$. Se $p_i \leq C(S_l)$ o acesso é negado a T_i.

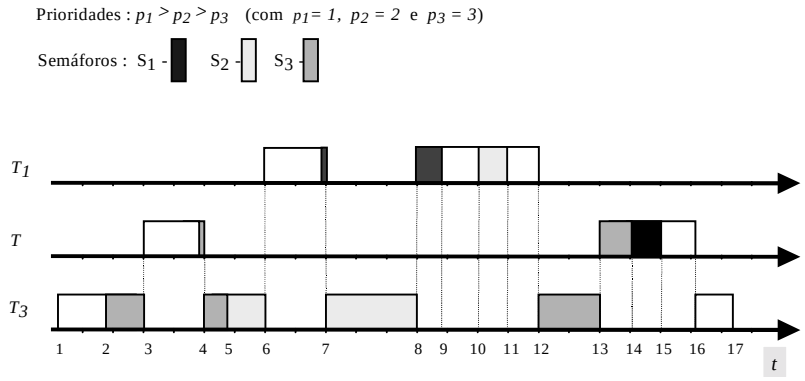
Quando nenhum recurso estiver bloqueado então o acesso ao primeiro recurso será sempre permitido. A conseqüência do uso do *Protocolo de Prioridade Teto* (PCP) é que uma tarefa mais prioritária só pode ser bloqueada por tarefas menos prioritárias uma só vez por ativação [SRL90].

O exemplo apresentado em [Kop92c] é reproduzido aqui no sentido de ilustrar o efeito do PCP sobre um conjunto de tarefas. As tarefas T_1, T_2 e T_3 cujas evoluções são apresentadas na figura 2.12, acessam em exclusão mútua os recursos R_1, R_2 e R_3. A prioridade teto de cada semáforo é definido segundo o compartilhamento dos recursos indicado na figura: $C(S_1)=1$, $C(S_2)=1$ e $C(S_3)=2$. Os eventos na evolução das tarefas, sinalizados na figura, são também descritos na própria figura 2.12. Nesse exemplo, a tarefa T_1, em cada ativação, é bloqueada no máximo uma vez por uma seção crítica de uma tarefa menos prioritário.

Além dos bloqueios diretos e por herança, o PCP introduz uma outra forma de bloqueio conhecida como bloqueio de *"ceiling"* onde uma tarefa fica bloqueada porque não possui prioridade dinâmica superior a maior *prioridade teto* dentre os recursos ocupados. Esse bloqueio é necessário para evitar as cadeias de bloqueios e os *"deadlocks"* [But97]. Na figura 2.12 a tarefa T_1 sofre um bloqueio de *"ceiling"* no tempo do evento 7.

O *"Immediate Priority Ceiling Protocol*” (IPCP) é uma versão do PCP cuja finalidade principal é a de apresentar um melhor desempenho. A herança de prioridade no IPCP deixa de se dar quando a seção crítica bloqueia a tarefa mais prioritária. A tarefa menos prioritária tem sua prioridade ativa elevada, assumindo logo no início da seção crítica a prioridade teto do recurso acessado [BuW97]. Uma conseqüência dessa mudança é que uma tarefa pode sofrer bloqueio somente no início de sua execução. Uma vez que comece a executar, a tarefa terá todos os recursos que necessite para o seu processamento. Sua execução só poderá ser postergada pelas interferências de tarefas mais prioritárias. O IPCP é mais fácil de se implementar que o PCP e envolve menos troca de contextos de tarefas. O *"Priority Protect Protocol"* apresentado nas

especificações POSIX é baseado no IPCP.



Eventos	*Descrição*
1	T_3 começa sua execução
2	T_3 entra em seção crítica (fecha o Semáforo S_3)
3	T_2 inicia seu processamento interrompendo T_3 (preempção de T_3).
4	T_2 sofre bloqueio direto: S_3 "fechado" por T_3 que reassume e herda prioridade de T_2.
5	T_3 entra na seção crítica aninhada ao fechar semáforo S_2
6	T_1 inicia e interrompe T_3. T_1 é mais prioritária que T_3 com a prioridade herdada de T_2.
7	T_1 tenta fechar semáforo S_1, é bloqueado por "ceiling"; possui prioridade igual ao maior "ceiling" de semáforo já fechado pelas outras tarefas ($C(S_2)=1$).
8	T_3 libera semáforo S_2. T_1 é reativado e interrompe T_3. T_1 entra em S_1.
9	T_1 libera S_1.
10	T_1 fecha Semáforo S_2.
11	T_1 libera S_2.
12	T_1 termina e T_3 reassume na prioridade herdada de T_2.
13	T_3 libera S_3 retornando a sua prioridade estática. T_2 interrompe T_3 e fecha S_3.
14,15,16,17	T_2 libera S_3 ; fecha e libera S_1; e completa. T_3 reassume e completa.

Figura 2.12: Exemplo de uso do PCP

- **Extensões de Testes de Escalonabilidade Tomando como Base o PCP**

Os testes de escalonabilidade mostrados anteriormente – quando da aplicação do PHP sobre um conjunto de tarefas – continuam válidos na aplicação do Protocolo de Prioridade Teto (PCP). A diferença está no valor limite de bloqueio máximo B_i que pode experimentar uma tarefa T_i. No PCP esse valor limite corresponde a duração da maior seção crítica de tarefas menos prioritárias que podem bloquear T_i. Logo, nas equações [16], [17] e [18], quando se considera o uso do PCP, B_i assume sempre o

valor da maior seção crítica que bloqueia T_i.

Uma seção crítica pertencente a uma tarefa T_j, guardada pelo semáforo S_k e de duração $D_{j,k}$ pode bloquear por *"ceiling"* uma tarefa mais prioritária T_i se e somente se [SRL90]: $p_i > p_j$ e $C(S_k) \geq p_i$. O máximo bloqueio B_i que T_i pode sofrer é dado pela duração da maior seção crítica que pode bloquear por *"ceiling"* essa tarefa ([AuB90], [But97]):

$$B_i = \max_{j,k} \left\{ D_{j,k} \mid \left(p_j < p_i\right) \wedge \left(C(S_k) \geq p_i\right) \right\}.$$

tarefas	S_1	S_2	S_3
tarefa T_1	1	1	0
tarefa T_2	1	0	1
tarefa T_3	0	4	8

Tabela 2.6

Para ilustrar o cálculo de B_i sob o uso do PCP, considere a tabela 2.6 que descreve as durações de seções críticas ($D_{j,k}$) a partir de condições do problema apresentado na figura 2.12. De acordo com a equação acima os bloqueios máximos por tarefas são dados por:

$$B_1 = max\ (1,4) = 4$$
$$B_2 = max\ (8) = 8$$
$$B_3 = 0$$

2.7 Tarefas Dependentes: Relações de Precedência

Em muitas aplicações, alguns processamentos não podem ser executados em ordens arbitrárias mas sim, em ordens previamente definidas, o que determina o surgimento de *relações de precedência* entre tarefas do conjunto. As escalas produzidas devem refletir as ordens parciais definidas através destas relações.

Alguns autores preferem expressar as relações de precedência entre tarefas com o uso de *"offsets"* [Aud93]. Neste caso, a tarefa sucessora de uma precedência é liberada pela passagem do tempo (liberada por valor de tempo correspondente ao *"offset"* que garante o tempo de resposta da predecessora). O uso de *"offsets"* pode representar em sub-utilização de recursos, uma vez que a sucessora é sempre liberada por tempo em situação de pior caso: um *"offset"* é calculado para a pior situação possível em termos

de tempo de resposta da tarefa predecessora.

Relações de precedência podem ser definidas através das necessidades de comunicação e sincronização entre as tarefas. As tarefas tipicamente, recebem mensagens, executam seus processamentos e por fim, enviam seus resultados ou sinais de sincronização na forma de mensagens. Com isto, a liberação de uma tarefa sucessora pode se dar por meio de mensagem [TBW94]. Uma conseqüência direta destas liberações por mensagem é a existência de *"release jitters"* nas liberações de tarefas sucessoras.

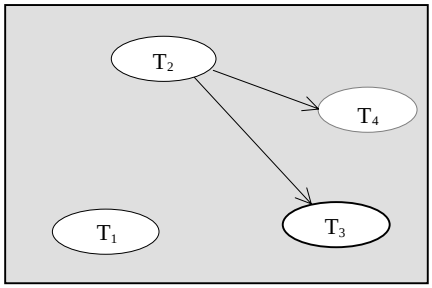
Conforme a técnica usada para a liberação de sucessoras, seja por passagem de tempo ou por mensagem, as relações de precedência são representadas nas análises de escalonabilidade, por valores de *"offsets"* ou de *"jitters"*. Em ambos os casos, esses valores devem garantir o pior caso de tempo de resposta da tarefa predecessora na relação de precedência. Ou seja, em termos de análise de escalonabilidade, os dois métodos são equivalentes pois tanto *"offsets"* como *"jitters"* devem assumir valores que garantam a execução da predecessora no seu pior caso de tempo de resposta, antes da liberação da sucessora. A diferença está em tempo de execução; enquanto, a liberação por passagem de tempo é uma técnica estática onde o *"offset"* é definido previamente, impondo sempre o pior caso de liberação, a liberação por mensagem é dinâmica e o pior caso de liberação eventualmente pode acontecer.

O conceito de *atividade* é usado como a entidade encapsuladora de *tarefas* que se comunicam e/ou se sincronizam. Cada atividade é representada por um grafo orientado acíclico onde os nodos representam tarefas e os arcos identificam as relações de precedência. As atividades são ditas *síncronas* (**"***loosely synchronous activity*") quando as tarefas liberam suas sucessoras pelo envio de mensagens; no outro caso, onde a liberação envolve *"offsets"*, as atividades são identificadas como *assíncronas* ("*asynchronous activity*") [BNT93]. Um exemplo de atividades assíncronas pode ser encontrada no sistema MARS [Kop97], onde as entidades encapsuladoras das tarefas dependentes são identificadas como *transações* e apresentam suas tarefas liberadas por passagem de tempo no sentido de implementar as relações de precedência.

Na seqüência deste item, concentramos nossas descrições em atividades síncronas (liberações por mensagem) para esquemas de prioridades fixas e nas relações de precedência que, para efeito de análise, são representadas como *"jitters"*. Nessas condições, o modelo de tarefas assume carga estática e, portanto, uma aplicação é constituída por *atividades periódicas*. Cada atividade periódica A_i corresponde a uma seqüência infinita de ativações ocorrendo em intervalos regulares de tempo P_i (período da atividade). Uma atividade A_i é caracterizada por um *"deadline"* D_i: limite máximo associado à conclusão de todas as suas tarefas. As tarefas de uma mesma atividade possuem os mesmos tempos de chegada, porém suas liberações dependem dos tempos de resposta de suas predecessoras.

A figura 2.13 mostra duas atividades: a primeira constituída de apenas uma tarefa T_1 e uma segunda, formada pelas tarefas T_2, T_3 e T_4. As relações de precedência nesta

segunda atividade implicam na ordem $T_2 \rightarrow T_3 \rightarrow T_4$ [2].



tarefas	J_i	C_i	P_i	D_i
T_1	1	10	40	40
T_2	3	10	80	25
T_3	-	5	80	40
T_4	-	10	80	80

Figura 2.13: Uma aplicação constituída por duas atividades

Quando se considera um grafo de precedências (uma atividade), a maneira natural de se atribuir prioridades é seguindo as relações do grafo com um decréscimo nas prioridades das tarefas envolvidas, ou seja, as prioridades são decrescentes ao longo do grafo de precedências seguindo as orientações dos arcos. Este tipo de atribuição, respeitando as relações de precedência, se aproxima da política *"Deadline Monotonic"*.

As comunicações usando variáveis compartilhadas – conforme visto no item 2.6 – podem levar a inversões de prioridades (bloqueios). Se a atribuição de prioridades é feita segundo as orientações dos grafos e as liberações de tarefa obedecem às relações de precedência, diminuem as possibilidades de bloqueios e inversões de prioridades, uma vez que as tarefas mais prioritárias são liberadas antes nas relações de precedência.

Considere como exemplo o conjunto de tarefas ilustrado na figura 2.13. Tomando as relações de precedência e as restrições temporais indicadas na figura, queremos verificar a escalonabilidade do conjunto. A atribuição de prioridades é feita segundo as orientações dos grafos; os índices das tarefas representam as suas respectivas prioridades (se T_i é mais prioritária que T_j então $i < j$).

O modelo introduzido coloca as atividades como síncronas o que implica em tratar precedências como *"release jitters"*. Como as tarefas possuem *"deadlines"* relativos menores que seus respectivos períodos, a verificação de escalonabilidade pode ser feita usando as equações [9] e [10] do item 2.5, onde os tempos de resposta são obtidos a partir de :

$$W_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{W_i + J_j}{P_j} \right\rceil \cdot C_j \qquad e \qquad R_i = W_i + J_i ,$$

[2] As precedências entre tarefas nas atividades podem ser expressas pela relação de ordem parcial "$\rightarrow$", definida sobre o conjunto de tarefas. Se Ti precede uma outra tarefa Tj (ou seja, Tj é sucessora de Ti), esta relação é representada por Ti$\rightarrow$Tj, indicando que Tj não pode iniciar sua execução antes de Ti terminar. A relação "$\rightarrow$" é transitiva.

onde a escalonabilidade é verificada por : $\forall i \;\; R_i \leq D_i$.

No cálculo destes tempos devem ser consideradas as relações de precedência no conjunto de tarefas. O teste acima permite a consideração de precedências na forma de *"jitters"*. O valor de *"jitter"* de uma tarefa é determinado a partir do tempo de resposta máximo da sua predecessora (pior situação de liberação).

O conjunto de tarefas com relações de precedências, passa a ser tomado como um conjunto de tarefas independentes com *"jitters"* associados. Mas as interferências assim calculadas, a partir de tarefas com *"jitters"* associados e tomadas como independentes, resultam em tempos de respostas extremamente grandes e muitas vezes, irreais. Na verdade, tarefas sujeitas a precedências determinam cenários mais restritos de interferência e bem distante do *instante crítico* [OlF97]. Por exemplo, na figura 2.13, a tarefa T_2 embora mais prioritária, não interfere com T_3 e T_4 porque ambas são liberadas após a sua conclusão; a influência de T_2 sobre estas duas tarefas se dá só na forma de *"jitter"*.

No caso da figura 2.13, T_1 é a mais prioritária e não sofre interferência de outras tarefas. O seu tempo de resposta é dado por seu tempo de computação acrescentado pelo *"jitter"* que sofre: $R_1 = C_1 + J_1 = 11$. A tarefa T_2 sofre interferência só da tarefa T_1 e o seu tempo de resposta máximo é calculado facilmente a partir das equações [9] e [10]:

$$W_2^0 = C_2 = 10$$
$$W_2^1 = 10 + \left\lceil \frac{10+1}{40} \right\rceil \times 10 = 20$$
$$W_2^2 = 10 + \left\lceil \frac{20+1}{40} \right\rceil \times 10 = 20$$

Com $W_2 = 20$ e tomando $J_2 = 3$, o valor do tempo de resposta é dado por $R_2 = 23$. A tarefa T_3, por sua vez, sofre interferências de T_1 e um *"jitter"* porque sua liberação depende da conclusão de T_2 ($J_3 = R_2$):

$$W_3^0 = C_3 = 5$$
$$W_3^1 = 5 + \left\lceil \frac{5+1}{40} \right\rceil \times 10 = 15$$
$$W_3^2 = 5 + \left\lceil \frac{15+1}{40} \right\rceil \times 10 = 15$$

O tempo de resposta de T_3 é dado por: $R_3 = W_3 + J_3 = 38$. A tarefa T_4, por sua vez, sofre interferências de T_1 e T_3 e um *"jitter"* de T_2 ($J_4 = R_2$):

$$W_4^0 = C_4 = 10$$
$$W_4^1 = 10 + \left\lceil \frac{10+1}{40} \right\rceil \times 10 + \left\lceil \frac{10+23}{80} \right\rceil \times 5 = 25$$

$$W_4^2 = 10 + \left\lceil \frac{25+1}{40} \right\rceil \times 10 + \left\lceil \frac{25+23}{80} \right\rceil \times 5 = 25$$

A tarefa T_4 tem o seu pior tempo de resposta portanto em 48 ($R_4=W_4+J_4$). Se compararmos os tempos de resposta encontrados com os *"deadlines'* relativos das respectivas tarefas na figura 2.13, verificamos que as tarefas são escalonáveis..

No modelo de tarefas apresentado, as relações de precedência são implementadas a partir de ativações por mensagens. Os tempos em comunicações locais nos modelos de tarefas ideais são desconsiderados; em situações reais, tempos não desprezíveis podem ser adicionados aos tempos de computação das tarefas predecessoras (emissoras de mensagens), aproximando então o modelos reais de premissas de tempos nulos em comunicações locais.

Os cálculos de tempos de resposta em ambientes distribuídos, envolvendo precedências, não é muito explorado na literatura [Fid98]. As atividades nestes ambientes se estendem por vários nós (vários domínios de escalonamento local) o que implica em precedências remotas. Estas situações exigem considerações especiais. Soluções para problemas distribuídos devem se basear na assim chamada "*holistic schedulability analisys*" para sistemas de tempo real distribuídos [TiC94]: O *"release jitter"* de uma mensagem depende do pior caso de tempo de resposta da tarefa emissora. O pior caso de tempo de resposta de uma tarefa receptora depende do tempo de resposta de suas mensagens.

2.8 Escalonamento de Tarefas Aperiódicas

Todas as técnicas de escalonamento apresentadas até este item eram dirigidas para modelos de tarefas periódicas. Mas aplicações de tempo real, de um modo geral, envolvem tanto tarefas periódicas como aperiódicas. Examinamos neste item o escalonamento de tarefas aperiódicas em abordagens mistas, envolvendo tarefas críticas e não críticas. As tarefas periódicas são assumidas como críticas, necessitando de garantias em tempo de projeto para condições de pior caso. As tarefas aperiódicas podem envolver diferentes requisitos temporais: críticos, não críticos ou ainda sem requisitos temporais.

As aperiódicas apresentando um mínimo intervalo entre suas ativações e um *"deadline hard"* são identificadas como tarefas esporádicas, possuindo um comportamento temporal determinista – o que facilita, portanto, a obtenção de garantias em tempo de projeto. As tarefas aperiódicas que não possuem seus tempos de chegada conhecidos e também não se caracterizam por um intervalo mínimo entre suas ativações, definem o que se pode chamar de uma carga computacional dinâmica. Com estas últimas tarefas é possível a obtenção de garantias dinâmicas ou, ainda, usar técnicas de melhor esforço no sentido de executá-las segundo as disponibilidades do

processador em tempo de execução. As aperiódicas que apresentam *"deadlines hard"* e necessitam de garantias dinâmicas em seus escalonamentos são chamadas de tarefas aperiódicas *"firm"*. As tarefas aperiódicas com requisitos não críticos (*"deadline soft"*) e as sem requisitos temporais (aplicações não de tempo real) necessitam apenas de bons tempos de resposta.

Num quadro misto, uma questão que pode ser colocada está ligada ao tipo de política adequado para tarefas periódicas e que possa ser estendido para carga dinâmica: *são as políticas de prioridade fixa (RM, DM e etc.) ou políticas de prioridade dinâmica (EDF) as mais apropriadas?* As políticas baseadas em prioridade fixa foram sempre as preferidas para esquemas mistos de escalonamento. Embora apresentem melhor fator de utilização, se comparado com esquemas de prioridade fixa, as políticas de prioridade dinâmica como o EDF eram consideradas por alguns autores até pouco tempo como instáveis para tratar com carga dinâmica [SSL89]. Nos últimos anos, a direção dos trabalhos tem mudado e o EDF tem sido também alvo de extensões para escalonamentos mistos. Os algoritmos dinâmicos apresentam os mais altos limites de escalonabilidade o que permite uma maior utilização do processador o que, por sua vez, aumenta a capacidade de processamento da carga aperiódica.

As sobras de processador nas escalas são importantes para o escalonamento de tarefas aperiódicas em modelos híbridos. Existem dois tipos de abordagens para a determinação de sobras de processador: as soluções baseadas em *servidores* [SSL89] e as baseadas em *tomadas de folgas* ("*slack stealing*") [DtB93], [LeR92]. Neste texto são apresentadas unicamente técnicas de escalonamento para tarefas aperiódicas baseadas no conceito de *servidor*. Os escalonamentos híbridos neste capítulo são construídos com políticas de prioridade fixa [SSL89]. No *Anexo A* são apresentados escalonamentos mistos usando políticas de prioridade dinâmica [SpB96].

2.8.1 Servidores de Prioridade Fixa [LSS87, SSL89]

As técnicas examinadas nesse item são para políticas baseadas em prioridades fixas, mais precisamente, o *"Rate Monotonic"*. As sobras nas escalas de carga periódica, são determinadas estaticamente, em tempo de projeto, e posteriormente, em tempo de execução, são atribuídas ao processamento aperiódico usando o conceito de *servidor*.

- **Servidor de *"Background"***

Este servidor é extremamente simples. A idéia central corresponde em atender as requisições aperiódicas quando a fila de prontos envolvendo tarefas periódicas está vazia, ou seja, se tarefas periódicas não estão se executando ou pendentes, o processador é entregue para a carga aperiódica.

A determinação de prioridades nesta abordagem é feita atribuindo - segundo o RM - as prioridades mais altas para as tarefas periódicas. As prioridades mais baixas são destinadas para as tarefas aperiódicas. Como conseqüência, o *"Background Server"*

(BS) apresenta tempos de resposta muito altos para cargas aperiódicas. Se a carga envolvendo as tarefas periódicas é alta, então a utilização deixada para o serviço de *"Background*" é baixa ou não freqüente.

tarefas	C_i	P_i	D_i	p_i
tarefa periódica A	4	10	10	1
tarefa periódica B	8	20	20	2
tarefa aperiódica C	1	-	-	3
tarefa aperiodica D	1	-	-	3

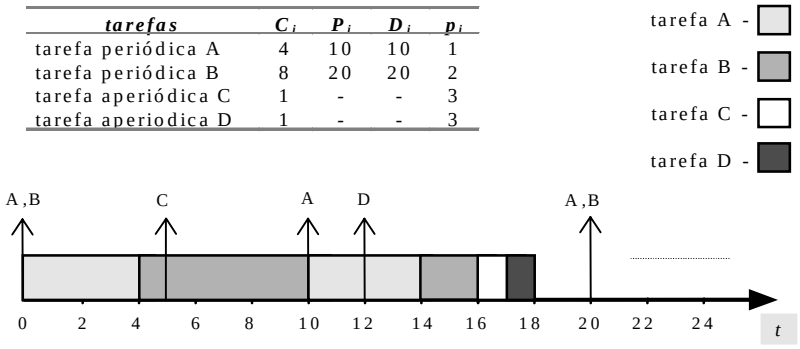


Figura 2.14: Servidora de "*Background*"

A figura 2.14 ilustra um exemplo introduzido em [SSL89] onde duas tarefas periódicas e duas requisições aperiódicas são executadas usando uma atribuição de prioridades RM. Com base no teste do RM, a carga periódica tem garantia em tempo de projeto pois a sua utilização não passa o limite de 0,828 (equação [2], item 2.4). As requisições C e D são executadas no fim da escala da figura 2.16, depois que a carga periódica foi completada.

O BS é bastante simples na sua implementação, porém só é aplicável quando as requisições aperiódicas não são críticas e a carga periódica não é alta.

- ***"Polling Server"***

O esquema do *"Polling Server"* (PS) consiste na definição de uma tarefa periódica para atender a carga aperiódica [SSL89]. Um espaço é aberto periodicamente na escala para a execução da carga aperiódica, através da tarefa Servidora de *"Polling"*. A tarefa servidora possui um período P_{PS} e um tempo de computação C_{PS} e, como as outras tarefas da carga periódica do sistema, tem a sua prioridade atribuída segundo o *"Rate Monotonic"*. Em cada ativação, a tarefa servidora executa as requisições aperiódicas pendentes dentro do limite de sua *capacidade* C_{PS} – o tempo destinado para o atendimento de carga aperiódica em cada período da servidora.

Quando não houver requisições aperiódicas pendentes, a tarefa PS se suspende até a sua nova chegada, no próximo período. Neste caso, a sua capacidade C_{PS} é entregue para a execução de tarefas periódicas pendentes. Se um pedido aperiódico ocorre logo depois da suspensão da tarefa servidora, o pedido deve aguardar até o início do próximo período da tarefa PS.

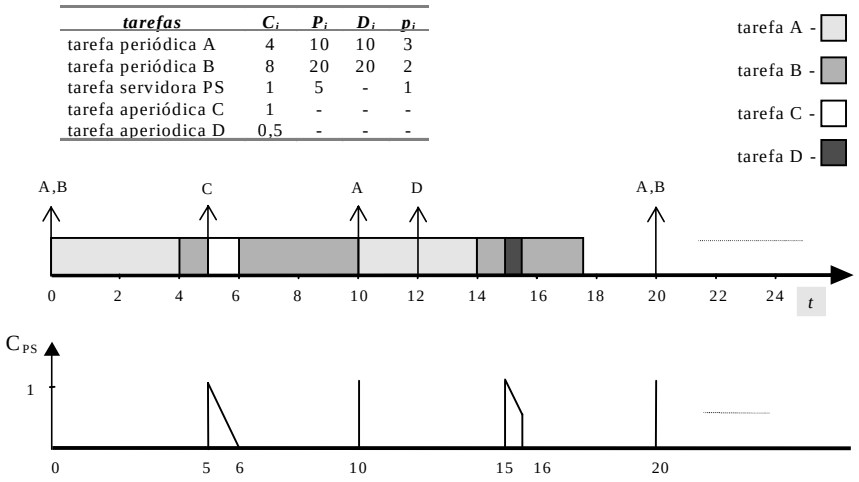


Figura 2.15: Algorítmo "Polling Server"

O mesmo exemplo usado com o BS é mostrado na figura 2.15 onde a carga aperiódica é escalonada segundo o algoritmo PS. Nesse caso, a tarefa servidora é criada com capacidade C_{PS} de uma unidade e o período P_{PS} de 5 unidades. Na ativação da servidora em *t=0* não existe carga aperiódica e a sua capacidade é entregue para a execução das tarefas periódicas. Em *t = 5*, a chegada de uma requisição aperiódica C coincide com a chegada da servidora PS. Com isto, a capacidade C_{PS} é consumida totalmente até *t* = 6. No período seguinte da servidora (*t = 10*), novamente não existe carga aperiódica pendente e a capacidade da servidora, que foi restaurada no seu máximo no início deste período, é entregue a carga periódica. A servidora, por não estar mais ativa, não atende a segunda requisição aperiódica D que chega em *t=12*. No início de seu período seguinte (em *t=15*), esta requisição é executada, consumindo a metade da capacidade da servidora, conforme mostra a figura.

A interferência da tarefa servidora sobre o conjunto de tarefas periódicas do sistema é no pior caso igual a interferência causada por uma tarefa com tempo de computação C_{PS} e período P_{PS} ou seja, dada a utilização do PS, a escalonabilidade do conjunto periódico é garantido por :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} + \frac{C_{PS}}{P_{PS}} \leq (n+1)\left(2^{1/(n+1)} - 1\right),$$

ou seja

$$U_P \leq (n+1)\left(2^{1/(n+1)} - 1\right) + U_S .$$

A abordagem do *"Polling Server"*, se comparada com a abordagem BS, melhora o tempo de resposta médio de tarefas aperiódicas. O PS porém não fornece serviço de

resposta imediato para processamentos aperiódicos. O tempo de resposta de requisições aperiódicas depende do período e da capacidade da tarefa servidora.

- ***"Deferrable Server"***

O *"Deferrable Server"* (DS) também é baseado na criação de uma tarefa periódica que no conjunto de tarefas da carga estática, recebe uma prioridade segundo uma atribuição RM. Ao contrário do PS, o DS conserva a sua capacidade – tempo destinado para o processamento aperiódico – mesmo quando não existir requisições durante a ativação da tarefa DS. Requisições não periódicas podem ser atendidas no nível de prioridade da tarefa servidora, enquanto a sua capacidade C_{DS} não se esgotar no período correspondente. No início de cada período da tarefa servidora, a sua capacidade processamento é restaurada.

Por preservar sua capacidade, a abordagem DS fornece melhores tempos de resposta para as tarefas aperiódicas que o *"Polling Server"*. Como a tarefa servidora usualmente executa na prioridade mais alta do conjunto periódico, se a capacidade for suficiente, o atendimento de requisições aperiódicas é imediato.

tarefas	C_i	P_i	D_i	p_i
tarefa periódica A	4	10	10	3
tarefa periódica B	8	20	20	2
tarefa servidora PS	1	5	-	1
tarefa aperiódica C	1	-	-	-
tarefa aperiodica D	0,5	-	-	-

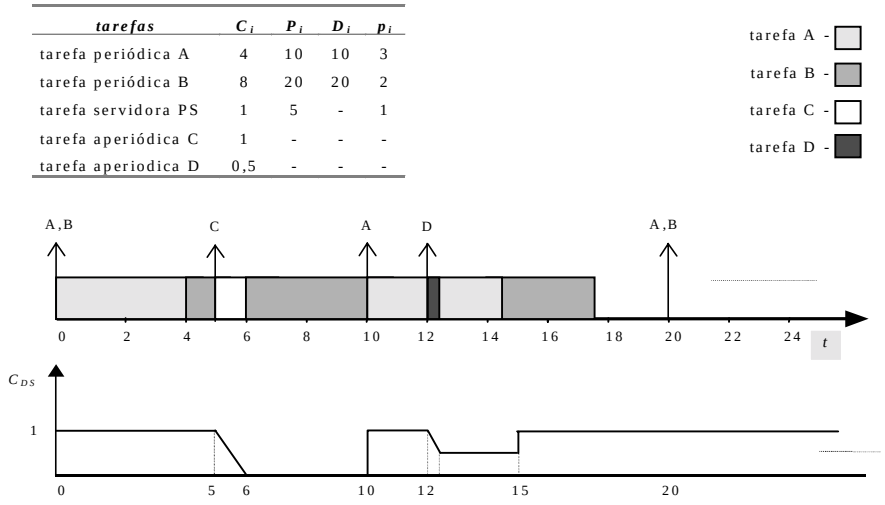


Figura 2.16: Algorítmo "Deferrable Server"

A figura 2.16 ilustra o uso do algoritmo DS no escalonamento da carga aperiódica com o mesmo exemplo introduzido em [SSL89]. Neste exemplo é criada uma tarefa servidora com capacidade $C_{DS}=1$ e de período $P_{DS} = 5$. Na ativação da servidora em $t=0$ não existe carga aperiódica e a sua capacidade é preservada durante todo o período P_{DS}. Em $t=5$, a chegada de uma requisição aperiódica C coincide com a chegada da servidora DS, o que determina o consumo total da capacidade da servidora até $t=6$

(figura 2.16). No período seguinte da servidora (*t=10*), a capacidade C_{DS} é novamente preenchida ao seu máximo. Este valor de capacidade se mantém até a chegada da requisição aperiódica D em *t=12* que então consome 0,5 da capacidade da servidora até *t=12,5*. No início do período seguinte da servidora (*t=15*), a sua capacidade volta ao seu valor máximo ($C_{DS}=1$). A figura 2.16 confirma sobre o mesmo exemplo usado nas técnicas anteriores, o melhor desempenho do servidor DS sobre os anteriores em termos de tempo de resposta e serviço de resposta imediata.

Quando usando a política RM, a influência da tarefa servidora DS sobre a utilização da carga periódica não pode ser determinada de maneira tão simples como no caso do PS. O comportamento da servidora com a prioridade mais alta, podendo se executar em qualquer ponto do seu período não é captada pelo teste do RM (equação [2], item 2.4). Nas condições do teste original do RM, a tarefa periódica de mais alta prioridade necessita executar em seu tempo de chegada; qualquer atraso pode prejudicar as tarefas menos prioritárias. Em [SSL89], derivada do ajuste no teste do RM para captar o comportamento singular da servidora DS, é apresentada uma relação entre a utilização da servidora e a utilização da carga periódica:

$$U_P \leq \ln\left(\frac{U_{DS}+2}{2U_{DS}+1}\right). \quad [19]$$

A equação [19] é válida somente para um muito grande número de tarefas periódicas no sistema.

- **Servidor Troca de prioridade (“*Priority Exchange Server*”)**

Uma outra técnica de escalonamento apresentada em [LSS87] e [SSL89] para o processamento de requisições aperiódicas em escalonamento híbrido é o *"Priority Exchange Server*" (PE). Diferentemente do DS, neste servidor, diante da ausência de requisições aperiódicas, a capacidade de processamento aperiódico C_{PE} (tempo de computação da tarefa servidora) é preservada executando trocas de prioridades da servidora com tarefas periódicas pendentes. Não será discutido neste texto o algoritmo do PE devido a pouca possibilidade de aplicação deste servidor ligada à complexidade do mecanismo de troca de prioridades. Os leitores interessados podem encontrar informações sobre este servidor nas indicações bibliográficas acima.

- **Servidor Esporádico**

O *"Sporadic Server"* (SS), a exemplo dos algoritmos DS e PE, é outra técnica introduzida em [SSL89] que apresenta bons tempos de resposta e de serviço imediato para requisições aperiódicas. Com características semelhantes a dos anteriores, foi introduzido para possibilitar a execução de tarefas aperiódicas com restrições críticas.

O SS cria uma tarefa periódica que atua em um só nível de prioridade para executar requisições aperiódicas. Para entender o funcionamento do algoritmo do SS é

necessário que se introduza alguns termos:

- p_s : corresponde ao nível de prioridade em execução no processador;
- p_i : é um dos níveis de prioridades do sistema.
- Intervalo Ativo : uma prioridade p_i é dita em um intervalo ativo quando $p_i \leq p_s$.
- Intervalo de Prioridade Desativada: uma prioridade p_i é dita desativada quando $p_i > p_s$.
- Tempo de Preenchimento RT_i : define o instante de tempo em que se dá a restauração da capacidade consumida durante o intervalo em que a prioridade p_i estava ativa.

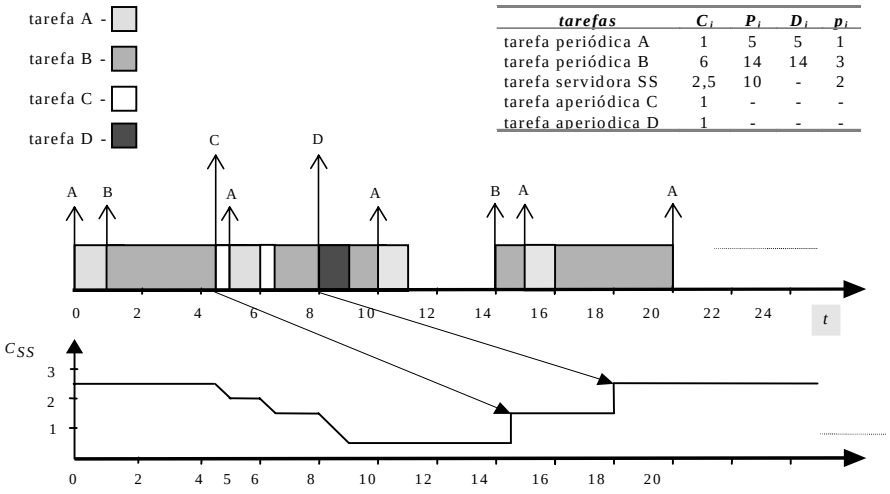
A tarefa servidora no SS preserva sempre a sua capacidade no nível em que foi projetada. Mas difere das outras abordagens anteriores na forma do preenchimento de sua capacidade:

- Se a servidora tem seu tempo computação (capacidade) consumido em um de seus períodos, o preenchimento correspondente ocorrerá no seu tempo de preenchimento (RT_i) que é determinado adicionando o valor do período da servidora ao tempo de início do intervalo onde p_i era ativo e ocorreu o consumo considerado.
- A quantidade a ser preenchida é igual a capacidade do servidor consumida no intervalo ativo.

A figura 2.17 apresenta um exemplo de um escalonamento híbrido com um servidor esporádico possuindo prioridade média no conjunto de tarefas periódicas. Neste exemplo também apresentado em [SSL89], a tarefa servidora SS é definida com capacidade $C_{SS}=2,5$ e período $P_{SS}=10$. Em $t=0$, a tarefa A (a mais prioritária) começa a executar. A prioridade p_s em execução ($p_s = p_A$) é então maior que a prioridade da servidora SS (p_{SS}); o que define o primeiro intervalo ativo da servidora SS nesta execução de A ($p_{SS} < p_A$). Neste intervalo não ocorre consumo de capacidade C_{SS} devido a ausência de requisições aperiódicas.

Em $t=4,5$ uma requisição aperiódica C chega e como a tarefa SS é mais prioritária que B (tarefa em execução), ocorre a preempção da tarefa periódica. O consumo da capacidade da servidora por parte de C vai até $t = 5$ quando, pela chegada da tarefa periódica A, ocorre a interrupção da tarefa aperiódica C. Pelo RM a tarefa A é a mais prioritária. Concluída esta ativação de A, a tarefa C reassume. Em $t = 6,5$ o processamento aperiódico C é concluído. O tempo de preenchimento (RT_i), referente ao consumo de capacidade por parte da requisição C, é programado considerando o intervalo ativo correspondente ($p_{SS} \leq p_s$) que, neste caso, inicia com a chegada da

requisição aperiódica (*t = 4,5*). Portanto, como pode ser visto na figura 2.17, o preenchimento da capacidade consumida neste intervalo ocorre em *t=14,5* ($RT_i=t_a+P_{SS}$). A preempção de C pela tarefa A em *t = 5* não define dois intervalos ativos da servidora para as duas partes de C da figura 2.17. Na verdade, um mesmo intervalo ativo se mantém durante as execuções de C e A porque, entre os tempos *t=4,5* e *t = 6,5*, a condição $p_{SS} \leq p_s$ se mantém como válida.



tarefas	C_i	P_i	D_i	p_i
tarefa periódica A	1	5	5	1
tarefa periódica B	6	14	14	3
tarefa servidora SS	2,5	10	-	2
tarefa aperiódica C	1	-	-	-
tarefa aperiodica D	1	-	-	-

Figura 2.17: Algorítmo "Sporadic Server"

Uma outra requisição aperiódica (tarefa D) chega em outro intervalo ativo da servidora SS. A requisição D também interrompe a tarefa B e consome uma unidade de C_{SS}. O tempo de início do intervalo ativo de D coincide com a sua chegada e, portanto, o tempo de preenchimento (RT_i) correspondente deve ocorrer em *t=18*.

A tarefa servidora SS não apresenta um comportamento convencional de tarefa periódica, uma vez que, a capacidade desta servidora é preservada no mesmo nível como o DS e a sua execução é postergada até a ocorrência de uma requisição aperiódica. Porém em [SSL89], é provado que a técnica de preenchimento da capacidade da servidora compensa este comportamento não convencional, permitindo que, em termos de análise de escalonabilidade, esta tarefa possa assumir um comportamento periódico. Ou seja, a servidora SS pode ser substituída no teste do RM ([2] no item 2.4) por uma tarefa periódica com período P_{SS} e tempo de computação C_{SS}. A limitação que a servidora SS impõe sobre uma carga periódica é dada por :

$$U_p \leq \ln\left(\frac{2}{U_{SS}+1}\right). \qquad [20]$$

A equação [20] é idêntica à obtida para o servidor PE e também só é válida para um número muito grande de tarefas periódicas no sistema.

A tabela 2.7 mostra um exemplo de comparação das utilizações dos servidores DS, PE e SS quando envolvidos com uma mesma carga periódica ([SSL89]). O algoritmo SS possui a simplicidade do DS e a vantagem da maior capacidade do PE para o processamento de requisições aperiódicas. Porém, diferente destes outros algoritmos, o SS pode também ser usado na garantia em tempo de projeto.

tarefas	C_i	P_i	U_i (%)
tarefa 1	2	10	20,0
tarefa 2	6	14	42,9
servidor DS	1,00	5	20,0
servidor PE	1,33	5	26,7
servidor SS	1,33	5	26,7

Tabela 2.7: Utilização dos servidores DS, PE e SS

Na verdade, o SS foi introduzido com o objetivo de garantir a execução de tarefas esporádicas – um caso especial de aperiódicas onde existe um limite conhecido como o mínimo intervalo entre ativações (min_i). Os *"deadlines"* associados a estas tarefas são críticos (*"hard"*) e, portanto, precisam de uma garantia em tempo de projeto. Esta garantia pode ser obtida criando uma servidora SS para tratar exclusivamente uma tarefa esporádica nas suas diversas ativações. A servidora SS assume os *"deadlines"* das requisições da tarefa associada. O período P_{SS}, por sua vez, deve ser no máximo igual ao intervalo mínimo entre ativações da tarefa esporádica (min_i). Esta tarefa servidora conserva a capacidade de processamento aperiódico no seu nível de prioridade até a ocorrência de uma requisição esporádica. A capacidade C_{SS} da servidora deve ser suficiente para atender as necessidades de tempo de computação da tarefa esporádica associada, em cada uma de suas ativações. Nestas condições, os *"deadlines"* críticos podem ser garantidos em tempo de projeto.

O algoritmo SS pode ser usado para garantir tarefas esporádicas apresentando *"deadlines"* relativos (D_i) iguais ou menores que os seus respectivos intervalos mínimos entre ativações (min_i). Nos casos onde os *"deadlines"* relativos são iguais aos respectivos intervalos mínimos, as prioridades da tarefa servidora SS e da carga periódica são determinadas seguindo uma atribuição RM e as escalas são produzidas usando os mesmos algoritmos citados acima.

Para os casos onde tarefas esporádicas apresentam $D_i < min_i$ a atribuição RM não pode ser usada; é necessária uma outra atribuição de prioridades que não seja mais baseada na freqüência de chegada das tarefas periódicas. A figura 2.18 ilustra um exemplo de uma escala com atribuição RM onde ocorre uma sobrecarga com perda de *"deadline"* da tarefa aperiódica C em $t=10$. Neste exemplo, a servidora SS que possui o *"deadline"* relativo mais restritivo ($D_{SS}=10$) apresenta o maior período ($P_{SS}=32$) e portanto a menor prioridade segundo o RM.

tarefas	C_i	P_i	D_i	Min_i	p_i
tarefa periódica A	4	12	12	-	1
tarefa periódica B	4	20	20	-	2
tarefa servidora SS	8	32	10	-	3
tarefa aperiódica C	8	-	10	32	-

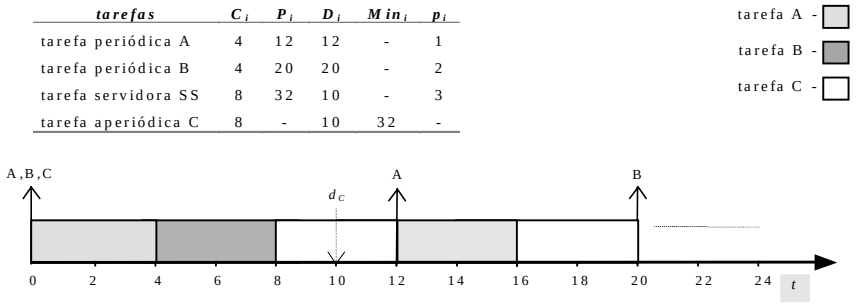
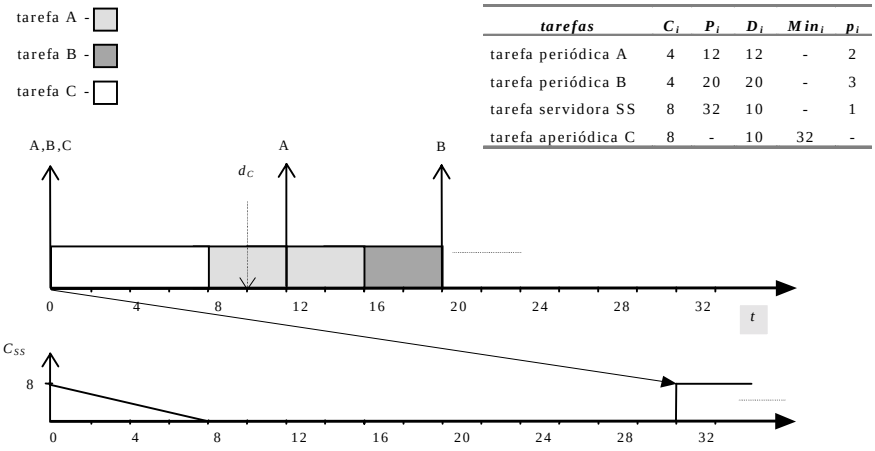


Figura 2.18: Servidor SS e RM usados em carga aperiódica com $D_i = Min_i$

Nos casos de tarefas esporádicas com $D_i<min_i$, são necessárias políticas que sejam dirigidas por *"deadlines"*, permitindo então a atribuição do mais alto nível de prioridade à servidora SS associada. A política de atribuição estática *"Deadline Monotonic"* é a mais apropriada nestes casos. Na figura 2.19 uma escala é mostrada com o mesmo conjunto de tarefas sujeito a uma atribuição DM de prioridades e onde todos os *"deadlines"* são respeitados. A tarefa C se executa em *t=0* (a servidora SS é a tarefa mais prioritária) e, o preenchimento da capacidade consumida correspondente ocorre em *t=32*, portanto, antes de esgotado o intervalo mínimo entre ativações de C (min_C).



tarefas	C_i	P_i	D_i	Min_i	p_i
tarefa periódica A	4	12	12	-	2
tarefa periódica B	4	20	20	-	3
tarefa servidora SS	8	32	10	-	1
tarefa aperiódica C	8	-	10	32	-

Figura 2.19: Servidor SS e DM usados em carga aperiódica com $D_i < Min_i$

2.8.2 Considerações sobre as Técnicas de Servidores

Algumas das premissas assumidas para os servidores seguiram modelos de tarefas originais dos algoritmos de escalonamento usados, mas isto não limita o uso destas técnicas de servidores. Os algoritmos de servidores apresentados neste texto podem ser usados em modelos com tarefas periódicas possuindo *"deadlines"* relativos arbitrários e com recursos compartilhados. Neste caso a análise de escalonabilidade deve levar em consideração as particularidades do modelo de tarefas usado.

Neste texto, as requisições aperiódicas foram apresentadas como processamentos sem prazos (*"deadlines"*), escalonadas segundo abordagens de melhor esforço usando políticas FIFO. Tarefas aperiódicas podem possuir restrições temporais e serem ordenadas segundo estas restrições com políticas diferentes das que conduzem a ordenação das periódicas no escalonamento híbrido. As tarefas aperiódicas necessitando a cada ativação de uma garantia dinâmica são identificadas como *tarefas firmes* (item 2.8). Neste caso, um teste de aceitação é necessário para verificar a escalonabilidade da tarefa aperiódica recém chegada junto com as tarefas previamente garantidas. Se o teste falha a tarefa aperiódica é descartada. Em [But97] é discutido essa verificação de tarefas firmes. Os algoritmos PS, PE e DS são apropriados na verificação dinâmica da escalonabilidade de tarefas aperiódicas firmes. O servidor SS, por sua vez, permite o tratamento de tarefas esporádicas que necessitam de garantias em tempo de projeto para os seus *"deadlines hard"*.

2.9 Conclusão

Em sistemas onde as noções de tempo e de concorrência são tratadas explicitamente, conceitos e técnicas de escalonamentos formam o ponto central na previsibilidade do comportamento de sistemas de tempo real. Esse capítulo se concentrou sobre técnicas para escalonamentos dirigidos a prioridades. Essa escolha na abordagem de escalonamento é porque a mesma cobre diversos aspectos de possíveis comportamentos temporais em aplicações de tempo real e, também, devido a importância da literatura disponível.

Vários *problemas de escalonamento* – que podem ser vistos como extensões aos problemas propostos em [LiL73] – foram examinados neste capítulo. Particularmente, foram apresentados escalonamentos de tarefas periódicas com *"deadlines"* arbitrários, o compartilhamento de recursos e a implementação de relações de precedência. As tarefas aperiódicas são escalonadas usando escalonamentos híbridos baseados no conceito de *servidor*. Todos estes problemas foram discutidos usando atribuições de prioridades fixas neste capítulo. Estes mesmos problemas são revistos com políticas de prioridade dinâmica no *Anexo A*. Escalonamentos com atribuições dinâmicas – como os definidos pelo EDF, embora determinem uma maior utilização apresentam sempre uma complexidade maior em tempo de execução.

A grande difusão de suportes (núcleos, sistemas operacionais), na forma de produtos, que baseiam seus escalonamentos em mecanismos dirigidos a prioridade é sem dúvida uma forte justificativa para o uso das técnicas apresentadas neste capítulo em problemas práticos. Alguns dos algoritmos apresentados nesse capítulo são recomendados por entidades de padronização como a POSIX e a OMG (*"Object Management Group"* [OMG98]).

Leituras complementares recomendadas referente ao assunto tratado neste capítulo são encontradas em: [AuB90], [Bak91], [Fid98], [RaS94], [SRL90], [SSL89], [Spu96], [TBW94].

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.

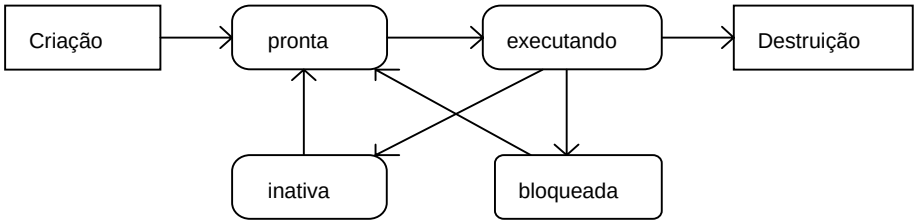


Figura 3.1 - Estados de uma *"thread"*.

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.

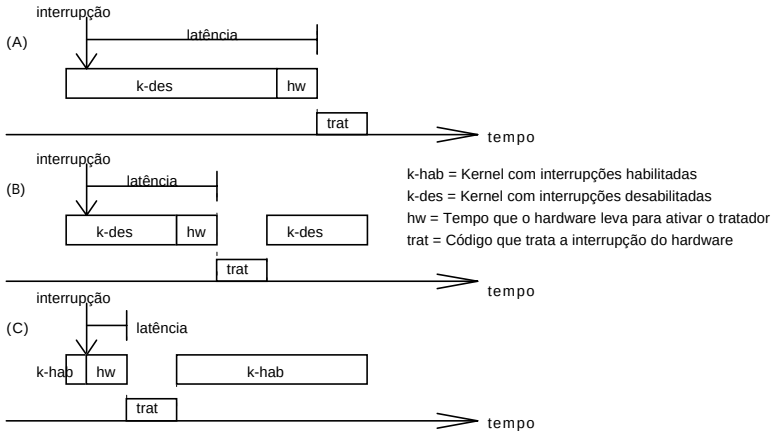


Figura 3.2 - Formas do *"kernel"* lidar com interrupções de hardware.

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.

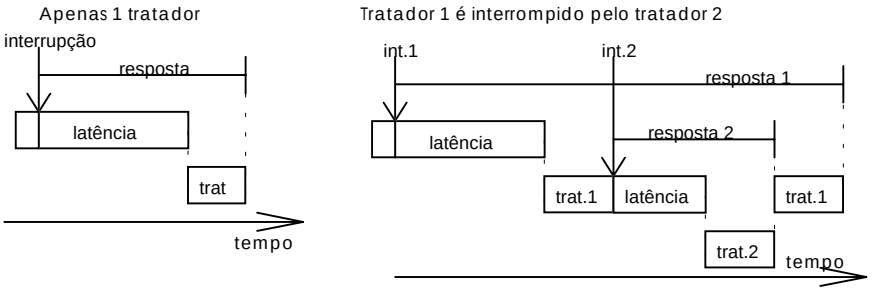


Figura 3.3 - Tempo de resposta de um tratador simples.

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.

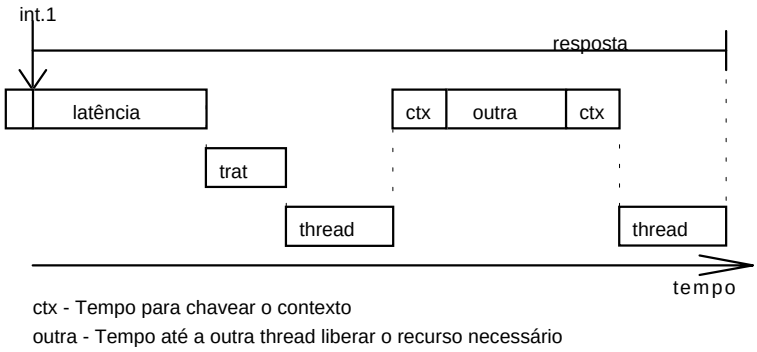


Figura 3.4 - Comportamento de tratador complexo em *"kernel"* que sofre interrupções.

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.

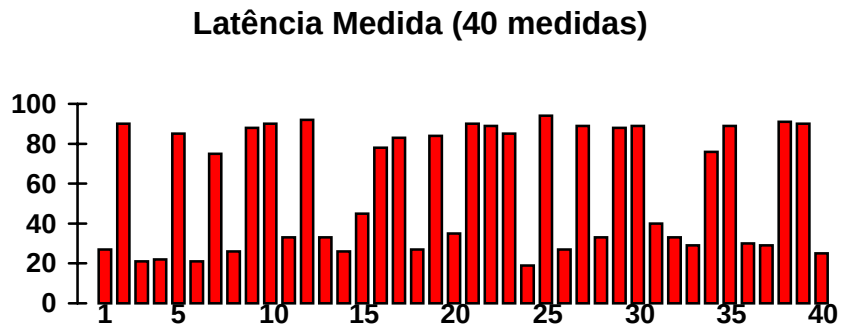


Figura 3.5 - Latência medida em 40 oportunidades consecutivas.

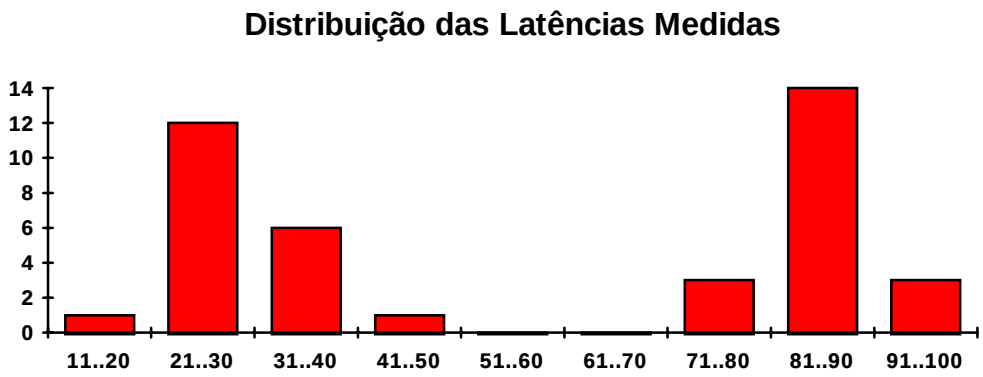


Figura 3.6 - Distribuição das latências medidas conforme sua duração.

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.

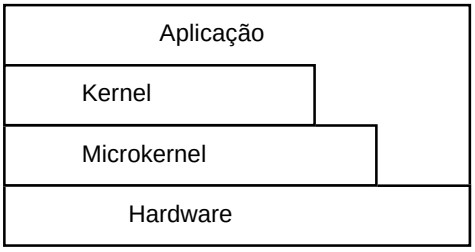


Figura 3.8 – Estratificação de serviços em um sistema.

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.

Capítulo 4

O Modelo de Programação Síncrona para os Sistemas de Tempo Real

4.1 Introdução

Os *Sistemas de Tempo Real* são sistemas que reagem, gerando respostas, à estímulos de entrada vindos de seus ambientes. Estas características os colocam como *Sistemas Reativos* especiais que estão submetidos a restrições temporais em suas reações. Neste tipo de sistemas, devem se garantir não somente a correção lógica ("*correctness*") mas também a correção temporal ("*timeliness*").

O modelo de programação síncrona parte do principio que o ambiente não interfere com o sistema (ou o programa) durante os processamentos das reações. Na recepção do evento de entrada, após um eventual cálculo, a resposta é considerada emitida simultaneamente à entrada, o que caracteriza uma reação como instantânea. O modelo é dito *síncrono* porque as saídas do sistema podem ser vistas como sincronizadas com as suas entradas. A principal conseqüência desta hipótese – a *Hipótese Síncrona* – é uma simplificação conceptual que facilita a modelagem e a análise formal das propriedades do sistema.

A hipótese síncrona, que forma a base conceitual deste estilo de programação, no sentido da reação instantânea, parte do pressuposto que existe uma máquina suficientemente rápida para executar o processamento correspondente à reação em tempos não significativos. O entendimento desta hipótese, em um sentido mais prático, é que a duração do processamento referente a reação, se comparado aos tempos relacionados com o ambiente externo, é desprezível. O ambiente externo não evolui durante esse processamento. A hipótese síncrona também define que os eventos ocorridos no sistema sejam percebidos instantaneamente em diferentes partes do sistema.

O modelo de programação síncrono é natural do ponto de vista do programador por facilitar a construção e a compreensão de programas e por lhe permitir a verificação destes. Ao separar a lógica do comportamento de um sistema das características de sua implementação, esse estilo de programação facilita a programação do mesmo.

Uma das características mais importantes encontrada nos modelos síncronos é a rejeição do não determinismo. Um sistema é visto como determinista se a mesma

seqüência de entradas produz sempre a mesma seqüência de saídas e não determinista em caso contrário. É desejável e quase sempre mandatário que os sistemas reativos se comportem de forma determinista, com as suas saídas dependendo unicamente das entradas vindas do ambiente e de exigências temporais; o comportamento esperado no controle de um avião ou de outro veículo, por exemplo ilustram esta necessidade. Além do mais o estudo do comportamento de sistemas não deterministas é mais complexo que o de sistemas deterministas por apresentar situações de erro que podem não ser reproduzíveis e por tratar com grande número de estados. Consequentemente é aconselhável tratar sistemas intrinsecamente deterministas como os sistemas reativos por modelos e linguagens deterministas e reservar abordagens não deterministas as aplicações que o necessitam como é o caso de aplicações nas quais não há como garantir comportamentos repetitivos nem garantir tempos de resposta; Internet é um exemplo claro deste comportamento.

As linguagens síncronas que apresentam características de concorrência e de determinismo e permitem ter o controle dos tempos de respostas são bem adaptadas para a programação dos Sistemas Reativos e dos Sistemas de Tempo Real. Ferramentas automáticas que possibilitam a verificação da correção lógica e temporal desses sistemas são associadas à estas linguagens. A hipótese de sincronismo permite ainda que programas escritos numa linguagem síncrona sejam compilados em autômatos eficientes e depois facilmente implementados em linguagens de programação clássicas.

Os Sistemas Reativos e os Sistemas de Tempo Real incluem em doses diferentes segundo as aplicações, atividades de manuseio de dados e atividades de controle. Quando as atividades de manuseio de dados são importantes e complexas enquanto as de controle são reduzidas como em aplicações de processamento de sinal, as técnicas de especificação e de programação mais apropriadas seguem um estilo orientado a fluxo de dados como nas linguagens síncronas declarativas Lustre [HCR91] e Signal [LLG91]. Nestas linguagens, a reação gera saídas a partir da avaliação de um conjunto de equações que as definem em função das entradas atuais e das entradas previas (armazenadas).

Quando predominam as atividades de controle e que o manuseio de dados é simples, como é o caso em aplicações de controle de processos, sistemas embutidos, supervisão de sistemas, protocolos de comunicação, interfaces homem-máquina, drivers de periféricos, entre outras, é mais apropriada adotar um estilo orientado ao fluxo de controle como nas linguagens síncronas imperativas como Esterel [BoS91] ou nos autômatos hierárquicos como Statecharts [Har87]. Nestas linguagens, cada reação corresponde a passagem de uma situação em termos de controle à uma nova situação.

Grandes aplicações podem ter partes mais orientadas ao manuseio de dados e outras ao controle; no momento atual, não existe ainda unificação entre os dois estilos ao nível de linguagem de programação, apesar de existir atividades de pesquisa nesta área. Neste livro, adotaremos a linguagem Esterel como ferramenta para a programação de aplicações tempo real dentro do modelo de programação síncrona, por ela ser imperativa, ter uma sintaxe e semântica de fácil aprendizagem e por ter disponível um ambiente automático de especificação, validação e implementação.

4.2 Princípios Básicos do Modelo de Programação da Linguagem Esterel

A linguagem Esterel é uma linguagem síncrona desenvolvida a partir de 1982 conjuntamente por dois laboratórios franceses do INRIA e da ENSMP. A necessidade de atender simultaneamente concorrência e determinismo é a base do modelo de programação síncrono da linguagem Esterel. Os princípios básicos deste modelo são os seguintes:

- *Reatividade*: O modelo é reativo uma vez que se aplica a sistemas que entram em ação reagindo à presença de estímulos vindos do ambiente em instantes discretos. Cada reação, portanto, está associada a um instante preciso; o conjunto destes instantes caracteriza a vida do sistema reativo.

- *Sincronismo*: As reações sendo instantâneas, entradas e saídas se apresentam sincronizadas − é a base da hipótese síncrona. As reações são atômicas, o modelo síncrono não permite uma nova ativação do sistema enquanto o mesmo estiver reagindo ao estímulo atual. Portanto, não há concorrência entre as reações, eliminando assim uma fonte de não determinismo que corresponderia ao entrelaçamento ("*interleaving*") de execuções concorrentes.

- *Difusão instantânea*: A comunicação entre componentes é sempre realizada por um mecanismo de difusão instantânea ("*broadcasting*"). Um sinal emitido é visto no mesmo instante da sua emissão por todos seus receptores. A difusão é limitada aos instantes de reação: um sinal emitido num instante é visto como presente em todos os receptores neste mesmo instante; pode haver entretanto varias emissões e recepções de sinal em seqüência num mesmo instante.

- *Determinismo*: Contrariamente as abordagens assíncronas, onde a concorrência leva ao não determinismo, o modelo faz conviver concorrência e determinismo. O modelo síncrono é determinista, o que tem como resultado a simplificação das programações reativas e a capacidade de reproduzir seus comportamentos, simplificando testes e verificações destas.

Na abordagem síncrona, o tempo é considerado como uma entidade multiforme sendo visto como um evento externo entre outros, de características diversas do tempo físico; a noção de tempo físico é na verdade substituída pela noção de ordem e de simultaneidade entre eventos. Essa visão de tempo multiforme e a não ocorrência de "*interleaving*" facilitam em muito o entendimento e a análise dos sistemas de tempo real.

Técnicas de compilação geram a partir destes modelos síncronos – geralmente descritos na forma de linguagens – autômatos de estado finito deterministas, nos quais

o paralelismo e a comunicação expressos no formalismo inicial desaparecem; em decorrência disto, estes autômatos apresentam um grau elevado de eficiência e previsibilidade temporal. A estas características, são adicionados a simplicidade do autômato resultante em termos do número de estados e o uso de técnicas de verificação ou de prova que são facilitadas. Estes autômatos são também facilmente implementados em qualquer linguagem clássica (C, Java, etc.) ou mesmo, em lógica de circuitos seqüências booleanos.

4.3 O Estilo de Programação da Linguagem Esterel

4.3.1 Programando num estilo imperativo

Vamos a seguir a partir de um pequeno exemplo mostrar a facilidade de programação síncrona, usando o estilo imperativo da linguagem Esterel e introduzir algumas das construções de base que o caracterizam.

Considera a especificação informal:

"Emite uma saída O, tão logo que as duas entradas A e B tenham ocorrido. Reinicialize este comportamento a cada vez que ocorrer uma entrada R".

O comportamento descrito acima pode ser representado por vários formalismos – entre estes, por um autômato. Na figura 4.1, é apresentado o autômato [Ber99] que representa a especificação anterior com 4 estados, 8 transições; as entradas e saídas

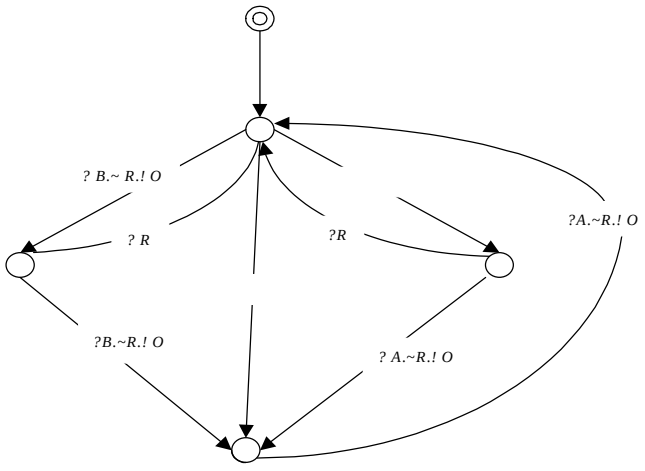


Figura 4.1: Autômato para a especificação ABRO

aparecem várias vezes (3 vezes para A, B e O e 8 vezes para R) neste autômato, tornando a representação complexa o que permite concluir na inadequação de realizar uma especificação direta na forma de um autômato. Além do mais, se o número de entradas cresce, aumenta consequentemente a complexidade na representação por causa da explosão exponencial do autômato.

A linguagem imperativa síncrona Esterel permite ao usuário escrever de forma simples e natural, sem repetição a especificação anterior no módulo ABRO:

```
module ABRO:
input A, B, R;
output O;
loop
        [await A || await B];
        emit O
each R
end module
```

Note que na especificação acima, diferente do autômato, ocorrem uma só vez as entradas e saída A, B, R e O. O aumento do número de entradas é também facilmente absorvido por este estilo de programação e o módulo ABCRO conteria apenas esta modificação:

```
loop
        [await A || await B || await C];
        emit O
each R
```

O princípio de base de um bom estilo de programação consiste em escrever cada parte da especificação apenas uma vez, evitando repetições que dificultam o entendimento e a manutenção do programa e podem ser eventuais fontes de erro. As construções da linguagem Esterel são particularmente bem adaptadas para ajudar o usuário a gerar programas para sistemas reativos que seguem este estilo de programação.

O código anterior contém algumas construções básicas da linguagem Esterel:

- *O atraso*: a construção temporal "**await**" significa a espera por um evento. Quando iniciada, corresponde a uma pausa até o evento ocorrer, instante no qual conclui a operação: "**await A**" espera o sinal **A** e termina quando este ocorre.

- *A emissão de sinal*: A emissão instantânea de sinal é realizada pela construção "**emit ...**", No exemplo acima "**emit O**" corresponde a emissão instantânea do

sinal **O**, tão logo a última entrada **A** ou **B** seja recebida. Não pode ocorrer mais de um "**emit**" por instante (a regra correspondente será vista mais tarde).

- *Seqüência*: "**p;q**" transfere imediatamente o controle a **q**, quando **p** termina.

- *Concorrência*: O operador de paralelismo "**||**" define as construções separadas pelo operador em paralelismo síncrono. A menos da intervenção de algum mecanismo de preempção ou de exceção, a construção termina quando todos seus ramos terminaram. Neste exemplo, "**await A || await B**" termina instantaneamente desde que as duas componentes concluam com as duas entradas **A** e **B** sendo recebidas.

- *aborto ou preempção*: Na construção " **loop p each R**", o corpo **p** é imediatamente inicializado e executa repetidamente até o instante de ocorrência de **R** no qual **p** é abortado e imediatamente reinicializado; esta construção é dita de *aborto ou preempção forte* pois o evento **R** é prioritário sobre o corpo em execução. No exemplo do módulo ABRO se **A**, **B** e **R** ocorrem simultaneamente, **O** não será emitido.

O estado de um sinal envolvido numa reação é *presente* ou *ausente*. O ambiente fornece o estado dos sinais de entrada, e a execução da instrução **emit** transforma o estado dos outros sinais de *ausente* para *presente*.

A comunicação em Esterel é realizada por difusão instantânea. Na entrada, a difusão instantânea é implícita para as instruções concorrentes. A difusão instantânea vale para todos os sinais e permite que processos interessados num sinal emitido (**emit O)** esperem simplesmente o mesmo através de uma instrução **await** por exemplo, sem ter que revelar suas existências ao módulo emissor e independentemente do número de processos receptores.

Após esta breve introdução ao estilo de programação da linguagem Esterel, vamos apresentar as principais construções que caracterizam este estilo, descrevendo declarações e comportamentos no contexto de pequenos exemplos ilustrativos. Uma apresentação mais detalhada da sintaxe e semântica da linguagem Esterel pode ser encontrada no Anexo C.

4.3.2 Declaração de interface

Seja a especificação de um medidor de velocidade descrito informalmente por

"*Contar o número de centímetros por segundo e difundi-lo a cada segundo como sendo o valor de um sinal* ***Velocidade***".

Os sinais de entrada do módulo medidor de velocidade são gerados a cada centímetro e a cada segundo e são representados por **Centímetro** e **Segundo** que definem cada um, uma unidade de tempo independente, caracterizando desta forma que

o tempo é multiforme no modelo síncrono. Para simplificar o exemplo, supõe-se que esses dois sinais não podem ser simultâneos (hipótese plausível devido ao ambiente de execução); a relação de exclusividade de um sinal se representa por # numa declaração **relation**. A difusão do valor da velocidade será feita por um sinal com valor, **Velocidade**, que a cada instante além do seu estado contém um valor com tipo **integer**.

O código do módulo medidor de velocidade em Esterel se escreve como:

```
module Medidor-Velocidade:
input Centímetro, Segundo;
relation Centímetro # Segundo;
output Velocidade : integer;
loop
        var Distancia := 0 : integer in
                abort
                        every Centímetro do
                                Distancia := Distancia + 1
                        end every
                when Segundo do
                        emit Velocidade (Distancia)
                end abort
        end var
end loop
end module
```

O estilo de programação adotado em Esterel prioriza o controle de atividades pelo uso da preempção. Dentro da malha infinita "**loop ... end**", a construção "**abort ... when ... do ... end abort**" interrompe seu corpo quando o sinal **Segundo** ocorre e executa a claúsula de *timeout* que segue o "**do**" que permite emitir o sinal **Velocidade** que contém o ultimo valor **Distancia**. Dentro do corpo do "**abort**" uma malha "**every ... end every**" permite incrementar a variável **Distancia** a cada sinal de entrada **Centímetro**; esta malha "**every**" difere da malha "**loop**" anteriormente apresentada por depender da ocorrência de um sinal (neste exemplo **Centímetro**) para iniciar o seu corpo. Considerando a transmissão do controle no programa e as operações suficientemente rápidas para serem vistas como instantâneas dentro da hipótese de sincronismo, a velocidade é emitida exatamente quando ocorre o *tick* indicando o segundo.

Um sinal com valor (como **Velocidade** neste exemplo) tem um único estado e um único valor a cada reação; contrariamente ao estado, o valor deste permanece igual ao da reação anterior até sofrer mudança. O ambiente determina o valor para os sinais de entrada e as instruções "**emit**" para os sinais de saída. A expressão "**?S**" permite acessar o valor corrente de um sinal **S**. O estado e o valor de um sinal podem ser difundidos instantaneamente.

A construção "**relation ...**" permite representar condições supostamente garantidas

pelo ambiente entre os sinais do tipo "**input**" (e também do tipo "**return**"). Os dois tipos de relação são de incompatibilidade ou exclusão entre os sinais ("**#**") como é o caso neste exemplo e de sincronização entre sinais ("=>"). As relações são úteis para evitar a programação de situações de menor importância e reduzir em conseqüência o tamanho do autômato gerado, facilitando a sua verificação.

Em algumas situações onde é apenas necessário a leitura de um valor a qualquer momento, sem a necessidade de gerar também um evento de entrada, pode se utilizar o sinal de entrada "**sensor**" que tem apenas o valor mas sem a presença da informação de estado. Assim como o sinal com valor, definido anteriormente, o sensor tem o seu valor lido pelo operador "**?**".

4.3.3 Declaração de variáveis

Uma variável é um objeto ao qual podem ser atribuídos valores. Uma variável é local à uma "*thread*" que corresponde ao corpo **p** de um módulo (no exemplo anterior a variável **Distancia** é incrementada a cada sinal de entrada **Centímetro** dentro da malha "**every ... end every**" do corpo do "**abort**"). O valor da variável pode ser fornecido por uma instrução de atribuição instantânea ou passado numa chamada de um procedimento externo instantâneo ou ainda passado pela instrução "**exec**" que permite a execução de tarefas externas de cálculo que tomam tempo (esta instrução será vista futuramente). Contrariamente ao sinal, uma variável pode tomar vários valores sucessivos no mesmo instante.

4.3.4 Os diferentes tipos de preempção

Foi percebida nos exemplos anteriores a importância para o estilo de programação Esterel, da construção de preempção que permite matar o seu corpo quando um evento ocorre ou um prazo se esgota. Foi apresentado até o momento um tipo de preempção chamado forte que é expressado pelas construções "**abort p when S**" e "**loop p each S**". Entretanto é interessante poder fornecer ao usuário outras opções com semânticas diferentes.

O comportamento da construção de preempção forte "**abort p when S**" é o seguinte:

- no começo da execução, **p** é imediatamente inicializado, independentemente da presença ou ausência de sinal **S**;
- se **p** termina antes da ocorrência de **S**, então a instrução "**abort**" termina neste mesmo instante;
- se **S** ocorre antes do término de **p**, então a instrução "**abort**" termina imediatamente e **p** não recebe mais o controle.

Comportamentos diferentes podem ser necessários para disponibilizar ao usuário. A possibilidade de terminação da instrução "**abort**" desde o primeiro instante mesmo sem **p** ter iniciado, é permitida pela construção "**abort p when immediate S**" que introduz a palavra chave "**immediate**".

Para permitir que o corpo **p** receba ainda o controle no instante da preempção, para uma última atividade, utiliza-se uma construção de preempção fraca: "**weak abort p when S**". O comportamento de preempção fraca é também possível através da construção "**weak abort p when immediate S**". Para o exemplo do módulo ABRO citado anteriormente, a utilização da construção "**weak abort**" no lugar de "**loop ... each**" torna possível programar, se desejado, a emissão do sinal de saída **O** mesmo no caso de uma situação de simultaneidade entre os sinais **A**, **B**, **R**. Da mesma forma, o uso de "**weak abort**" no módulo Medidor-Velocidade permite no caso de ser autorizada a ocorrência simultânea dos sinais **Centímetro** e **Segundo (**a relação de exclusividade **#** seria retirada do exemplo anterior), de levar em conta o último incremento da variável **Distancia** antes que a velocidade seja emitida; a utilização da preempção forte "**abort**" neste caso levaria a ignorar o último centímetro na instrução "**every Centímetro**" por ter sido abortada pela ocorrência de **Segundo**.

A construção em Esterel "**every S do p end**" encontrada no módulo Medidor-Velocidade é uma abreviação de:

```
await S;
loop
        abort [p; halt] when S
end loop
```

e permite realizar também uma preempção forte de seu corpo e a seguir ser atrasado (**halt** é uma primitiva que significa "espera para sempre"). Esta construção tem ainda uma forma imediata "**every immediate S do p end**" que permite levar em conta o sinal desde o primeiro instante. A solução do problema da simultaneidade de **Centímetro** e **Segundo** no módulo Medidor-Velocidade pode ser também resolvido usando um "**abort**" forte e um "**every immediate**"; neste caso o centímetro não será contado durante o segundo que esta acabando más será levado em conta no inicio do próximo segundo.

Apesar de ter discutido essas construções apenas em termos de sinal, expressões de sinais – que são na verdade expressões booleanas formadas com operadores "**not**", "**and**", e "**or**" – aplicadas sobre o estado de vários sinais podem ser utilizadas nas construções temporais "**abort**", "**every**".

A linguagem Esterel fornece ainda um outro tipo de preempção mais brando que tem um efeito de suspensão (do tipo ^Z do Unix em contraposição com o tipo ^C mais duro da construção "**abort**"): "**suspend p when <S ou expressão-sinal>**". Quando a construção "**suspend**" inicia, o corpo **p** é imediatamente iniciado. A cada instante, se o sinal **S** esta presente ou a expressão de sinal for "*true*", o corpo **p** se suspende e é

congelado no estado no qual se encontrava no momento da suspensão; em caso contrario, o corpo **p** é executado neste instante. Como nas outras construções de preempção, é necessário usar a palavra chave "**immediate**" na construção "**suspend p when immediate <S ou expressão-sinal>**" para poder testar a eventual presença do sinal **S** ou a expressão de sinal no instante inicial.

4.3.5 Mecanismo de exceção

O mecanismo de exceção da linguagem Esterel é implementado pela construção "**trap T in p end trap**" que permite programar um ponto de saída para o corpo **p**. O corpo **p** é imediatamente executado quando a construção "**trap**" inicia. A sua execução continuará até o término de **p** ou até a execução de uma construção "**exit T**" introduzida no corpo **p**. O término do corpo **p** leva ao término da construção "**trap**". A saída por um "**exit**" leva o "**trap**" a terminar imediatamente, abortando **p** por preempção fraca, do tipo "**weak abort**.

No caso da existência de um tratador de exceção, o controle é imediatamente transferido para o mesmo, após a execução do "**exit**". A ativação do tratamento da exceção é realizado pela construção "**handle**", permitindo o inicio imediato de **q** após o corpo **p** ter sido abortado pelo "**exit**" do "**trap**":

```
trap T in
        p
handle T do
        q
end trap
```

4.3.6 Testes de presença

Além de manusear sinais a partir de construções de preempção, é possível realizar testes de presença de sinais com a construção "**present S else p end present**". Este teste permite decidir entre várias ramificações de programa, em função do valor instantâneo do sinal **S**. Se o sinal **S** está presente, a construção "**present**" termina a seguir, caso contrário a ramificação "**else**" é imediatamente iniciada; a claúsula "**then**" é omitida mas poderia ser usada no lugar de "**else**" se o teste fosse sobre a ausência de sinal ("**not S**").

Para aumentar o poder de expressão, expressões de sinais $\mathbf{e_i}$ e construções condicionais múltiplas ("**case**") podem ser utilizadas em testes de presença:

```
present
    case e1 do p1
    case e2 do p2
    case e3 do p3
    else q
end present
```

4.3.7 Módulo

O módulo é a unidade básica para construir programas Esterel. Ele tem um nome, uma declaração de interface e um corpo que é executável.

```
module <nome> :
        <declaração de interface>
        <corpo>
end module
```

O estilo de programação Esterel permite construir um módulo com nomes padrão para os sinais de entrada e saída. O módulo pode utilizar submódulos que são módulos instanciados pela construção executável "**run**" e cujas entradas e saídas podem ser renomeadas explicitamente usando o símbolo "/". O comportamento do submódulo é o mesmo do módulo, substituindo os nomes padrão (à direita de "/") pelos nomes reais dos sinais (situados à esquerda de "/"). Em nenhum caso é permitido recursividade sobre a instanciação.

Uma forma alternativa de programar o módulo ABCRO a partir do uso do módulo ABRO como módulo genérico é apresentada a seguir como exemplo:

```
module ABCRO:
input A, B, C, R;
output O;
signal AB in
        run ABRO [signal AB / O]
||
        run ABRO [signal AB / A , C / B]
end signal
end module
```

O submódulo ABRO [signal AB/O] se comporta como "loop [await A || await B]; emit AB each R"; o submódulo ABRO [signal AB/A, C/B] como "loop [await AB || await C]; emit O each R"; a difusão instantânea do sinal AB nos dois submódulos paralelos torna o comportamento do módulo ABCRO equivalente ao comportamento já descrito anteriormente "loop [awaitA || await B || await C]; emit O each R". Constata-se ainda que o modelo de programação síncrona fornece como resultado adicional interessante, a reinicialização simultânea dos dois submódulos pelo sinal R.

A declaração de interface do módulo define os objetos que este importa ou exporta: *objetos de dados* (tipos, constantes, funções, procedimentos, tarefas) declarados de forma abstrata em Esterel e implementados externamente e *objetos de interface reativa*: os sinais e sensores. Todos os dados são globais ao programa. Cada dado usado num módulo, deve ser declarado nele.

4.3.8 O conceito de tempo no modelo de programação

No modelo de programação síncrono que suporta a linguagem Esterel, a noção de tempo físico não existe; o tempo físico é visto como um sinal entre outros. Qualquer sinal pode ser considerado para definir uma unidade de tempo independente. O tempo é dito multiforme, i.e. qualquer sinal repetido pode ser considerado como definindo sua própria medida de tempo. O estilo de programação Esterel requer o reconhecimento de todas as unidades de tempo da aplicação e o entendimento de como estas se relacionam entre si. Para representar esta noção, poderia se imaginar uma representação gráfica na forma de vários eixos de tempo com unidades diferentes correspondentes aos diversos sinais repetidos. No módulo Medidor-Velocidade, as unidades de tempo são **Centímetro** e **Segundo** com eixos de tempo próprios; neste caso simples, a relação entre os mesmos permite que se calcule a velocidade.

4.4 Um exemplo ilustrativo do estilo de programação

O estilo de programação é ilustrado a seguir por um pequeno exemplo que descreve o treinamento de um corredor e cuja especificação é:

"*Cada manhã, o corredor faz um número fixo de voltas num estádio. A cada volta, ele corre devagar durante 100 metros, depois ele pula a cada passo durante 15 segundos e termina a volta correndo rápido*".

Num primeiro tempo, a partir da especificação informal anterior, determina-se os sinais que corresponderão as unidades de tempo. Os sinais de entrada são **Manha**, **Volta**, **Metro**, **Passo** e **Segundo**. Os sinais de saída são **Correr-Devagar**, **Pular** e **Correr-Rápido**. Relações entre sinais são estabelecidas: **Manha e Segundo** são sincronizados bem como **Volta** e **Metro.** Os sinais **Correr-Devagar** e **Correr-Rápido**

serão emitidos de forma contínua; o sinal de saída **Pular** será emitido em reação ao sinal de entrada **Passo**. Desta forma fica definida a interface do módulo **Corredor** cujo o código será apresentado a seguir. As relações de sincronização entre sinais são expressas no programa pelo símbolo "=>".

O corpo do programa é uma tradução natural da especificação informal seguindo o estilo de programação apresentado anteriormente cujo o princípio fundamental consiste em controlar as atividades a partir das construções de preempção. O uso de preempções aninhadas é uma forma simples e natural para expressar prioridades entre estas. Uma linha de conduta para uma boa programação consiste a partir de uma primeira leitura da especificação, em construir inicialmente a estrutura de preempções aninhadas, usando a indentação das construções de preempção para visualizar o aninhamento e depois, a partir de uma releitura da especificação, em introduzir as outras construções no corpo do programa. O código em Esterel do módulo Corredor é o seguinte:

```
module Corredor
constant Número-Voltas: integer;
input Manha, Volta, Metro, Passo, Segundo;
relation Manha => Segundo,
  Volta => Metro;
output Correr-Devagar, Pular, Correr-Rápido;
every Manha do
        abort
                abort
                        abort
                                sustain Correr-Devagar
                        when 100 Metro;
                        abort
                                every Passo do
                                        emit Pular
                                end every
                        when 15 Segundo;
                        sustain Correr-Rápido
                when Volta
        when Número-Voltas Volta
end every
end module
```

Para emitir um sinal de forma continua (caso de **Correr-Devagar** e **Correr-Rápido**), utiliza-se a construção "**sustain S**"; a construção fica ativa para sempre e emite **S** a cada instante.

Destaca-se ainda que poderia ser utilizado também a construção "**loop ... each ...**" em algumas situações como por exemplo em "**loop ... each Volta**" no lugar de "**abort ... when Volta**".

Nota-se que se a volta for inferior a 100 metros, o corredor apenas correrá devagar e se esta for mais curta que 100 metros mais 15 segundos, o corredor nunca correra rapidamente; também não é necessário que cada volta seja dimensionada de forma igual.

Ao acrescentamos à especificação anterior do treinamento do corredor o seguinte:

"*Durante a fase na qual o corredor pula, o coração dele é monitorado e em caso de alguma anomalia ser constatada, o corredor parará e retornará ao vestiário*".

O novo código que leva em conta esta especificação adicional toma a seguinte forma:

```
every Manha do
    trap Anomalia in
        abort
            abort
                abort
                    abort
                        sustain Correr-Devagar
                    when 100 Metro;
                    abort
                        every Passo do
                            emit Pular
                        end every
                    ||
                        <Monitorar-Coração>
                    when 15 Segundo;
                    sustain Correr-Rapido
                when Volta
            when Número-Voltas Volta
    handle Anomalia do
        <Voltar-Vestiário>
    end trap
end every
```

A atividade **<Monitorar-Coração>** não é descrita aqui mas deverá obrigatoriamente conter uma construção de levantamento de exceção do tipo "**exit Anomalia**" quando esta for constatada; a atividade de tratamento de exceção **<Voltar-Vestiário>** também não é objeto da nossa descrição.

4.5 A assíncronia na linguagem Esterel: a execução de tarefas externas

Procedimentos externos chamados pela construção "**call**" são considerados como instantâneos. Entretanto, é possível controlar ainda a execução de tarefas externas que levam tempo usando o mecanismo "**exec**". Essas tarefas se comportam como procedimentos a serem executados de forma assíncrona com o programa Esterel que terá apenas uma visão lógica destas, se interessando apenas pelo inicio ou fim das mesmas ou ainda, pela suspensão ou preempção desta tarefas através de construções

Esterel. A forma de assincronismo assim introduzida é restrita para permitir a sincronização com a tarefa apenas quando do término da mesma. Estas tarefas não são limitadas ao seu sentido computacional mas podem ser também atividades de um objeto real de natureza física.

A construção "**exec Task (<parâmetros-referência>) (<parâmetros-valores>) return R**" é que permite a execução de uma tarefa externa. Nesta construção, **R** é um sinal de retorno que se restringe a um único "**exec**". Como qualquer sinal de entrada no programa Esterel, cada **R** deve ser declarado na interface de sinal do módulo, utilizando a palavra chave "**return**" no lugar de "**input**". No inicio da construção "**exec**", o ambiente é sinalizado instantaneamente para que a tarefa inicie; o programa Esterel continue a reagir aos sinais de entrada, exceto na "thread" que disparou o "**exec**" que deve esperar o término da tarefa correspondente. Na recepção do retorno **R**, são atualizados instantaneamente os argumentos de referência em todo o programa Esterel, com os valores retornados e a construção "**exec**" termina instantaneamente. Uma operação "**exec**" pode ser suspensa ("**suspend**") ou abortada ("**abort**", "**weak abort**" ou "**trap**") durante sua execução e, em qualquer destas situações, a tarefa externa receberá a sinalização correspondente.

No caso da preempção fraca:

```
weak abort
        exec TASK (X) (...) return R;
when I
```

se **R** ocorre antes ou simultaneamente com **I**, **X** é atualizado e a construção "**weak abort**" termina; se **I** ocorre antes de **R**, a execução de **TASK** e consequentemente da tarefa externa é abortada e **X** não será atualizado.

No caso da preempção forte:

```
abort
        exec TASK (X) (...) return R;
when I
```

se **R** ocorre antes de **I**, **X** é atualizado e a construção "**abort**" termina; se **I** ocorre antes de **R**, a execução de **TASK** e consequentemente da tarefa externa é abortada e **X** não será atualizado; se **R** e **I** ocorre simultaneamente, a construção "**abort**" termina e **X** não é atualizado porque, apesar da tarefa ter sido concluída, o corpo do "**abort**" não recebe o controle.

Se for necessário saber se a tarefa externa terminou num dado instante, o teste "**present R then ... else ... end present**" quando usado na claúsula "**do**" do "**when ...**"

fornece esta informação.

No caso da suspensão:

```
suspend
        exec TASK (X) (...) return R;
when S
```

se **S** ocorrer após o instante inicial, a construção "**exec**" é suspensa e um sinal de suspensão especifico para cada implementação de tarefa é enviado ao ambiente. A terminação de "**exec**" pode ocorrer somente quando a construção é ativa; se **R** e **S** ocorrem simultaneamente, **R** não provocará o fim de "**exec**" e sua ocorrência será perdida. É possível também neste caso, introduzir um teste de presença de um sinal de retorno.

Quando for necessário o controle de várias tarefas simultaneamente, utiliza-se:

```
exec
        case T1 (...) (...) return R1 do p1
        ...
        case Tn (...) (...) return Rn do pn
end exec
```

Quando um "**exec**" múltiplo inicia, todas as tarefas são disparadas simultaneamente; quando pelo menos um sinal de retorno ocorre, a construção "**exec**" termina instantaneamente, abortando as outras tarefas, atualizando apenas o argumento referência da tarefa que terminou; em caso de preempção ou de suspensão todas as tarefas são abortadas simultaneamente.

Numa construção "**exec**" embutida numa malha:

```
loop
        exec TASK (X) (...) return R;
each I
```

se **I** ocorre antes da finalização da tarefa, o programa Esterel sinaliza ao ambiente o aborto da instância corrente da tarefa e nova instância desta é iniciada. Nunca pode haver duas instâncias de uma tarefa disparadas por um mesmo "**exec**" que possam ser ativas no mesmo instante – a não ser em situações onde a construção "**exec**" é abortada e reinicializada, instantaneamente.

4.6 O Ambiente de Ferramentas Esterel

A linguagem de programação Esterel para sistemas reativos vem sendo desenvolvida desde 1982 por equipes do INRIA e da Ecole des Mines de Sophia-Antipolis (France) e se encontra atualmente na sua versão 5 . Seu compilador (atualmente V5.21) esta sendo disponibilizado gratuitamente no site http://www-sop.inria.fr/meije/esterel/ para arquiteturas Solaris, Linux, AIX, OSF1 e Windows NT.

Esta versão do compilador usa uma técnica de compilação baseada em Diagramas de Decisão Binários (BDD) e permite gerar uma implementação de software de um programa reativo na forma de uma máquina de estados finita (FSM) ou uma implementação de hardware na forma de circuitos, obtidos a partir de sistemas de equações booleanos. O código gerado (por exemplo em C) pode ser embutido como um núcleo reativo num programa maior que trata da interface do núcleo e processa os dados da aplicação. Da mesma forma, nas implementações de hardware, a lista de "*gates*" gerada pode ser embutida em circuitos maiores. Além deste compilador, existe um conjunto de ferramentas disponíveis no mesmo site para desenvolver e para verificar programas em Esterel.

Uma das grandes vantagens do modelo usado em Esterel é de favorecer uma abordagem do tipo "o que você prova é o que você execute" (WYPIWYE "*What You Prove Is What You Execute*") [Ber89]. Este aspecto é determinante para que o modelo síncrono se diferencie de grande parte das outras abordagens que, para o desenvolvimento de programas, necessitam de um processo de tradução para passar da especificação validada ao programa implementado. Nesta tradução, são introduzidos detalhes de implementação que, geralmente, são fonte de erros ou de modificações no comportamento já verificado.

No que se refere a validação em linguagens síncronas, as principais propriedades a serem verificadas para os sistemas de tempo real são: "vivacidade" ("*liveness*") que pode ser entendida como "algo útil deverá acontecer em algum momento"; e "segurança" ("*safety*") que corresponde a "algo ruim nunca deverá acontecer" ou "algo útil deverá acontecer antes de algum tempo" no caso dos sistemas de tempo real.

A verificação consiste em comparar um programa com as especificações desejadas de algum aspecto ou da totalidade das mesmas. A verificação das propriedades citadas se faz sobre a estrutura de controle do programa, os dados não influem diretamente. Duas abordagens de verificação podem ser diferenciadas, conforme as especificações se apresentem no mesmo formalismo que o programa (por exemplo, autômato) ou em formalismos diferentes (por exemplo, autômato para o programa e lógica temporal para as especificações). No caso da primeira abordagem, pode se utilizar duas técnicas diferentes: a de verificação por equivalência e a de verificação por observadores.

A técnica de verificação por equivalência usa a noção de bisimulação fraca que foi definida por Park e usado por Milner [Arn94]. A bisimulação é uma relação binaria entre estados de dois sistemas de transições (no caso similares aos autômatos) que

define a equivalência entre estes. A verificação da equivalência consiste em verificar a existência desta relação de bisimulação. O primeiro passo consiste em distinguir os sinais relevantes para o aspecto da especificação que se pretende verificar, dos irrelevantes que passam a ser considerados como eventos internos. O segundo passo consiste em reduzir na sua forma mínima o autômato na forma obtida anteriormente, do ponto de vista da bisimulação fraca. Este autômato reduzido contém poucos estados e transições, podendo ser facilmente verificado as propriedades por simples leitura ou a sua equivalência com um autômato das especificações.

A técnica de verificação por observadores consiste em construir um observador **O** que é um outro programa reativo que expressa a propriedade a ser verificada e a colocá-lo em paralelismo síncrono com o programa **P** a verificar e com outro programa reativo **E** que representa o ambiente e gera seqüências de entrada para os dois anteriores. Como o papel do observador **O** é de ver o comportamento do programa **P**, ele tem ainda como entrada, as saídas do programa **P**. Técnicas de analise de alcançabilidade para autômatos permitem verificar o programa, detectando eventuais diferenças entre o comportamento do programa e aquele descrito pelo observador. Qualquer linguagem síncrona pode ser usada para programar o observador **O** e o ambiente **E**.

A verificação pela abordagem da lógica temporal consiste em verificar se formulas de lógica temporal que representam as propriedades desejadas são satisfeitas ou não pelo programa. Uma das formas de construir um verificador neste caso consiste em construir também um observador com as propriedades expressas em lógica temporal e compilado como programa Esterel. Existem na literatura, outras formas de expressar as formulas de lógica temporal que resultaram em varias ferramentas de verificação baseadas nesta técnica.

As ferramentas incluídas no ambiente de desenvolvimento Esterel permitem os três tipos de verificação descritas anteriormente.

O ambiente de desenvolvimento e verificação Esterel, **Xeve** disponibilizado em http://www-sop.inria.fr/meije/verification/Xeve/ é um ambiente com interface gráfica que permite a análise e a verificação de programas Esterel baseado na representação em máquinas de estados finitos. As ferramentas de **Xeve** são:

- **Hurricane** (disponível para Solaris e Linux) que é um verificador de modelos (*model checking*) sobre os programas Esterel, baseados em fórmulas de lógica temporal. As fórmulas de lógica temporal são transformadas em observadores Esterel usando um tradutor tl2strl. Esta ferramenta gera um novo programa Esterel obtido a partir do programa principal e das fórmulas e onde, as entradas são as do programa principal e as saídas são as do programa principal e as obtidas das fórmulas. A ferramenta verifica o estado dos sinais de saída e o resultado obtido indica para cada saída se a mesma pode ser emitida ou não.
- **fc2symbmin** que analisa as máquinas de estado finitos geradas pelo compilador Esterel e descritas no formato FC2 – formato definido pela

equipe de pesquisa deste projeto para facilitar a interface com outros software de verificação –. Esta análise inclui:

- a minimização de FSMs (máquinas de estados finitos) obtida a partir da noção de bisimulação dos autômatos reativos obtidos do programa Esterel,
- o diagnóstico através de observadores que são programas inicializados em paralelo com o programa principal, testando seus sinais, interagindo com ele e gerando sinais de falha ou sucesso; esses observadores podem ainda gerar sinais para simular o ambiente Esterel,
- o produto síncrono de FSMs para implementar o operador paralelo síncrona de Esterel,
- a ocultação de sinais para reduzir o tamanho da FSM e a complexidade da análise, e
- a verificação de equivalência que determina se dois autômatos podem se comportar da mesma forma ou não tendo o mesmo ambiente.

- **Atg** disponível em http://www-sop.inria.fr/meije/verification/index.html é um editor gráfico de autômato que permite a visualização da máquina de estado finitos minimizada.
- **Xes** que é um simulador gráfico habitualmente distribuído com o compilador para lhe servir de depurador (“debugger”) e que permite executar seqüências de execução como por exemplo as extraídas do diagnostico de verificação de modelo (*model checking*).

As figuras 4.2 a 4.4 apresentam algumas das telas do ambiente Xeve.

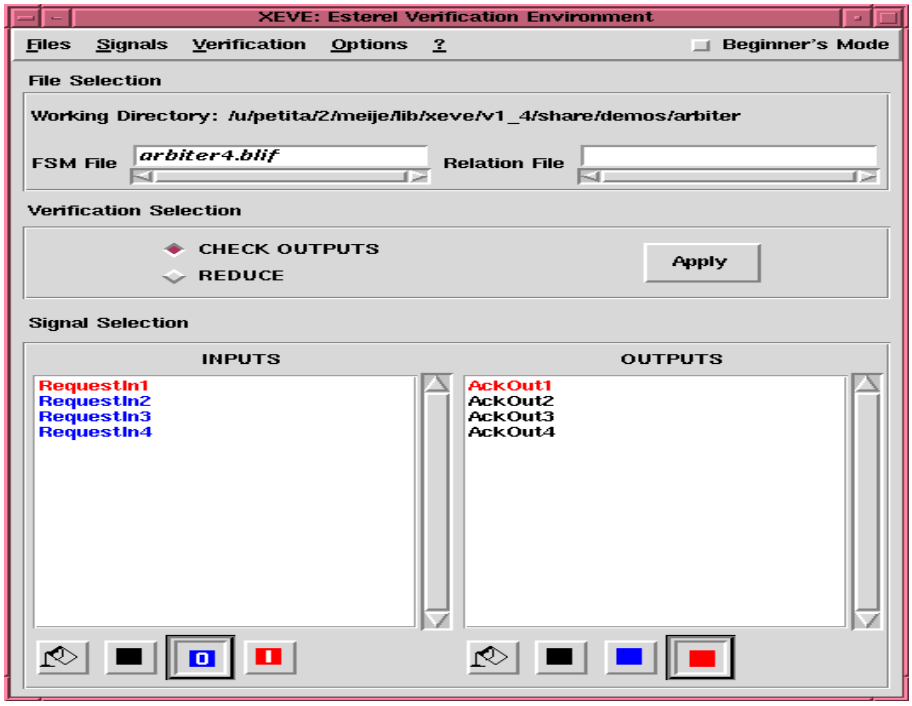


Figura 4.2 - Tela Principal do Ambiente XEVE

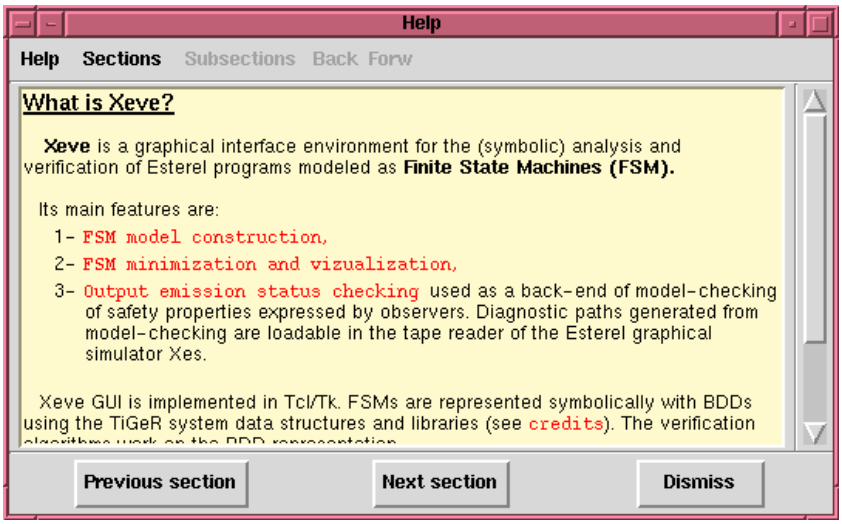


Figura 4.3 - Tela de Ajuda do Ambiente XEVE

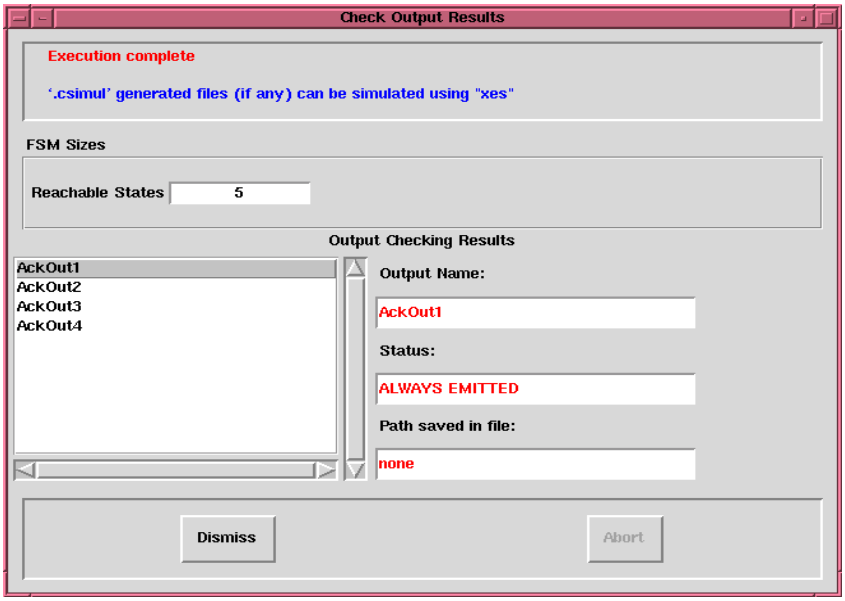


Figura 4.4 - Tela de Resultados do Ambiente XEVE

4.7 Implementações de programas em Esterel

Um programa em Esterel pode ser implementado de duas formas diferentes: por software usando um autômato ou por hardware usando circuitos booleanos seqüenciais. Neste livro, apresentaremos somente a primeira forma de implementação; o leitor que tiver interesse na segunda forma de implementação, deverá recorrer as seguintes referencias básicas [Ber92], [STB96].

- **Implementação usando um autômato**

Uma das características mais interessante do modelo síncrono e da linguagem Esterel é a capacidade de produzir um autômato (ou máquina de estados finitos), construído pelo compilador a partir das regras da sua semântica de execução que à cada evento de entrada, associa uma transição vindo de um estado.

O autômato resultante da compilação é determinista: o paralelismo e as comunicações de sinal em código Esterel desaparecem e são seqüencializados dentro de cada transição interna, produzindo um código seqüencial. Este tipo de compilação em autômatos oferece vantagens do ponto de vista da eficiência e em relação à previsibilidade, tão necessária em sistemas de tempo real.

A principal vantagem reside na eficiência na execução. Os sinais locais usados para comunicação interna entre construções desaparecem no autômato, como os nós não-terminais intermediários desaparecem durante a geração de “parser” numa compilação; os sinais locais podem em conseqüência ser considerados como tendo verdadeiro atraso zero em tempo de execução. Além do mais, como não há mais nenhum paralelismo, não há custo adicional em tempo resultando do gerenciamento em tempo de execução de tarefas ou da comunicação e sincronização de tarefas em tempo de execução, em contraposição com o que ocorre com o uso de outras linguagens concorrentes que não suportam o modelo síncrono e que necessitam de usar um suporte para gerenciamento e comunicação de tarefas, com o custo adicional e a não-previsibilidade que este proporciona.

Como vantagem adicional desta compilação em autômato, destaca-se a previsibilidade do tempo de transição máxima de um autômato, de especial relevância quando se trata de Sistemas de Tempo Real.

Entretanto a desvantagem de um autômato esta associada ao seu tamanho. Aplicações que resultam em autômatos relativamente pequenos como em protocolos, interface homem-máquina, controle de processos simples são facilmente manuseáveis, o que não é o caso em algumas aplicações mais complexas na qual a explosão de estados pode se tornar relevante.

No ambiente de desenvolvimento para a linguagem Esterel, o autômato é produzido num formato de saída intermediário, chamado OC que é comum à linguagem Lustre já citada e que pode ser traduzido numa linguagem alvo de uso geral tal como C, Ada, Java.

4.8 Discussão sobre o modelo e a linguagem

A abordagem adotada para tratar a reatividade nos modelos síncronos e em particular em Esterel se baseia na hipótese de sincronismo que determina a instantaneidade da reação. A cada reação, o sinal vindo do ambiente ou de um componente se encontra presente em todos os componentes paralelos. Esta abordagem pode levar a situações indesejáveis nas quais um sinal pode ser alterado devido ao teste sobre o mesmo sinal, resultando num problema de causalidade e consequentemente em programas incorretos. A seguir são apresentados alguns exemplos de situações que geram programas incorretos e que necessitam da definição de uma semântica formal de execução para resolve-los.

- **Causalidade:**

A expressividade da linguagem síncrona Esterel possibilita a construção de programas não-causais.

A capacidade de comunicação através de sinais difundidos instantaneamente, pode levar alguns programas a desrespeitar a causalidade. Por exemplo, o programa que segue:

```
present S else emit S end
```

corresponde a testar a ausência de um sinal **S**; se o mesmo se fizer ausente, o sinal é emitido e então presente; se estiver presente, o sinal não é emitido e consequentemente ausente. Este programa apresenta uma incoerência com o principio de comunicação com difusão e deve ser absolutamente rejeitado em tempo de compilação. O circuito lógico equivalente a este programa "**S = not S**" mostra facilmente o absurdo deste.

```
present S then emit S end
```

Da mesma forma, o programa é não determinista, pois **S** é presente se e somente se **S** for emitido, o que não determina seu estado. O circuito lógico equivalente a este programa "**S = S**" também não faz sentido.

O programa seguinte:

```
present S1 else emit S2 end
||
present S2 else emit S1 end
```

expressa as seguintes situações: $\mathbf{S_1}$ ausente e $\mathbf{S_2}$ presente; $\mathbf{S_2}$ ausente e $\mathbf{S_1}$ presente. Este programa fere o determinismo do modelo síncrono porque essas duas situações não podem ocorrer simultaneamente e, portanto, também deve ser rejeitado.

Podem ainda existir situações nas quais apesar de ser construído a partir de componentes paralelos sem problemas de causalidade, o sistema total pode apresentar incoerências do ponto de vista da causalidade. Em [BoS96] é apresentado o seguinte exemplo desta situação. Sejam dois programas p_1 e p_2 respectivamente que apresentam o mesmo comportamento (U emitido e T é emitido se S é presente):

```
present S then emit T end; emit U
```

e

```
emit U; present S then emit T end
```

o sistema total que consiste em colocar cada um destes programas em paralelo com um terceiro programa p_3 cujo código é:

```
present U then emit S end
```

apresenta uma solução correta para o caso $p_2 \| p_3$ e uma não-causalidade para $p_1 \| p_3$.

- **Malhas instantâneas:**

Uma outra situação indesejável que pode levar a uma situação sem solução é o caso das malhas instantâneas cujo corpo termina no instante onde é executado pela primeira vez; por exemplo na instrução:

```
loop x := x+1 end
```

- **Sinais com valores:**

Algumas situações envolvendo sinais com valores geram incoerências no programa. Por exemplo a instrução

```
emit S(?S + 1)
```

indica a emissão de um sinal **S** cujo o valor é fornecido por **?S + 1** que corresponde ao valor corrente de **S** acrescido de **1**; o efeito obtido é equivalente a uma realimentação positiva e é inaceitável num programa Esterel.

Todas essas situações paradoxais devem ser evitadas pelo programador ou detectadas e rejeitadas a partir de uma analise estática dos programas pelos compiladores da linguagem Esterel . Uma forma fácil consiste em proibir a auto-dependência estática dos sinais, da mesma forma que se exige que circuitos lógicos sejam acíclicos; mas esta forma se apresenta como restritiva para algumas situações raras nas quais pode se desejar programas com dependências cíclicas como no caso do mecanismo de arbitro de barramento simétrico numa rede local, conforme descrito em [Ber 99].

4.9 Conclusão

Este capítulo foi escrita a partir da bibliografia seguinte: [Ber89], [BeB91], [GLM94], [BoS96] na apresentação e discussão da abordagem síncrona e [BoS91], [BeG92], [Ber98], [Ber99] na apresentação da linguagem síncrona Esterel.

A abordagem síncrona apresentada neste capítulo adota o modelo síncrono que, basicamente, assume reações instantâneas a eventos externos. Esta hipótese de reações instantâneas pode ser entendida como qualquer tempo de resposta menor que o tempo mínimo entre eventos vindos do ambiente. Muito aplicações de tempo real sustentam esta hipótese como verdadeira.

A utilização de uma linguagem síncrona, neste livro Esterel, leva à implementações

eficientes dos sistemas de tempo real baseadas em autômatos ou em circuitos booleanos. Essas implementações são facilmente validadas segundo o principio “o que você prova é o que você execute” descrito anteriormente.

Sistemas embutidos, protocolos de comunicação, “*drivers*” para periféricos, sistemas de supervisão e controle, e interfaces homem-máquina são aplicações ou partes de aplicações mais complexas para as quais o uso da abordagem síncrona é particularmente adaptada. No capitulo 5, a partir de um exemplo simples será apresentada uma metodologia de programação baseada na abordagem síncrona e seguindo o estilo de programação da linguagem Esterel.

Capítulo 5

Aplicação das Abordagens Assíncrona e Síncrona

O objetivo deste capítulo é mostrar, através de exemplos mais significativos, como as técnicas apresentadas ao longo deste livro podem ser aplicadas na prática. A seção 5.1 trata de uma aplicação do tipo "veículo com navegação autônoma", a qual será tratada segundo a abordagem assíncrona, isto é, utilizando a análise de escalonabilidade, um sistema operacional de tempo real e uma linguagem de programação como C ou C++. A seção 5.2 apresenta uma aplicação do tipo "sistema de controle", a qual em função das suas características, será tratada segundo a abordagem síncrona, utilizando a linguagem Esterel. A seção 5.3 contém uma discussão sobre as duas abordagens e elementos para a comparação e a melhor utilização destas.

É importante observar que a limitação de espaço impossibilita uma descrição completa de tais aplicações. Entretanto, mesmo com a descrição simplificada apresentada aqui pretende-se permitir ao leitor uma melhor compreensão de como as técnicas apresentadas neste livro podem ser usadas em aplicações reais.

5.1 Aplicação com Abordagem Assíncrona

Esta seção descreve um sistema com requisitos de tempo real, o qual será utilizado para exemplificar a aplicação das técnicas apresentadas nos capítulos 2 e 3 deste livro. O sistema descrito consiste de um veículo com navegação autônoma (AGV – *"Automatically Guided Vehicle"*) bastante simplificado. O objetivo não é descrever o projeto completo de tal aplicação (tarefa para vários livros do tamanho deste), mas sim ilustrar aquilo que foi discutido. A literatura sobre o tema é vasta. Por exemplo, em [POS97] e [BCH95] que inspiraram o problema a ser tratado a seguir, são descritas soluções completas para um veículo submarino e um veículo para realizar inspeções em depósitos de lixo tóxico, respectivamente.

5.1.1 Descrição do Problema

Um veículo com navegação autônoma deve ser capaz de planejar e executar a tarefa especificada. Veículos com navegação autônoma são usados, por exemplo, para

transporte de material no chão da fábrica, para a inspeção em áreas de risco ou depósitos de material tóxico. Eles podem ser usados como trator para puxar algo, no transporte de material ou ainda como base móvel para um equipamento. Após a determinação do plano a ser seguido, sua execução é iniciada. Um sistema de sensores é usado para dirigir o veículo. Os componentes básicos de um AGV incluem:

- Partes mecânicas e sistema de propulsão;
- Sistemas eletrônicos;
- Sistema sensorial, com sensores internos e externos;
- Módulo de planejamento e navegação;
- Representação interna do mundo em volta do veículo.

O AGV considerado será usado para o transporte de material. Ele tipicamente desloca-se até o local onde deverá pegar o material a ser transportado (navegação), posiciona-se de forma a realizar a carga do material (atracagem), coloca o material em seu compartimento de carga (carga), manobra para sair do local de carregamento (desatracagem), desloca-se até o local destino da carga (navegação), manobra para entrar no local de descarga (atracagem) e finalmente coloca no material transportado no seu destino (descarga). É possível observar que existem 3 fases distintas na operação do AGV, as quais resultam em 3 modos de operação independentes:

- Carregamento e descarregamento de material;
- Planejamento da rota e navegação;
- Manobra para atracar e desatracar de algum tipo de estacionamento.

Com respeito ao modo de operação "planejamento da rota e navegação", é possível dividir o trabalho a ser feito em 3 níveis [RNS93]:

- Nível de planejamento;
- Nível de navegação;
- Nível de pilotagem.

Neste livro vamos considerar exclusivamente o nível de navegação. A navegação parte do plano original desenvolvido pelo nível de planejamento e, considerando as condições locais informadas pelos sensores, dirige o veículo. Diferentes tipos de sensores podem ser usados, tais como bússula, medidor de inclinação, contadores de pulsos nas rodas, cameras de vídeo, sensores de proximidade, sensores de contato, etc. A navegação pode ainda ser auxiliada por emissores de infra-vermelho ou rádio-freqüência colocados ao longo da fábrica.

Desvios da rota inicialmente traçada podem ser necessários em função de obstáculos e para evitar colisões. Quanto maior o número de objetos móveis no chão da fábrica mais complexa torna-se a navegação. No caso da rota original tornar-se inviável, em função de um obstáculo que não pode ser contornado, uma nova etapa de planejamento torna-se necessária. O objetivo da navegação é determinar a direção e velocidade desejada para o veículo, a qual é passada para o nível de pilotagem, responsável pelas operações de controle elementares. A pilotagem utiliza laços de

controle para garantir a correta implementação das decisões de direção e velocidade recebidas do nível de navegação.

No projeto em questão a navegação utiliza uma combinação de três informações: monitoração do movimento das rodas, observação de pontos de referência no ambiente e conhecimento prévio de um mapa básico do ambiente. O movimento das rodas é medido e marcado sobre o mapa do ambiente, o qual foi carregado previamente. A estimativa da posição do veículo a partir do movimento das rodas torna-se mais imprecisa na medida que a distância percorrida aumenta. O erro retorna para zero quando a observação de pontos de referência no ambiente permite determinar com exatidão a posição do veículo. Além disto, obstáculos podem ser observados e, neste caso, o mapa do ambiente é atualizado. A partir da posição, velocidade e direção atuais do veículo, e do plano a ser seguido, são definidas a nova direção e velocidade a serem implementadas. A figura 5.1 apresenta em forma de diagrama de fluxo de dados as funções do nível de navegação.

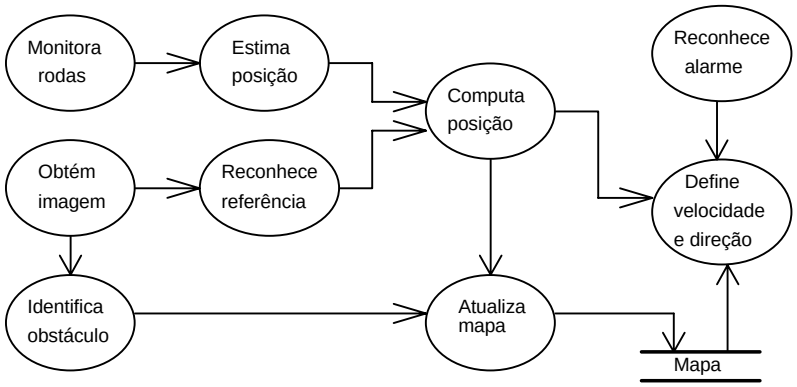


Figura 5.1 - Diagrama do nível de navegação.

A física do veículo impõe restrições temporais para a navegação, para que esta seja capaz de evitar colisões e realizar as manobras com segurança. A definição de velocidade e direção tem o período definido pela física do veículo, sua velocidade máxima, caraterísticas do terreno, etc. Neste caso os engenheiros de controle definiram 100ms como período máximo. A estimativa da posição a partir da monitoração das rodas deve ser feita a cada 100 ms. O reconhecimento de referências não necessita ser tão freqüente, até porque o algoritmo usado é complexo. Uma tentativa de reconhecimento de referências a cada 1300 milisegundos é considerado satisfatório. Por outro lado, a identificação de obstáculos deve ser feita a cada 500 ms. Esta freqüência permite que o veículo, mesmo em velocidade máxima, desvie de um obstáculo que surge repentinamente, por exemplo um outro veículo. Além das funções normais, um auto-diagnóstico deve ser executado no prazo de 20 ms sempre que um sinal de comando for recebido, por exemplo, de uma estação de supervisão.

No caso do AGV descrito, os períodos de funções que não foram definidos explicitamente pela especificação podem ser facilmente deduzidos. Como obstáculos são identificados a partir das imagens, a obtenção de imagens também deve ser feita a cada 500ms. Como a atualização do mapa acontece em função da identificação de obstáculos, ela também deverá ser executada a cada 500ms.

5.1.2 Definição das Tarefas

Uma vez especificado o comportamento temporal desejado conforme descrição anterior, o projetista de software define as tarefas e os tempos associados a partir da especificação, conforme a função de cada uma. O fato de duas tarefas trocarem mensagens ou acessarem estruturas de dados compartilhadas podem criar situações de precedência ou exclusão mútua. Finalmente, a codificação das tarefas e o processador escolhido vão definir os tempos máximos de execução de cada tarefa. Em resumo, temos tipicamente que:

- Períodos e *"deadlines"* são definidos pelos algoritmos de controle empregados;
- Relações de precedência e exclusão mútua são definidos pelo projeto do software;
- Tempo máximo de execução é definido pela programação das tarefas e escolha do processador.

Neste livro, adota-se valores arbitrários para os tempos de execução das tarefas. Entretanto, estes tempos assim como as demais características das tarefas, foram em parte baseadas em uma descrição semelhante encontrada em [But97].

A partir de um estudo cuidadoso da especificação é possível perceber que todas as funções são periódicas com deadline igual ao período, com exceção do tratamento de sinais de alarme, os quais podem acontecer a qualquer instante. Serão empregadas 7 tarefas de software para implementar o nível de navegação do AGV. É suposto que elas devem executar no mesmo processador.

- COMPUTA_POSIÇÃO (C_P)

Tarefa periódica que lê os sensores acoplados às rodas do veículo e estima a posição atual do veículo, em função da posição anterior e do comportamento das rodas, com período de 100ms e tempo máximo de execução de 20ms. Se alguma referência tiver sido reconhecida desde a última execução desta tarefa, então esta informação é também utilizada para computar a posição atual do veículo.

- LÊ_IMAGEM (L_I)

Tarefa periódica que lê a imagem gerada por uma câmara CCD, trata a imagem e armazena em uma estrutura de dados. Período de 500ms e tempo máximo de execução de 20ms. A imagem gerada é colocada em uma estrutura de dados do tipo *"buffer*

duplo", de forma a não criar situações de bloqueio para as tarefas que consomem esta informação.

- ATUALIZA_MAPA (A_M)

Tarefa que usa a imagem gerada pela tarefa LÊ_IMAGEM e procura identificar obstáculos. Em caso positivo, o mapa do ambiente mantido pelo veículo é atualizado. Período de 500ms e tempo máximo de execução de 100ms. Existe uma relação de precedência direta entre LÊ_IMAGEM e ATUALIZA_MAPA.

- RECONHECE_REFERÊNCIA (R_R)

Tarefa que usa a imagem mais recentemente gerada pela tarefa LÊ_IMAGEM e procura reconhecer referências. Em caso positivo, armazena a informação em estrutura de dados a ser consumida pela tarefa COMPUTA_POSIÇÃO, sem entretanto estabelecer uma relação de precedência. Período de 1300ms e tempo máximo de execução de 200ms.

- DEFINE_VELOCIDADE_DIREÇÃO (D_V_D)

Tarefa que usa a posição atual computada e a versão mais recente do mapa do ambiente para definir a velocidade e direção que o veículo deve assumir. Estas informações são passadas para o nível de pilotagem, responsável pela sua implementação. Esta tarefa possui período de 100ms e tempo máximo de execução de 30ms. Existe uma relação de precedência entre a tarefa COMPUTA_POSIÇÃO e a tarefa DEFINE_VELOCIDADE_DIREÇÃO.

- EXECUTA_DIAGNÓSTICO (E_D)

Tarefa aperiódica disparada por um sinal de rádio, ela executa um autodiagnóstico sobre o sistema, sendo capaz de parar o veículo de maneira controlada se uma situação de emergência for detectada. Possui um deadline de 20 ms e é suposto que o intervalo mínimo entre ativações é de 2 segundos. O tempo máximo de execução desta tarefa é 1 ms.

- RELATA (R)

Tarefa que informa a estação de supervisão sobre sua posição, velocidade e direção, a partir dados obtidos pelos sensores. Esta tarefa não é crítica (*"soft"*), pois não afeta o deslocamento e o comportamento do veículo.

Existe ainda uma relação de exclusão mútua entre as tarefas RECONHECE_REFERÊNCIA e COMPUTA_POSIÇÃO, no momento que esta última acessa as informações sobre referências identificadas. O tempo máximo de execução da seção crítica neste caso é de 1ms. Também existe uma relação de exclusão mútua entre as tarefas ATUALIZA_MAPA e DEFINE_VELOCIDADE_DIREÇÃO em função do mapa do mundo. O tempo máximo de bloqueio neste caso é de 3 ms.

Uma vez definidas as tarefas e suas características temporais, é necessário avaliar a

escalonabilidade do sistema, o que será feito na próxima seção.

5.1.3 Modelo de Tarefas

A tabela 5.1 resume a definição das tarefas, como descrito na seção anterior. Antes de proceder à análise de escalonabilidade, é necessário adaptar este conjunto de tarefas no sentido de refletir também os custos do sistema (*"overhead"*) e outros aspectos relativos à implementação.

Tarefa	Tipo	P (ms)	D (ms)	C (ms)	Predecessor	Bloqueio (ms)
C_P	Periódica	100	100	20		1 (c/R_R)
L_I	Periódica	500	500	20		
A_M	Periódica	500	500	100	L_I	3 (c/D_V_D)
R_R	Periódica	1300	1300	200		1 (c/C_P)
D_V_D	Periódica	100	100	30	C_P	3 (c/A_M)
E_D	Esporádica	2000	20	1		
R	Soft					

Tabela 5.1 - Definição das tarefas, versão inicial

O tempo de execução de alguns elementos do sistema operacional são computados juntos com o tempo de execução das tarefas. Por exemplo, o tempo de computação das chamadas de sistema e o tempo necessário para carregar o contexto da tarefa já estão considerados nos tempos de execução das tarefas.

O temporizador em hardware (*"timer"*) do sistema gera interrupções periodicamente. Estas interrupções são usadas para marcar a passagem do tempo e, entre outras coisas, liberar as tarefas sempre no início do respectivo período. Entretanto, a execução periódica do tratador de interrupções do *"timer"* representa uma interferência sobre as demais tarefas. Logo, o tratador de interrupções do *"timer"* entra no cálculo de escalonabilidade como uma tarefa periódica com período igual ao do *"timer"* e tempo de execução igual ao tempo de execução deste tratador de interrupções. São supostos os valores P_{timer}=10ms e C_{timer}=0.1ms para a tarefa *"timer"* que representa este tratador.

Também é necessário ampliar o modelo de tarefas inicial no sentido de incluir outros efeitos causados pelo suporte de execução. É suposto que a implementação emprega um sistema operacional de tempo real cujo *"kernel"* permanece quase a totalidade do tempo com as interrupções habilitadas. Entretanto, algumas seções críticas do *"kernel"* desabilitam as interrupções por breves instantes. A maior seção de código

do *"kernel"* que executa com interrupções desabilitadas o faz durante B_k = 0.1 ms.

Caso uma interrupção do *"timer"* aconteça enquanto o *"kernel"* está com interrupções desabilitadas, ela somente será tratada com um atraso (latência) de B_k. Suponha que esta interrupção do *"timer"* sinalize o início do período de uma tarefa. Neste caso, a liberação da tarefa somente vai acontecer B_k unidades de tempo após o início de seu período. Este atraso caracteriza claramente um *"release jitter"*. Todas as tarefas que não possuem predecessores sofrem um *"release jitter"* com duração máxima de B_k unidades de tempo, inclusive o próprio tratador do *"timer"*.

Outrossim, as tarefas que possuem predecessores não são liberadas por passagem de tempo mas sim pela sinalização da conclusão de sua tarefa predecessora. Desta forma, podemos considerar que a tarefa em questão possui um *"release jitter"* igual ao tempo máximo de resposta da sua tarefa predecessora. Observe que a tarefa predecessora é ativada pela passagem do tempo, e inclui no seu tempo máximo de resposta a latência de interrupção do sistema. A tarefa sucessora herda o *"release jitter"* da tarefa predecessora como parte do seu tempo máximo de resposta.

No caso do bloqueio por exclusão mútua, é importante observar que apenas ocorre bloqueio quando a tarefa de prioridade mais alta deve esperar a tarefa de prioridade mais baixa. Quando a tarefa de prioridade mais baixa tem que esperar, não ocorre bloqueio mas uma simples interferência. Assim, cada exclusão mútua gera apenas uma situação de bloqueio e não duas, como aparece na tabela 5.1. Por exemplo, a tabela 5.1 mostra bloqueio entre entre as tarefas R_R e C_P, e vice-versa. Na verdade somente existirá bloqueio da tarefa mais prioritária pela menos prioritária.

A tarefa esporádica E_D é assumida como periódica a nível de teste de escalonabilidade, como permitido pelas equações [9] e [10] do capítulo 2. Para executar a tarefa R podemos a princípio empregar qualquer tipo de servidor (PS ou DS, por exemplo). Mas a nível de teste ambos os servidores podem ser tratados como uma tarefa periódica. Se assumirmos que não serão pedidos dois relatórios em um intervalo de 10 segundos, então podemos tratar a tarefa R também como esporádica.

As tarefas E_D e R são ativadas por sinais de rádio e compartilham a mesma rotina que trata interrupções deste tipo. O tempo de execução deste tratador de interrupções é de B_m = 0.1 ms. Nesta aplicação o tratador de interrupções deste tipo tem o seu código acrescido ao da tarefa aperiódica esporádica E_D, pois é a tarefa mais prioritária que as demais da aplicação. Desta forma, ele aparece como interferência para as demais tarefas, com exceção da tarefa que representa o *"timer"*. Este tratador não interfere com o *"timer"* porque o vetor de interrupção é menos prioritário que o do *"timer"*. Entretanto, quando este tratador executa em função da tarefa R, ele aparece como uma situação de bloqueio para a tarefa E_D.

Será usada a política Deadline Monotônico (DM) neste sistema, pois algumas tarefas apresentam *"deadlines"* relativos menores que os seus respectivos períodos. Nas atividades a atribuição de prioridades obedece a orientação das relações de precedência,

isto é, tarefas predecessores possuem prioridade mais alta. A tabela 5.2 resume a definição das tarefas, após terem sido considerados todos os fatores discutidos acima. As tarefas aparecem na tabela 5.2 em ordem crescente de deadline, o que significa que as tarefas com prioridade mais alta aparecem antes na tabela.

Tarefa	Tipo	P (ms)	D (ms)	C (ms)	J (ms)	Predecessor	Bloqueio (ms)
timer	periódica	10	10	0,1	B_k		
C_P	periódica	100	100	20	B_k		1 (B_{R_R})
L_I	periódica	500	500	20	B_k		
A_M	periódica	500	500	100	R_{L_I}	L_I	
R_R	periódica	1300	1300	200	B_k		
D_V_D	periódica	100	100	30	R_{C_P}	C_P	3 (B_{A_M})
E_D	esporádica	2000	20	1	B_k		B_m
R	servidora	10000	80	5	B_k		

Tabela 5.2: Definição das Tarefas, Versão Final

5.1.4 Teste de Escalonabilidade

Para verificar a escalonabilidade do conjunto usaremos o teste baseado em tempos de resposta apresentado em [TBW94], onde tarefas experimentam também atrasos em suas liberações. Este teste é constituído pelas equações [9] e [10] do capítulo 2, as quais são reproduzidas abaixo, e pela verificação da condição:

$\forall i : 1 \leq i \leq n, R_i \leq D_i$.

$$W_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{W_i + J_j}{P_j} \right\rceil \cdot C_j \qquad [9]$$

$$R_i = W_i + J_i . \qquad [10]$$

Vamos a seguir calcular o tempo máximo de resposta de cada uma das tarefas da aplicação, como descritas no modelo de tarefas final. É importante destacar que o modelo de tarefas final inclui não somente as tarefas da aplicação, mas também tarefas que representam os custos associados com o suporte de execução.

Cálculo do tempo de resposta do *"timer"*

$$W_{timer} = C_{timer} = 0{,}1$$

$$R_{tiner} = W_{timer} + B_k = 0{,}2\,ms$$

Cálculo do tempo de resposta da tarefa E_D

$$W_{E\_D} = C_{E\_D} + B_M + \left\lceil \frac{W_{E\_D} + B_k}{P_{timer}} \right\rceil \times C_{timer} = 1{,}2$$

$$R_{E\_D} = W_{E\_D} + B_k = 1{,}3ms$$

Cálculo do tempo de resposta da tarefa R

$$W_R = C_R + \left\lceil \frac{W_R + B_k}{P_{timer}} \right\rceil \times C_{timer} + \left\lceil \frac{W_R + B_k}{P_{E\_D}} \right\rceil \times C_{E\_D} = 6{,}1$$

$$R_R = W_R + B_k = 6{,}2\,ms$$

Cálculo do tempo de resposta da tarefa C_P

$$W_{C\_P} = C_{C\_P} + B_{R\_R} + \left\lceil \frac{W_{C\_P} + B_k}{P_{timer}} \right\rceil \times C_{timer} + \left\lceil \frac{W_{C\_P} + B_k}{P_{E\_D}} \right\rceil \times C_{E\_D} +$$
$$+ \left\lceil \frac{W_{C\_P} + B_k}{P_R} \right\rceil \times C_R = 27{,}3$$

$$R_{C\_P} = W_{C\_P} + B_k = 27{,}4ms$$

Cálculo do tempo de resposta da tarefa D_V_D

$$W_{D\_V\_D} = C_{D\_V\_D} + B_{A\_M} + \left\lceil \frac{W_{D\_V\_D} + B_k}{P_{timer}} \right\rceil \times C_{timer} + \left\lceil \frac{W_{D\_V\_D} + B_k}{P_{E\_D}} \right\rceil \times C_{E\_D} + \\ + \left\lceil \frac{W_{D\_V\_D} + B_k}{P_R} \right\rceil \times C_R = 39{,}6$$

$$R_{D\_V\_D} = W_{D\_V\_D} + R_{C\_P} = 67\,ms$$

Cálculo do tempo de resposta da tarefa L_I

$$W_{L\_I} = C_{L\_I} + \left\lceil \frac{W_{L\_I} + B_k}{P_{timer}} \right\rceil \times C_{timer} + \left\lceil \frac{W_{L\_I} + B_k}{P_{E\_D}} \right\rceil \times C_{E\_D} + \left\lceil \frac{W_{L\_I} + B_k}{P_R} \right\rceil \times C_R + \\ + \left\lceil \frac{W_{L\_I} + B_k}{P_{C\_P}} \right\rceil \times C_{C\_P} + \left\lceil \frac{W_{L\_I} + R_{C\_P}}{P_{D\_V\_D}} \right\rceil \times C_{\mathrm{D\_V\_D}} = 123{,}3$$

$$R_{L\_I} = W_{L\_I} + B_k = 127{,}4\,ms$$

Cálculo do tempo de resposta da tarefa A_M

$$W_{A\_M} = C_{A\_M} + \left\lceil \frac{W_{A\_M} + B_k}{P_{timer}} \right\rceil \times C_{timer} + \left\lceil \frac{W_{A\_M} + B_k}{P_{E\_D}} \right\rceil \times C_{E\_D} + \left\lceil \frac{W_{A\_M} + B_k}{P_R} \right\rceil \times C_R + \\ + \left\lceil \frac{W_{A\_M} + B_k}{P_{C\_P}} \right\rceil \times C_{C\_P} + \left\lceil \frac{W_{A\_M} + R_{C\_P}}{P_{D\_V\_D}} \right\rceil \times C_{\mathrm{D\_V\_D}} = 258{,}6$$

$$R_{A\_M} = W_{A\_M} + R_{L\_I} = 386{,}0\,ms$$

Cálculo do tempo de resposta da tarefa R_R

$$W_{R\_R} = C_{R\_R} + \left\lceil \frac{W_{R\_R} + B_k}{P_{timer}} \right\rceil \times C_{timer} + \left\lceil \frac{W_{R\_R} + B_k}{P_{E\_D}} \right\rceil \times C_{E\_D} + \left\lceil \frac{W_{R\_R} + B_k}{P_R} \right\rceil \times C_R +$$
$$+ \left\lceil \frac{W_{R\_R} + B_k}{P_{C\_P}} \right\rceil \times C_{C\_P} + \left\lceil \frac{W_{R\_R} + R_{C\_P}}{P_{D\_V\_D}} \right\rceil \times C_{\mathrm{D\_V\_D}} + \left\lceil \frac{W_{R\_R} + B_k}{P_{L\_I}} \right\rceil \times C_{L\_I} +$$
$$+ \left\lceil \frac{W_{R\_R} + R_{L\_I}}{P_{A\_M}} \right\rceil \times C_{A\_M} = 1228{,}3$$

$$R_{R\_R} = W_{R\_R} + B_k = 1228{,}4\,ms$$

Como todas as tarefas apresentam tempo máximo de resposta menor do que o respectivo deadline (por exemplo, a tarefa R_R tem um tempo máximo de resposta de 1228.4 milisegundos, enquanto o seu deadline é de 1300ms), conclui-se que o sistema é escalonável.

Na próxima seção, serão discutidos aspectos da implementação desta aplicação e entre outras coisas será, a título de ilustração, definido como suporte deste projeto o RT-Linux. Como na abordagem assíncrona é necessário que se considere aspectos de implementação nas análises, todos os tempos referentes ao *"kernel"* devem sempre ser considerados tomando como base o suporte escolhido. Os tempos referentes aos tratadores e bloqueios do *"kernel"* (B_m, C_{timer} e B_k) foram super estimados quando assumidos nas nossas análises como sendo da ordem de 0.1ms. As medidas de latência de interrupção apresentadas em [YoB99] para a versão 2 do RT-Linux são da ordem de microsegundos. Este pessimismo exagerado dá uma boa margem de segurança que, uma vez o conjunto de tarefas passe pelos testes de escalonabilidade, estas tarefas quando implementadas terão suas restrições temporais respeitadas em tempo de execução.

5.1.5 Programação Usando RT-Linux

Uma aplicação de tempo real é tipicamente um programa concorrente. Logo, todas as técnicas de programação e métodos de depuração usados na programação concorrente podem ser usados aqui. Em particular, o ambiente de programação escolhido define como a solução será expressa em termos sintáticos. Ambiente de programação neste contexto significa a linguagem de programação e o sistema operacional escolhido. O tema "linguagens de programação" não será abordado neste livro, embora seja um tópico importante (um levantamento neste sentido pode ser

encontrado em [Coo96]). Apesar de diversas propostas a nível acadêmico, aplicações de tempo real são ainda programadas principalmente com C e C++. Linguagens de programação especialmente projetadas para tempo real procuram embutir na própria sintaxe da linguagem as construções relacionadas com tempo e concorrência, mas isto não acontece com C e C++. Estas são linguagens a princípio para programação seqüencial e qualquer capacidade além desta é obtida através de chamadas de rotinas de uma biblioteca ou do sistema operacional.

Com o propósito de ilustrar o tipo de serviço encontrado na prática, assim como a sintaxe típica, esta seção descreve parcialmente a API (*"Application Programming Interface"*) do Real-Time Linux versão 2, como descrita em [YoB99] e disponível, assim como o próprio sistema operacional, na página http://www.rtlinux.org. O RT-Linux está em rápida evolução e, provavelmente, quando este livro for publicado, novas versões já estarão disponíveis. A versão 2 do RT-Linux, implementada para processadores x86 genéricos, apresenta uma latência de interrupção máxima de 15 microsegundos. Uma tarefa periódica inicia sua execução com um desvio máximo de 25 microsegundos do instante planejado (se não existir outra tarefa liberada com prioridade maior). São valores que permitem o emprego do RT-Linux em uma ampla variedade de aplicações.

O conjunto de rotinas do RT-Linux é suficiente para implementar uma aplicação de tempo real composta por várias tarefas. Este sistema operacional foi projetado a partir da premissa que tarefas de tempo real executam sobre o *"microkernel"* com funcionalidade mínima e excelente comportamento temporal, ao passo que tarefas sem restrição temporal executam como tarefas Linux convencionais. Desta forma, qualquer acesso a disco, tela ou rede local deve ser feito por tarefas convencionais. A comunicação entre estes dois tipos de tarefas pode ser feita através do RT-FIFO ou através de memória compartilhada.

Abaixo está o exemplo de uma tarefa periódica simples, usando a API do RT-Linux versão 1. Neste exemplo foram utilizadas as seguintes chamadas de sistema:

int rt_get_time (void);

Retorna a hora em *"ticks"*.

int rt_task_init (RT_TASK *task, void (*fn)(int data), int data,
int stack_size, int priority);

Inicializa mas não escalona uma tarefa.

int rt_task_make_periodic (RT_TASK *task, RTIME start_time,
RTIME period);

Transforma a *"thread"* em periódica.

int rt_task_wait (void);

Libera o processador até a próxima ativação da tarefa.

```
#define     STACK_SIZE  3000
RT_TASK     my_task;

/* Código da tarefa de tempo real, um laço infinito */
void codigo_da_tarefa( unsigned int x) {
      /* Variáveis locais desta tarefa */
      . . .
      while ( 1 ) {
            /* Faz o que esta tarefa tem que fazer */
            . . .
            /* Espera até o próximo período */
            rt_task_wait();
            }
}

/* Módulo de inicialização da aplicação */
int init_module( void ){
      RTIME now = rt_get_time();

      /* Inicializa a tarefa */
      rtl_task_init( &my_task, codigo_da_tarefa, arg, STACK_SIZE, 1 );

      /* Executa a cada 50 ms, aproximadamente 450 na unidade usada */
      rtl_task_make_periodic(&mytask, now, 450);
      return 0;
}
```

Um exemplo completo usando RT-Linux pode ser encontrado em [Wur99], onde uma aplicação de controle é descrita. Uma interface gráfica de usuário usando X-Windows comunica-se com tarefas de tempo real através de RT-FIFO, as quais controlam uma variável física através de um algoritmo de controle simples.

5.2 Aplicação com Abordagem Síncrona

O objetivo deste exemplo é de ilustrar uma metodologia e um estilo de programação para a abordagem síncrona. A aplicação escolhida é simples o suficiente para o leitor não precisar de conhecimentos adicionais e poder focalizar apenas os aspectos associados a sua programação. A aplicação consiste de um sistema de controle de processo, inspirado no exemplo apresentado em [AnP93] e cuja descrição segue. O sistema a controlar consiste de um reservatório de água do qual pretende-se controlar o nível para que ele fique sempre entre um mínimo e um máximo. Uma bomba quando ligada tende a esvaziar o reservatório e uma válvula permite enche-lo quando aberta. A informação de nível é fornecida por dois sensores indicando os limites inferior e superior para a água contida no reservatório (respectivamente nível mínimo e nível máximo). A figura 5.2 mostra um diagrama esquemático da aplicação de regulação de nível de um reservatório.

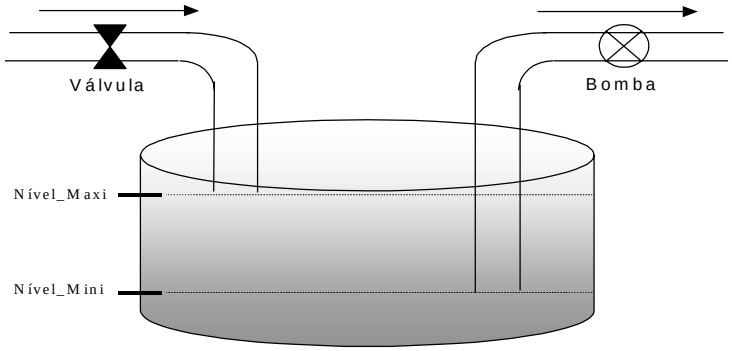


Figura 5.2: Esquema da Regulação de Nível de um Reservatório

O sistema de controle será estudado e construído num primeiro tempo para o comportamento normal do sistema a controlar e a seguir para uma situação de falha neste. Inicialmente, é tratado o caso do funcionamento normal do sistema a controlar. O problema é decomposto em duas partes, uma correspondente a atividade da válvula para regular o nível do reservatório e outra a atividade de consumo de água pela bomba. Essas duas atividades são independentes e ocorrem em paralelo. A partir desta visão do problema, é possível construir o programa do sistema de controle do reservatório como sendo o resultado da composição de dois módulos atuando em paralelo, um responsável pela regulação de nível a partir da ação da válvula (**REGNIVEL**) e outro pelo consumo de água pela bomba (**CONSUMO**). A arquitetura do sistema de controle é apresentada na figura 5.3.

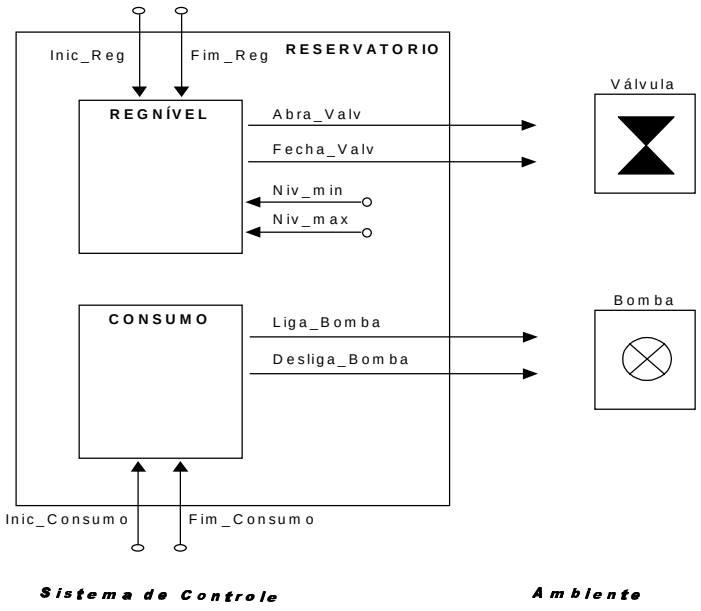


Figura 5.3: Arquitetura do Sistema de Controle

O módulo **REGNIVEL** visa manter o nível do reservatório entre o mínimo e o máximo indicado pelos sensores de nível (sinais **Niv_min** e **Niv_max),** abrindo e fechando a válvula de regulação do reservatório a partir da emissão dos sinais **Abra_Valv** e **Fecha_Valv**. O mecanismo de regulação é ativado por um comando externo de inicio (sinal **Inic_Reg)** e atua até ocorrer o comando de fim (sinal **Fim_Reg)**. Para efeito de simplificação do problema, é assumido que no inicio, o reservatório se encontra com um nível intermediário entre o mínimo e o máximo. O código em Esterel do módulo é apresentado a seguir:

```
module REGNIVEL :
input Inic_Reg, Fim_Reg, Niv_min, Niv_max;
output Abra_Valv, Fecha_Valv;

await Inic_Reg;
emit Fecha_Valv;
abort
        loop
                await Niv_min;
                emit Abra_Valv;
                await Niv_max;
                emit Fecha_Valv;
        endloop
when Fim_Reg
endmodule
```

Deve se notar que a programação do módulo **REGNIVEL** segue o estilo de programação Esterel descrito no capítulo 4. Após a inicialização do processo de regulação a partir do sinal **Inic_Reg**, o corpo do programa descrevendo a regulação de nível pela válvula está embutido numa construção do tipo preempção forte que termina com a chegada do sinal **Fim_Reg**.

O módulo **CONSUMO** gera os comandos de liga e desliga da bomba de água (sinais **Liga_Bomba, Desliga_Bomba**), a partir de comandos externos de inicio e fim de consumo (sinais **Inic_Consumo**, **Fim_Consumo**). O código deste módulo em Esterel é:

```
module CONSUMO :
input Inic_Consumo, Fim_Consumo;
output Liga_Bomba, Desliga_Bomba;

loop
        await Inic_Consumo;
        emit Liga_Bomba;
        await Fim_Consumo;
        emit Desliga_Bomba
endloop
endmodule
```

O programa em Esterel do sistema de controle do Reservatório em situação de funcionamento normal do sistema a controlar é o módulo **RESERVATORIO**, resultado da composição paralela dos dois módulos anteriores:

```
module RESERVATORIO :
input Inic_Reg, Fim_Reg, Inic_Consumo, Fim_Consumo;
output Abra_Valv, Fecha_Valv, Liga_Bomba, Desliga_Bomba;

signal Niv_min, Niv_max in
        run CONSUMO
||
        run REGNIVEL
end signal
end module
```

No caso de considerar também as situações anormais do sistema a controlar, o sistema necessita além do módulo de controle, um módulo de detecção de falha e um de tratamento desta. Uma situação anormal clássica neste exemplo do reservatório corresponde ao funcionamento da bomba "a vazio"; é necessário detectar então quando o nível do reservatório é muito baixo para poder parar o funcionamento da bomba, evitando danifica-la. Para descrever o comportamento do sistema, torna-se necessário introduzir um módulo encarregado de detectar falha (no caso o nível de água insuficiente) e um módulo de recuperação da falha. Esses módulos são respectivamente os módulos **MONITORA_CONSUMO** e **REPARO**.

Para ter a garantia que existe realmente uma falha no sistema, o mecanismo de detecção do módulo **MONITORA_CONSUMO** atuará somente se o sinal **Niv_min** estiver presente em três ocorrências seguidas de um sinal de relógio **Rel**, confirmando desta forma a existência da falha. A ocorrência desta falha gera uma exceção que interrompe imediatamente o bloco do mecanismo de detecção e passa o controle para o bloco de tratamento de exceção. A construção Esterel "**trap T ... exit T ... handle T ...**" é perfeitamente adequada para implementar o mecanismo de levantamento de exceção ("**trap T ... exit T**") e o tratamento desta ("**handle T**"); o módulo de detecção de falha **MONITORA_CONSUMO** esta construído a partir dela. O corpo da construção **trap** contém o mecanismo de detecção de falha adotado que quando acionado, desvia para o tratador de exceção desta **handle**. O módulo **MONITORA_CONSUMO** tem o código mostrado na próxima página.

O tratamento da exceção levantada no módulo **MONITORA_CONSUMO** é realizado através do módulo **REPARO** que se encarrega do reparo da situação anormal correspondente ao nível insuficiente do reservatório. A especificação do sistema define que este reparo deve ser realizado num tempo inferior a 10 minutos, senão um alarme para um nível mais alto de supervisão deverá ser enviado para assinalar a impossibilidade de reparo.

```
module MONITORA_CONSUMO :
input Rel, Niv_min, Consumo_em_Curso

var contador_tempo: integer in

contador_tempo := 0;
loop
        trap MONITOR in
                loop
                        present Niv_min then
                                contador_tempo := contador_tempo + 1;
                                present Consumo_em_Curso then
                                        if contador_tempo >= 3 then
                                                exit MONITOR
                                        end if
                                end present
                        else
                                contador_tempo := 0
                        end present
                each Rel
        handle MONITOR do
        % tratamento da situação detectada de insuficiência do nível de água
        % pela execução do módulo REPARO
        end trap
end loop
end var
end module
```

O reparo é realizado por uma tarefa assíncrona **Reparo_Nivel_Insuficiente ( ) ( )** do módulo **REPARO** que iniciará logo após a exceção **MONITOR** do módulo **MONITORA_CONSUMO** ter sido levantada. A execução desta tarefa assíncrona externa (não descrita aqui) é realizada utilizando a primitiva **exec** de Esterel. No mesmo tempo que inicia esta tarefa**,** o módulo **REPARO** envia o sinal **Bomba_em_Reparo** a todos os outros módulos, segundo o mecanismo de difusão instantânea de Esterel, para impedi-los de atuar enquanto a situação de reparo estiver mantida. A finalização da tarefa assíncrona de reparo **Reparo_Nivel_Insuficiente ( ) ( )** pode ser feita de forma normal se a duração de reparo for inferior a 10 minutos, emitindo um sinal de **Reparo_Bomba_Concluido** que permitirá abortar a execução concorrente, autorizando novamente a atuação dos outros módulos. Se o tempo de reparo ultrapassar o tempo fixado de 10 minutos, um *"time-out"* é detectado, ativando a emissão de um sinal de alarme **Alarme_Bomba** no nível superior de supervisão e aguardando que este lhe permita retomar a execução no futuro através de um sinal **Bomba-Autorizada** vindo do nível de supervisão. O código do módulo **REPARO** é apresentado a seguir:

```
module REPARO :
input MINUTE, Bomba-Autorizada;
output Bomba_em_Reparo, Alarme_Bomba, Reparo_Bomba_Concluido;
task Reparo_Nivel_Insuficiente ( ) ( );

        [
                abort
                        exec Reparo_Nivel_Insuficiente ( ) ( )
                when 10 MINUTE do
                        emit Alarme_Bomba;
                        await Bomba-Autorizada
                end abort;
                emit Reparo_Bomba_Concluido
        ]
||
        abort
                sustain Bomba_em_Reparo
        when Reparo_Bomba_Concluido
endmodule
```

Deve se notar que o módulo **REPARO** é composto de dois blocos em paralelo, um deles responsável pela execução do reparo da bomba e o outro pela informação aos outros módulos (via difusão instantânea) da situação de reparo desta, enquanto durar. A informação de conclusão do reparo no primeiro bloco (sinal **Reparo_Bomba_Concluido**) é imediatamente difundida ao segundo que deixa de emitir para os outros módulos o sinal **Bomba_em_Reparo**.

Para levar em conta, no programa em Esterel do sistema de controle do Reservatório, o comportamento faltoso e sua recuperação descritos anteriormente é necessário substituir no módulo **RESERVATORIO** o módulo **CONSUMO** por um módulo **CONSUMO_SEGURO**. Neste módulo, a indicação de ativação e desativação do sistema é feita respectivamente a partir dos sinais **Inic** e **Fim**. No módulo **CONSUMO_SEGURO**, após a ativação do sistema (sinal **Inic**), é emitido o sinal de inicio da regulação (sinal **Inic_Reg)** que será interrompida quando o sinal de desativação do sistema ocorrer (sinal **Fim**). Entre essas duas ocorrências (ativação e desativação), o módulo **MONITORA_CONSUMO** atuará em paralelo com o bloco que descreve o consumo de água e emite comandos para ligar e desligar a bomba. O código do módulo **CONSUMO_SEGURO** é apresentado a seguir:

```
module CONSUMO_SEGURO :
input Inic, Fim, Inic_Consumo, Fim_Consumo, Rel, Niv_min, MINUTE, Bomba-Autorizada;
output Liga_Bomba, Desliga_Bomba, Alarme_Bomba, Inic_Reg, Fim_Reg;
task Reparo_Nivel_Insuficiente ( ) ( );

signal Bomba_em_Reparo, Consumo_em_Curso, Reparo_Bomba_Concluido in
        await Inic;
        emit Inic_Reg;
        abort
                loop
                        await Inic_Consumo;
                        emit Liga_Bomba;
                        abort
                                sustain Consumo_em_Curso
                        when Fim_Consumo;
                        emit Desliga_Bomba
                end loop
        ||
                run MONITORA_CONSUMO
        when Fim
end signal
end module
```

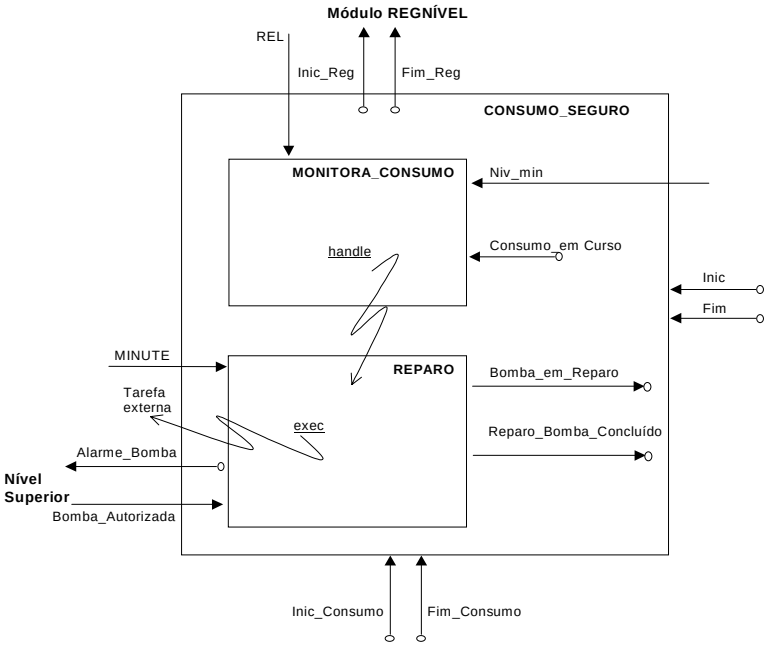


Figura 5.4: Representação do Módulo **CONSUMO_SEGURO**

A figura 5.4 apresenta o módulo **CONSUMO_SEGURO** com seus módulos internos **MONITORA_CONSUMO** e **REPARO** e suas interfaces com o módulo **REGNIVEL**, com o nível superior de Supervisão e com o ambiente (ou sistema a controlar).

No exemplo anterior, foi apresentado um sistema simples no qual a modelagem do comportamento não apresentava muitas alternativas. Entretanto, dependendo da complexidade do problema, podem existir várias possibilidades de modelar o comportamento deste. A seguir, são apresentadas e discutidas algumas abordagens possíveis para a modelagem de sistemas mais complexos como por exemplo sistemas de comando de uma célula de manufatura composta de vários equipamentos (robôs, maquinas, esteiras, sensores, etc.). Exemplos de tais aplicações podem ser encontrados em [Cos 89], [Bud94], [CER92] e servem de base para a apresentação das diversas abordagens de modelagem:

Abordagem 1: Decomposição por componente físico
Esta abordagem consiste em associar um módulo Esterel a cada componente físico do problema (p.ex. um módulo para cada robô) e em aproveitar o conceito de paralelismo da linguagem Esterel. Cada módulo é constituído por um conjunto de atividades geralmente realizadas em seqüência (p.ex. o comportamento de um robô pode ser visto como a realização de uma atividade somente após a conclusão de outra). Todos estes módulos são colocados em paralelo e a troca de sinais entre eles permitem que eles cooperam entre si.

Abordagem 2: Decomposição funcional
A abordagem anterior esta baseada numa visão totalmente paralela da aplicação, com alguns pontos de sincronização. Entretanto, esta visão do problema não leva em conta na modelagem a existência de tarefas da aplicação que se executam seqüencialmente. A segunda abordagem consiste em programar a aplicação a partir de uma decomposição funcional desta. A idéia básica desta abordagem consiste em considerar que algumas tarefas só podem ser realizadas após o término de outras; a modelagem da aplicação e sua conseqüente programação leva em conta este fato e só permite o inicio da execução das varias tarefas quando necessário, evitando a espera de eventos que não tem nenhuma probabilidade de ocorrer.

Abordagem 3: Decomposição por recursos compartilhados
Esta abordagem segue a decomposição funcional como na abordagem anterior, entretanto ela leva em conta a existência de recursos comuns e decompõe a aplicação em função destes. Cada uma das funcionalidades que utilizam um mesmo recurso corresponde a um módulo; todos esses módulos são colocados em paralelo.

Essas três abordagens foram comparadas nos trabalhos citados anteriormente do ponto de vista da complexidade do autômato resultante (i.e. número de estados e de

transições). Os resultados obtidos mostram diferenças grandes (da ordem de 10 a 20 vezes) entre o número de estados e de transições dos autômatos resultantes da abordagem 1 e das abordagens 2 e 3, apesar de descrever as mesmas funcionalidades. A arquitetura adotada na abordagem 1 é a causa deste aumento no número de estados, pois por causa do paralelismo, os módulos esperam um grande número de sinais (eventos) de entrada; como parte deles são impossíveis de serem recebidos antes de outros terem sido emitidos, a abordagem 1 cria um grande número de estados nunca atingidos e então desnecessários. Por outro lado, as abordagens 2 e 3 implicam em uma implementação centralizada enquanto a abordagem 1 permite também adotar uma implementação distribuída, com programas separados para cada módulo-equipamento. Estas considerações devem ser levadas em conta quando da programação de sistemas complexos e fazem parte da metodologia a ser adotada para programar aplicações segundo a abordagem síncrona.

5.3 Considerações sobre as abordagens usadas

Os exemplos apresentados neste capítulo mostram que as abordagens assíncrona e síncrona apresentam características muito diferentes e consequentemente existem campos de aplicação para as quais cada uma delas é mais apropriada. Esta seção procura destacar estas diferenças, assim como indicar critérios utilizados para selecionar uma ou outra abordagem frente a uma dada aplicação. Essas abordagens, as quais visam fornecer soluções para o problema de tempo real, devem ser analisadas tanto do ponto de vista da especificação e verificação quanto da implementação.

A abordagem assíncrona é considerada como orientada à implementação e consequentemente descreve o sistema da forma a mais completa possível. Essa descrição leva em conta a ocorrência e a percepção de todos os eventos numa ordem arbitrária mas não simultânea; o comportamento observado corresponde a todas as combinações de ocorrência de eventos (*"interleaving")*. A abordagem síncrona é considerada como orientada ao comportamento da aplicação e a sua verificação. A premissa básica da abordagem síncrona é a simultaneidade dos eventos de entrada e saída, partindo da hipótese que no nível de abstração considerado, cálculos e comunicações não levam tempo. A abordagem síncrona se situa num nível de abstração dos aspectos de implementação tal que essa hipótese seja verdadeira.

Como conseqüência destes princípios e modelos diferentes nas duas abordagens, leva-se em conta na especificação e no projeto segundo a abordagem assíncrona, características do suporte de software e de hardware das aplicações, o que pode dificultar requisitos de portabilidade. A abordagem síncrona é mais distante das questões de implementação, o que lhe dá uma vantagem do ponto de vista da portabilidade.

A descrição completa do comportamento e a introdução de considerações de implementação na abordagem assíncrona torna complexa a análise das propriedades do

sistema por causa do grande número de estados gerados e do possível não determinismo introduzido. A abordagem síncrona, se situando num nível de abstração maior, facilita a especificação e a verificação das propriedades de sistemas de tempo real. Como resultado, os sistemas construídos a partir da abordagem síncrona são muito mais confiáveis do que aqueles construídos segundo a abordagem assíncrona. A abordagem síncrona, entretanto por se situar num nível de abstração maior do que a abordagem assíncrona pode encontrar dificuldades em algumas situações do ponto de vista da implementação da qual está mais afastada, em particular em implementações que necessitam ser distribuídas.

Na abordagem assíncrona, a concorrência e o tempo estão tratados de forma explícita, tornando fundamental o estudo da previsibilidade dos sistemas de tempo real e consequentemente a questão do escalonamento tempo real. As premissas da abordagem síncrona permitem resolver a concorrência sem o entrelaçamento de tarefas, não existindo a necessidade de escalonamento. Esta surge quando diferentes atividades requerem o uso do mesmo recurso (processador, qualquer periférico ou até mesmo as estruturas de dados), o qual não pode ser usado simultaneamente pelas várias atividades. Na abordagem síncrona, como todas as atividades são instantâneas, o recurso não é ocupado, e deixa de ser um problema. Além do mais, na abordagem síncrona, o tempo não aparece de maneira explícita.

Na abordagem síncrona, a única preocupação do projetista da aplicação é garantir a resposta correta aos eventos do ambiente, no sentido lógico. A correção no sentido temporal está automaticamente garantida pois, como a velocidade de processamento é considerada infinita, o tempo de resposta será sempre zero. A abordagem assíncrona parte da programação concorrente clássica e procura tratar da questão tempo, além das questões de sincronização entre tarefas. Para isto as políticas de escalonamento são tornadas explícitas (algo que não acontece na programação concorrente clássica) e os mecanismos de sincronização (semáforos por exemplo) são dotados de um comportamento mais apropriado para a análise de escalonabilidade. Interferências, precedências e bloqueios são considerados no momento de determinar se os *"deadlines"* da aplicação serão ou não cumpridos. Um sistema construído segundo a abordagem assíncrona não precisa ser mais rápido que o ambiente a sua volta. Ele apenas precisa, usando os recursos de forma inteligente, atender aos requisitos temporais (*"deadlines"*) expressos na especificação do sistema. Na abordagem síncrona esta análise de escalonabilidade usada na abordagem assíncrona torna-se trivial e desnecessária, pois os *"deadlines"* são muitas ordens de grandeza maiores do que o tempo que o processador leva para executar as tarefas da aplicação.

O campo de aplicação da abordagem síncrona se limita a situações de sistemas ou partes de sistemas nas quais a velocidade do ambiente pode ser considerada como menor que a do computador, atendendo as exigências da hipótese de sincronismo. Nestas situações, a abordagem síncrona se impõe pela sua simplicidade, facilidade de especificação e potencialidade em verificar o sistema especificado. Esta abordagem não necessita de um processo de tradução para passar da especificação validada ao programa a ser executado; ela implementa diretamente a especificação, o que se

configura numa grande vantagem por evitar a introdução de erros durante o processo de tradução. Uma aplicação construída com a abordagem assíncrona é mais complexa e mais difícil de verificar e consequentemente mais sujeita a falhas do que a mesma aplicação na abordagem síncrona. Entretanto, quando o computador não consegue ser muito mais rápido que o ambiente ou quando não é possível determinar com garantia a relação de tempo entre o ambiente e o computador (p.ex. é o caso da Internet), não existe escolha e a abordagem assíncrona deve ser utilizada. Geralmente nos sistemas complexos, as duas abordagens podem se tornar necessárias; partes mais reativas do sistema tais como interfaces homem-máquina, controladores, protocolos de comunicação e "drivers" de periféricos podem ser tratadas pela abordagem síncrona, enquanto as partes correspondendo aos cálculos, ao manuseio e tratamento de dados e à comportamentos não-deterministas tem que fazer uso da abordagem assíncrona.

5.4 Conclusão

Este capítulo apresentou duas aplicações de tempo real. A aplicação "veículo com navegação autônoma" foi implementada através da abordagem assíncrona, com análise de escalonabilidade e emprego de um sistema operacional de tempo real. A aplicação "sistema de controle" foi implementada através da abordagem síncrona, com o emprego da linguagem Esterel. Finalmente, a seção 5.3 procurou fazer uma comparação entre as duas abordagens.

Embora a limitação de espaço permita apenas uma descrição superficial destas aplicações, elas permitem que o leitor tenha contato com o método implícito na adoção de cada uma das abordagens. A partir destes exemplos, fica mais fácil identificar situações onde uma e outra podem ser melhor aplicadas.

Capítulo 6

Tendências Atuais em Sistemas de Tempo Real

Sistemas de Tempo Real são reconhecidos por possuírem problemas bem definidos e únicos. Um conjunto de técnicas, métodos, ferramentas e fundamentação teórica formam uma disciplina necessária na compreensão e na concepção de sistemas de tempo real. Apesar da evolução nos últimos anos, em termos de conceitos e métodos, os meios mais convencionais continuam a ser usados na prática. Além do aspecto dessas ferramentas usuais não tratarem com a correção temporal, um outro fato que se pode considerar é o relativo desconhecimento do que seria tempo real. A grande maioria dos sistemas de tempo real é projetada a partir de um entendimento errado que reduz tempo real, a simplesmente uma questão de melhoria no desempenho.

A literatura de tempo real tem sido pródiga no definição de modelos, metodologias e suportes que consideram restrições temporais. Foi no intuito de melhorar o conhecimento desta disciplina que nos propomos neste texto a apresentar duas abordagens para a construção de STRs: a síncrona e a assíncrona.

6.1 Abordagem Síncrona

A abordagem síncrona é baseada na hipótese síncrona que, em seus princípios básicos, considera os processamentos e comunicações instantâneos. O tempo é visto com granularidade suficientemente grossa para que essas premissas sejam aceitas como verdadeiras. A observação dos eventos nessa abordagem é cronológica.

A abordagem síncrona é considerada como orientada ao comportamento de uma aplicação e à sua verificação. Por se situar num nível de abstração maior, os aspectos de implementação são em grande parte desconsiderados, facilitando a especificação e a análise das propriedades de sistemas de tempo real. A abordagem síncrona forma uma base conceitual bastante sólida permitindo a definição de conjuntos de ferramentas integradas próprias para o desenvolvimento de aplicações de tempo real. Entretanto, aplicações complexas que envolvem programas de calculo, grande quantidade de dados a manusear e distribuição são dificilmente tratáveis por completo pela abordagem síncrona. A abordagem síncrona é bem adaptada para sistemas reativos ou partes reativas de sistemas complexos que são dirigidos pelo ambiente e para os quais o determinismo no comportamento é fortemente desejável.

Os sistemas reativos para os quais o uso da abordagem síncrona é uma solução recomendada se encontram nos seguintes campos de aplicação: controle de processo tempo real, sistemas de manufatura, sistemas de transporte, sistemas embutidos, sistemas autônomos, sistemas de supervisão, protocolos de comunicação, "*drivers*" de periféricos, interface homem-máquina, processamento de sinais, entre outros.

Geralmente, as linguagens síncronas não são suficientes para a programação completa dos sistemas complexos. As partes não reativas têm que ser construídas usando a abordagem assíncrona. As linguagens assíncronas devem fornecer mecanismos para a interação entre as partes reativas e não reativas segundo um modo de comunicação assíncrona. Propostas de formalismos ou de metodologias que fazem uso de modos de comunicação síncronos e assíncronos em conjunto podem ser encontradas em [SBS93] e [BCJ97].

A distribuição do código em vários computadores é uma característica indispensável dos sistemas complexos por uma ou mais das razões seguintes: aumento da capacidade de cálculo, melhoria de desempenho, tolerância a falhas, localização geográfica dos sensores e atuadores, entre outras. As dificuldades encontradas para tratar a distribuição na abordagem síncrona provem da necessidade, imposta pela hipótese de sincronismo, da existência de um relógio global e, conseqüentemente, de um algoritmo (síncrono) de sincronização de relógios. A distribuição de código pode ser feita de três formas diferentes [CGP99]:

(i) Compilando separadamente cada parte do programa total, e alocando aos computadores do sistema cada uma destas partes que devem se comunicar. Apesar de ser uma solução fácil, nem sempre a compilação em separado gera programas seqüenciais deterministas [Maf93];

(ii) Compilando o programa todo e gerando programas seqüenciais para cada computador, cada programa podendo se comunicar com os outros, segundo o *método de grafo abstrato* apresentado em [Maf93];

(iii) Compilando o programa num único programa objeto e distribuindo-o posteriormente em tantos programas quanto computadores que realizarão apenas os cálculos que lhes correspondem [CaK88] e [CPG99]. Esta abordagem tem a vantagem de otimizar a compilação e de permitir a verificação e depuração do programa antes de distribui-lo.

Maiores detalhes sobre o estado atual das pesquisas sobre distribuição de programas síncronos podem ser encontradas em [CGP99], [BCL99] e [BeC99].

6.2 Abordagem Assíncrona

A abordagem assíncrona visa uma descrição a mais exata possível de um sistema de

tempo real e por isto é considerada como orientada à implementação. Vários aspectos referentes a uma aplicação são tratadas explicitamente a nível de programação e gerenciadas em tempo de execução; o tempo e a concorrência são exemplos deste conhecimento explicito necessário a um programador de aplicações de tempo real nesta abordagem. Como conseqüência, torna-se necessário levar em conta já na especificação e no decorrer do projeto, algumas características dos suportes de software e de hardware. Por outro lado, na abordagem assíncrona, a procura de uma descrição completa do comportamento e a introdução de considerações de implementação torna complexa a análise das propriedades do sistema de tempo real.

Num sentido mais clássico, dentro desta ótica da abordagem assíncrona, uma fundamentação teórica já é bem definida para sistemas onde é possível uma previsibilidade determinista. Uma coleção de algoritmos e técnicas de analise existem para problemas envolvendo escalonamentos com garantias em tempo de projeto, caracterizando o que Stankovic chama de *ciência da garantia de desempenho* [Sta96]. Os sistemas, neste perspectiva de garantia em tempo de projeto, são de dimensões não muito extensas, construídos para realizar um conjunto específico de objetivos e envolvendo, em termos de implementação, soluções ditas proprietárias. Estão dentro desta ciência da garantia do desempenho as abordagens que tratam com ambientes dinâmicos, ainda limitados, usando técnicas para obter garantias dinâmicas. Estas abordagens envolvem testes de aceitação baseados em hipóteses de pior caso dos tempos de computação da carga presente no sistema. O estudo apresentado neste texto de certa maneira cobriu estas técnicas que formam esta ciência da garantia.

Como os sistemas se tornam cada vez maiores e mais complexos, interagindo com ambientes também complexos e dinâmicos – e, portanto, não deterministas – uma expansão desta *ciência da garantia* é necessária para captar as características destes novos sistemas. As técnicas existentes, mesmo as de abordagens assíncronas que tratam com garantias dinâmicas, não são abrangentes o suficiente para que possam ser eficazes diante destes novos sistemas.

A evolução prevista para os próximos anos caracteriza sistemas como sendo distribuídos, de larga escala, oferecendo uma grande variedade de serviços que trata com várias formas de dados e se apresentam constantemente mudando e evoluindo. Em contraste, um desafio crescente colocado a comunidade de tempo real é como construir esses sistemas de propósito geral, abertos, que permitam conviver várias aplicações independentes e de tempo real.

Uma perfeita análise de escalonabilidade a priori é impraticável em um ambiente destes. É necessário uma nova disciplina, com modelos e abordagens criados no sentido de captar as complexidades desses novos ambientes que estão se caracterizando envolvendo novas tecnologias e aplicações. Esta nova disciplina para aplicações de tempo real vai exigir esforços de pesquisa na engenharia de software, em linguagens de programação, em sistemas operacionais, na teoria de escalonamento e na comunicação.

Sistemas Abertos

A meta principal nesta tendência é o desenvolvimento de arquiteturas, suportes de "*middlewares*" e componentes com interfaces públicas e bem definidas, que possam ser desenvolvidos de forma independente, para posteriormente serem combinados e usados na construção de sistemas complexos. Esta estratégia conduz à obtenção, além de grande flexibilidade, de vantagens como melhor portabilidade, interoperabilidade e facilidades na evolução dos sistemas. Estas interfaces favorecem também o uso de componentes/softwares de prateleira ("*off-the-shelf*").

As tecnologias abertas para sistemas distribuídos de tempo real são recentes. O RT CORBA [OMG98] e o ANSAware/RT [Li95] são exemplos destes esforços de organizações padronizadoras na definição de padrões para componentes de "*middleware*" que suportem aplicações distribuídas de tempo real. E, mesmo para a automação industrial e sistemas embutidos em geral, estão previstos certos requisitos para as próximas gerações que envolvem padrões abertos [Sta96]: os controladores (computadores especializados) devem apresentar arquitetura modular que suporte mudanças em tempos mínimos sem comprometer os requisitos de desempenho de uma planta industrial; devem suportar facilmente a adição de funcionalidade ou a alteração no comportamento da aplicação e, também, não depender de fabricante ou tecnologias proprietárias.

A adequação de padrões abertos para o desenvolvimento de aplicações de tempo real sempre foi vista com uma certa desconfiança por projetistas da área. A necessidade de previsibilidade no atendimento das restrições temporais, historicamente tem contribuído para soluções proprietárias, específicas para suas aplicações alvo. Algumas das dificuldades de uma arquitetura aberta para aplicações de tempo real em sistemas distribuídos de larga escala incluem:

- Características de hardware desconhecidas até o tempo de execução (velocidades de processadores, "*caches*", barramentos , memória variam de máquina para máquina).

- Ambientes onde convivem diferentes aplicações onde suas necessidades de recursos e seus requisitos temporais são desconhecidos até o tempo de execução.

Linguagens

Até recentemente, muito pouco tinha sido feito em termos de linguagens de programação para sistemas de tempo real. Na prática corrente, no uso de abordagens assíncronas em muitas aplicações, as partes críticas das aplicações eram (ou ainda são) implementadas em códigos de máquina ou *"Assembly"*. Isto forçava um conhecimento

prévio sobre a arquitetura dos processadores usados no sistema. As implicações são muitas no uso desta prática: os códigos produzidos não são portáveis, e a própria modificação ou evolução do sistema fica comprometida. Todos esses fatores apontam para a necessidade do uso de linguagens de programação de alto nível com construções explícitas que facilitem a descrição do comportamento temporal de uma aplicação de tempo real.

Em quase todos os modelos de escalonamento citados no capítulo 2, é assumido o pior caso nos tempos de computação dos códigos das tarefas. Os códigos gerados na compilação destas linguagens de alto nível devem ter, portanto, bem determinados os valores de seus piores tempos de computação. Para a obtenção destes tempos é necessário evitar a utilização de certas construções de linguagens convencionais e certas práticas correntes de programação. O uso de estruturas dinâmicas deve também ser limitado.

São necessárias ferramentas para preverem o pior caso e o tempo de computação médio destes códigos. A estimação do pior caso pode ser obtida por análises ou por medidas. Algumas ferramentas existentes, baseada em análises do código, são protótipos acadêmicos [Gus94], [Har94]. A dificuldade de métodos baseados em análise é a necessidade de um modelo que capte todas as características da arquitetura do computador ("*cache*", "*pipeline*", etc). O problema de técnicas baseadas em medidas está na dificuldade de se conseguir o pior caso de tempo de computação a partir de um certo número de ativações sob o qual é submetido o código de uma tarefa.

As linguagens de programação em geral são fracas no sentido de suportarem a implementação das abordagens de escalonamento de tempo real. A falta de expressividade fica evidente quando o objetivo é a generalização de requisitos temporais e de estratégias de implementação. É improvável que uma única linguagem seja abrangente o suficiente para acomodar todos os requisitos temporais das abordagens existentes e mais os novos que certamente deverão surgir nos próximos anos. São necessárias novas estruturas de linguagens mais flexíveis que permitam a especialização diante dos diferentes tipos de aplicações de tempo real. A Reflexão Computacional, através dos protocolos Meta-Objeto ([Mae87], [TaT92]), e a Programação Orientada a Aspectos [Kic97] são exemplos de novas técnicas e mecanismos de programação que são incentivadas porque trazem esta flexibilidade necessária na programação em tempo real. Ambas técnicas enfatizam a separação das funcionalidades de uma aplicação de seu gerenciamento.

Recentemente, o grande sucesso da linguagem Java no contexto da Internet e mesmo fora dela, atraiu a atenção das pessoas que desenvolvem software de tempo real. Na sua forma original Java não é apropriada para tempo real devido às diversas fontes de não determinismo presentes. O exemplo clássico é o seu coletor de lixo, o qual pode executar a qualquer momento e ocupar o processador por um tempo não desprezível, sempre que o suporte ficar sem memória para instanciar novos objetos. Existem dois fortes consórcios definindo uma versão de Java para tempo real (*Real-Time Java*, [Ort99]). Embora neste momento não esteja claro qual será o consórcio vencedor, o fato das maiores empresas de computação do mundo participarem deles é um sinal de que

Java poderá ser usada no futuro também para sistemas de tempo real.

Sistemas Operacionais

O mercado de sistemas operacionais de tempo real está passando por uma fase de rápidas mudanças. A teoria de escalonamento de tempo real, ignorada até alguns anos atrás, começa a ser incorporada aos sistemas operacionais. Desta forma, pode-se esperar para os próximos anos suportes de execução cada vez mais previsíveis. Ao mesmo tempo, Posix firmou-se como a interface padrão nesta área. Embora o próprio Posix seja grande e por vezes ambíguo, ele estabelece um padrão para as chamadas de sistema a serem oferecidas para aplicações de tempo real.

Variações do Linux para tempo real estão surgindo com força neste mercado. Como o capítulo 3 mostrou, muitos grupos de pesquisa em todo o mundo estão trabalhando no sentido de criar versões do Linux apropriadas para aplicações de tempo real. Enquanto alguns trabalhos nesta área buscam uma previsibilidade rigorosa para atender sistemas críticos, outros procuram melhorar o desempenho do escalonamento para melhor suportar aplicações com requisitos temporais brandos. Não é possível ignorar que a disponibilidade de um sistema operacional Unix completo, com fonte aberto e adaptado para tempo real, vai causar um grande impacto no mercado. Por exemplo, no momento em que este livro é escrito, é anunciado que o QNX será fornecido sem custo para aplicações não comerciais (http://get.qnx.com).

Com respeito aos sistemas distribuídos, ainda existe um longo caminho a ser percorrido. Embora RT-CORBA padronize interfaces e crie uma estrutura conceitual baseada em objetos, sua implementação depende dos serviços de escalonamento e comunicação tempo real fornecidos pelo sistema operacional e protocolos de comunicação subjacentes. A medida que a qualidade destes serviços aumentar, implementações do RT-CORBA também apresentarão melhor qualidade com respeito a previsibilidade temporal.

Apesar de todos estes esforços, a verdade é que o mercado de sistemas operacionais de tempo real continuará segmentado ainda por muitos anos. Com aplicações tão diversas quanto uma teleconferência e o controle de um motor elétrico apresentando restrições temporais, obviamente existe espaço para um grande conjunto de soluções. Estas incluem desde sistemas operacionais completos, adaptados para fornecer escalonamento do tipo melhor esforço, até núcleos totalmente previsíveis, capazes de garantir *"deadlines"* em aplicações críticas.

Escalonamento de Tempo Real

A evolução prevista com novos ambientes de larga escala e novas aplicações preconiza o surgimento de novas abordagens com algoritmos e testes de escalonamento

que possam tratar com estruturas complexas de tarefas e de recursos, definindo escalas que atendam granularidades variáveis de restrições temporais [Sta96]. As técnicas de escalonamento devem ser robustas ou ainda, flexíveis e adaptativas quando necessário.

Existe uma necessidade de testes e algoritmos mais robustos. Na chamada *ciência da garantia* testes e algoritmos, definidos segundo certas premissas, são válidos para determinados modelos de tarefas. Quando aplicados em um sistema, a atuação de um algoritmo é tida como correta se todas as premissas assumidas no modelo são válidas. Se algumas dessas premissas não são verificadas e mesmo assim as propriedades do algoritmo se mantém corretas, as restrições de tempo real da aplicação são atendidas e o algoritmo é dito robusto. O uso de um algoritmo robusto reduz, significativamente, a necessidade da caracterização da aplicação e de seu ambiente de execução nos esforços de análises e medidas para a validação do conjunto de tarefas que formam a sua carga [Sta96]. A eficiência e a robustez podem ser concretizadas facilmente se o grau de sucesso da validação não é o objetivo maior. Testes de aceitação (ou testes de escalonabilidade), por serem excessivamente pessimistas, apresentam um baixo grau de sucesso nestes novos e complexos ambientes.

As abordagens de *escalonamento adaptativo* ("*adaptive scheduling*") têm sido propostas no sentido de se adotar modelos mais flexíveis. Essas abordagens assumem que as condições do sistema são monitoradas, e as decisões do escalonamento são baseadas nas condições observadas [LSL94]. Modelos e abordagens de escalonamento adaptativos são empregados com objetivo de lidar com a incerteza na carga do sistema e a obtenção de uma *degradação suave* [CLL90], [GNM97], [MFO99], [TaT93]. Degradação suave em sistemas de tempo real, significa a manutenção das propriedades de previsibilidade na sobrecarga.

De forma diversa às abordagens mais clássicas, algumas abordagens adaptativas não necessitam o conhecimento *a priori* dos piores casos de tempo de computação das tarefas. A determinação destes tempos das tarefas, necessários em abordagens mais clássicas, sempre é um trabalho árduo.

Diversas técnicas adaptativas têm sido propostas recentemente. Por exemplo, uma técnica de adaptação pode ser baseada no ajuste dos períodos de tarefas periódicas em tempo de execução. A adaptação dos períodos de tarefas pode ser usada em alguns sistemas de controle realimentados ou também, através da mudança da freqüência de apresentação de quadros, em uma aplicação de vídeo sob demanda [KuM97]. A computação imprecisa [LSL94] — uma outra técnica que permite a combinação de garantia determinista com degradação suave — é usada para o tratamento de sobrecargas. Nessa técnica, cada tarefa é decomposta em duas partes: uma parte obrigatória e uma parte opcional. A parte obrigatória de uma tarefa deve ser completada antes do deadline da tarefa para produzir um resultado aproximado com uma qualidade aceitável. A parte opcional refina o resultado produzido pela parte obrigatória. Durante uma sobrecarga, um nível "mínimo" de operação do sistema pode ser garantido de forma determinista, através da execução apenas das partes obrigatórias.

Em [HaR95], foi proposto o conceito de "*deadline (m,k)-firm*", definindo que uma

tarefa periódica deve ter pelo menos *m "deadlines"* atendidos em cada janela de *k* ativações. O limite superior tolerado de perdas de *"deadlines"* é dado por *k-m*. Uma falha dinâmica é assumida no sistema quando esse limite é excedido. O DBP ("*Distance Based Priority Assignment*") é a heurística de escalonamento usada com essa técnica no sentido de minimizar o número de falhas dinâmicas. Essa heurística de escalonamento atribui as mais altas prioridades para as tarefas que estão próximas de exceder o limite superior especificado para as perdas de *"deadlines"*. Outras técnicas semelhantes "*deadline (m,k)-firm*" foram introduzidas na literatura [KoS95], [CaB97] e [BeB97]. Todas se utilizam do descarte de certas ativações de tarefas periódicas para aumentar a escalonabilidade do sistema.

Comunicação

Se considerarmos os sistemas de tempo real distribuídos, a comunicação desempenha um papel importantíssimo nos tempos de resposta que envolve uma aplicação nestes sistemas. Os objetivos de protocolos de comunicação em sistemas de tempo real são diferentes dos sistemas que não são de tempo real. Em sistemas mais convencionais, não tempo real, o aumento do desempenho na forma de taxas de transferências é o ponto chave. Por outro lado, na comunicação em sistemas de tempo real o que se procura é a obtenção de altas probabilidades que uma mensagem será entregue dentro de um deadline específico ou atendendo uma latência máxima.

Em sistemas críticos ("*hard real-time systems*") e, portanto na ótica da *ciência da garantia*, os protocolos de comunicação para tempo real devem apresentar um limite máximo na latência de uma mensagem durante uma comunicação. Em sistemas mais brandos, tais como multimídia e videoconferência, onde dados de diferentes naturezas estão sendo transmitidos, é evidente a necessidade da entrega de mensagens segundo certas restrições temporais. Ocasionais falhas em atender estas restrições temporais é perfeitamente admissível; porém, atrasos ou perdas de mensagens excessivos degradam a qualidade do serviço fornecido a nível da aplicação.

São duas as abordagens usadas para tratar comunicações em sistemas de tempo real. A primeira é baseada sobre a utilização do conhecimento sobre a máxima latência nas transferências de mensagens no sistema [ARS91]. As análises e escalas produzidas para o sistema consideram então este pior tempo de comunicação. Na segunda abordagem as mensagens estão sujeitas elas próprias a restrições temporais (*"deadlines"*, períodos, etc) e escalonadores atuam, segundo políticas de tempo real, em filas de emissão levando em conta estas restrições [ZhB94].

Sistemas de larga escala com diferentes tipos de aplicações, implicam na necessidade do suporte a vários tipos de comunicação, sujeitas a diferentes necessidades de garantias e de restrições de tempo. Novas tecnologias de comunicação estão surgindo para suprir essas necessidades. Exemplo destas tecnologias são as redes baseadas no protocolo IP, que inicialmente foram concebidas no sentido das técnicas de melhor

esforço e que estão sendo estendidas para implementar diferentes Qualidades de Serviço (QoS). Redes ATM também suportam diferentes classes de serviço. Além de serviços de melhor esforço estas tecnologias oferecem classes de s*erviços garantidos* que, com reservas de recursos, fornecem serviços garantidos fim-a-fim com o controle rígido da largura de banda e do retardo [SPG97]. Com base nestas tecnologias novos modelos de comunicação de tempo real deverão surgir, possibilitando a associação de parâmetros de QoS relacionados com tempo real a um canal de comunicação.

Metodologias

Atualmente, a complexidade dos sistemas computacionais exige cada vez mais o uso de metodologias para especificar, analisar, projetar e implementar. Os sistemas de tempo real em particular quando distribuídos, apresentam características que se encaixam nesta categoria. As metodologias de propósito geral, normalmente utilizadas na comunidade acadêmica e no setor industrial e de serviços, tem apresentado inconvenientes. As razões estão na inadequação da linguagem de modelagem que não inclui o conjunto especifico de requisitos necessários para representar os sistemas tempo real e, as dificuldades de implementação que não facilitam a ligação entre os conceitos usados numa modelagem de alto nível de abstração e os conceitos usados na implementação. Metodologias para sistemas de tempo real e que ao mesmo tempo acrescentam as potencialidades da orientação objeto têm sido propostas para remediar este problema e diminuir as dificuldades na construção destes sistemas. Destacam-se entre estas linguagens de modelagem e metodologias para tempo real a ROOM ("*Real-Time Object-Oriented Modeling*") [SGW94] e a UML-RT ("*Unified Modeling Language for Real-Time*") [SeR98]. Essas metodologias apresentam as seguintes características: seguir o paradigma de orientação-objeto; permitir a modelagem e a construção de software tempo real; fornecer modelos executáveis em todos os níveis de abstração; permitir um processo de desenvolvimento incremental e iterativo; e facilitar a documentação.

Neste mesmo contexto, técnicas de descrição formal tem sido adotadas num número crescente de aplicações, devido aos benefícios significativos que o rigor de seus formalismos traz para a produção de software de qualidade, sobretudo quando esta qualidade se expressa em termos de consistência, segurança e correção temporal. Muitas pesquisas e desenvolvimentos de ferramentas vem sendo realizadas neste campo nos últimos anos. Esta área é muito ampla e promissora e sai do escopo deste livro; os interessados podem procurar, na ampla bibliografia disponível da área, as referências [HeM96], [BCN95] e [Rus93] que destacamos por apresentarem uma visão geral das técnicas de descrição formal próprias para sistemas tempo real.

ANEXO A

Extensões em Esquemas de Prioridade Dinâmica

A.1 Testes para Escalonamentos com Prioridades Dinâmicas

A análise de escalonabilidade do EDF baseada no conceito de utilização, embora corresponda a um teste exato, é limitada na sua aplicação pela simplicidade do modelo de tarefas de sua definição. Os testes que se seguem apresentam como finalidade a extensão do uso do EDF para modelos mais complexos.

A.1.1 Deadline igual ao período

Quando considerados uma política baseada em deadlines e um conjunto de tarefas periódicas com deadlines relativos iguais aos seus respectivos períodos, qualquer análise em uma janela de tempo deve levar em conta a carga computacional representada pelas ativações dessas tarefas que devem ser executadas dentro dessa janela. Ou seja, essa carga corresponde às ocorrências de tarefas que apresentam deadlines absolutos menores ou iguais ao limite superior desse intervalo de tempo. Por exemplo, a necessidade de processador que uma tarefa T_i possui em um intervalo $[t,t+l]$ para que suas ativações completem antes ou em $t+l$ é dado por $\lfloor l/P_i \rfloor C_i$.

Para um conjunto de n tarefas periódicas, as instâncias liberadas e que completam no intervalo $[0, t]$ correspondem à *demanda de processador C(t)* fornecido por:

$$C(t) = \sum_{P_i \leq t}^{n} \left\lfloor \frac{t}{P_i} \right\rfloor C_i .$$

A escalonabilidade de um conjunto de tarefas executadas sob o EDF (política dirigida a deadlines), é garantida se e somente se, em qualquer intervalo $[0, t]$, a demanda de processador ($C(t)$) das instâncias que devem se completar nesse intervalo é menor ou igual ao tempo disponível t [JeS93]:

$$t \geq \sum_{P_i \leq t} \left\lfloor \frac{t}{P_i} \right\rfloor C_i . \qquad [A1]$$

Para aplicar o teste [A1], a inspeção de valores de *t* pode ser limitada aos tempos de liberação das tarefas dentro do *hiperperíodo H* do conjunto de tarefas considerado[1]. Mas isto ainda pode representar um grande número de valores de *t* a serem verificados.

O conceito de *"busy period"* pode ser útil na determinação de um conjunto mais reduzido de valores de *t*. Retomando então um *período ocupado* como um intervalo de tempo com execuções contínuas de tarefas e considerando o instante crítico em *t=0*, o pior caso de execução de uma tarefa T_i ocorre no primeiro *i-busy period* que inicia na origem (*t=0*) e termina com a execução da instância considerada. A idéia é que o *completion time* de uma instância de T_i com deadline *d* deva estar no fim do *i-busy period* no qual se executarão instâncias de outras tarefas que tenham deadlines menores ou iguais a *d*.

O intervalo de tempo *L* que ocorre entre *t=0* e o primeiro *"idle time"* do processador, é assumido como o maior *"busy period"* no sistema. O valor de *L* é obtido a partir de [Spu96]:

$$\begin{cases} L^{(0)} = \sum_{i=1}^{n} C_i , \\ L^{(m+1)} = W(L^{(m)}), \end{cases} \qquad [A2]$$

onde *W(t)* corresponde a carga cumulativa no intervalo [*0, t*], ou seja a soma dos tempos de computação de todas as instâncias que chegaram antes do tempo *t* :

$$W(t) = \sum_{i=1}^{n} \left\lceil \frac{t}{P_i} \right\rceil C_i . \qquad [A3]$$

Quando $L^{(m+1)} = L^{(m)}$ os cálculos em [A2] estão terminados[2].

Os valores de *t* que devem servir na inspeção de [A1] são dados então por $t \in \Im$, onde:

---

[1] *hiperperíodo* de um conjunto de tarefas periódicas corresponde ao mínimo múltiplo comum dos períodos das tarefas do conjunto.

[2] $L^{(m)}$ converge para *L* em um número finito de iterações se [Spu96]:

$\sum^{n}$ $\frac{C_i}{}$ ≤ 1.

$$\Im = \left\{ d_{ik} \,\middle|\, d_{ik} = kT_i \ \wedge \ d_{ik} \le min(L,H),\ 1 \le i \le n,\ k \ge 1 \right\}. \qquad [A4]$$

Um modelo mais realista onde ocorrem *"jitters"*, determina que uma tarefa T_i seja retida depois de sua chegada até um tempo máximo J_i para a sua liberação. As equações acima devem ser modificadas no sentido de expressar esses tempos. O efeito de *"jitters"* no pior caso pode provocar em [*0, t*] a interferência de instâncias que em situações normais teriam suas liberações fora desse intervalo. Com isto, a carga cumulativa *W(t)* correspondente ao intervalo [*0,t*] (carga que ocorre em [0, t]) deve, no pior caso, aumentar [Spu96]:

$$W(t) = \sum_{i=1}^{n} \left\lceil \frac{t + J_i}{P_i} \right\rceil C_i .$$

No mesmo sentido, a demanda de processador *C(t)* (carga com deadlines menor ou igual a *t*, ou seja, concluída em [0,t]) passa a ser fornecida por:

$$C(t) = \sum_{P_i \le t + J_i} \left\lfloor \frac{t + J_i}{P_i} \right\rfloor C_i .$$

A equação [A1] é modificada igualmente:

$$t \ge \sum_{P_i \le t + J_i} \left\lfloor \frac{t + J_i}{P_i} \right\rfloor C_i .$$

Os valores de *t* continuam sendo definidos pelo conjunto $\Im$ de [A4].

A.1.2 Deadlines Arbitrários

Ao considerarmos um conjunto de tarefas periódicas com deadlines relativos (D_i) arbitrários e escalonadas segundo o EDF, o teste anterior deixa de ser suficiente. O teste definido pelas condições [A1] e [A4] pode ser facilmente estendido para tratar tarefas com deadlines arbitrários [Spu96].

A carga de trabalho acumulada em [*0, t*] deve levar em consideração esses deadlines arbitrários. Considerando $t - D_i$ o instante de liberação da última instância de T_i com garantia de execução em [*0, t*], a *demanda de processador C(t)* proposto por esse conjunto de tarefas é dada por :

$$C(t) = \sum_{D_i \le t} \left(1 + \left\lfloor \frac{t - D_i}{P_i} \right\rfloor \right) C_i . \qquad [A5]$$

Com isto uma condição necessária e suficiente para tarefas periódicas possuindo deadlines arbitrários e escalonadas pelo EDF, é obtida através de:

$$t \ge \sum_{D_i \le t} \left(1 + \left\lfloor \frac{t - D_i}{P_i} \right\rfloor \right) C_i \qquad [A6]$$

$$e \qquad \Im = \left\{ d_{ik} \,\middle|\, d_{ik} = kT_i + D_i,\ d_{ik} \le min(L, H), 1 \le i \le n, k \ge 0 \right\}, \qquad [A7]$$

onde *t* assume valores de $\Im$ para a inspeção da condição do teste. A ocorrência de *"jitter"* no modelo redefine as equações [A5] e [A6] [Spu96]:

$$C(t) = \sum_{D_i \le t + J_i} \left(1 + \left\lfloor \frac{t + J_i - D_i}{P_i} \right\rfloor \right) C_i \qquad [A8]$$

$$t \ge \sum_{D_i \le t + J_i} \left(1 + \left\lfloor \frac{t + J_i - D_i}{P_i} \right\rfloor \right) C_i \qquad [A9]$$

tarefas periódicas	C_i	P_i	D_i
tarefa A	2	10	6
tarefa B	2	10	8
tarefa C	8	20	16

Tabela I: Exemplo da figura 2.7 (capítulo 2)

Para ilustrar esse teste para modelos de tarefas com deadlines arbitrários em esquemas de prioridade dinâmica, considere o conjunto de tarefas descrito na *tabela I*. Usando a equação [A5] podemos determinar para essas tarefas a demanda de processador *C(t)*:

$$C(t) = \left(1 + \left\lfloor \frac{t-6}{10} \right\rfloor \right).2 + \left(1 + \left\lfloor \frac{t-8}{10} \right\rfloor \right).2 + \left(1 + \left\lfloor \frac{t-16}{20} \right\rfloor \right).8$$

Falta obtermos o valor *t* a ser usado no teste [A6]. Para isto, a carga cumulativa (tarefas que chegam antes e em *t*) é necessária:

$$W(t) = \left\lceil \frac{t}{10} \right\rceil .2 + \left\lceil \frac{t}{10} \right\rceil .2 + \left\lceil \frac{t}{20} \right\rceil .8 \cdot$$

Aplicando [A2] no sentido de determinar *L* (o maior *"busy period"*) :

$$L^{(0)} = \sum C_i = 12$$
$$L^{(1)} = W(L^{(0)}) = 16$$
$$L^{(2)} = W(L^{(1)}) = 16$$

Logo *L=16*, com isto obtemos o conjunto dos valores possíveis de *t* [A7]:

$$\Im = \left\{ d_{ik} \,\middle|\, d_{ik} \leq 16 \right\}$$

Considerando as tarefas da *tabela I* e a figura 2.7 (capítulo 2) verificamos que $\Im$ = *{6, 8, 16}*. Com isto obtemos as seguintes demandas de processador: *C(6) = 2, C(8) = 6* e *C(16) = 16*. Em todos esses valores de t é verificado o teste [A6], ou seja, $t \geq C(t)$. Logo o conjunto apresentado na *tabela I* e figura 2.7 é escalonável também em esquemas de prioridade dinâmica.

A.2 Compartilhamento de Recursos em Políticas de Prioridade Dinâmica

A técnica chamada de *Stack Resource Policy* (SRP) foi introduzida em [Bak91] para controlar em políticas de prioridade dinâmica, o compartilhamento de recursos de uma forma previsível. Basicamente, esse método estende os resultados do *"Priority Ceiling Protocol"* para esquemas de prioridade dinâmica. Ou seja, as inversões de prioridades se limitam a um bloqueio por ativação da tarefa.

A.2.1 Política de Pilha (*Stack Resource Policy*)

O SRP é similar ao IPCP na característica de que uma tarefa é bloqueada só no instante em que tenta a preempção. Esse bloqueio antecipado – diferente do PCP onde uma tarefa só é bloqueada quando tenta entrar em uma seção crítica – reduz a necessidade de trocas de contextos de tarefas, simplificando a implementação do protocolo.

Descrição do Protocolo

Em escalonamentos dirigidos por prioridades dinâmicas, a prioridade p_i de uma tarefa T_i indica a sua urgência em um determinado instante *t*. Quando usada a atribuição EDF, a prioridade p_i para refletir a urgência de T_i no instante *t*, é função direta ou assume o valor do deadline absoluto d_i desta tarefa. Além da prioridade p_i, quando usado o SRP, a tarefa T_i deve apresentar um *nível de preempção* π_i que é um parâmetro estático. Os níveis de preempção estabelecem regras para preempções de tarefas: uma tarefa T_j só poderá ser interrompida por uma tarefa T_i se $(\pi_i > \pi_j) \wedge (p_i > p_j)$. A utilidade dos níveis de preempção é que, por serem estáticos, esses parâmetros permitem a prevenção de bloqueios em esquemas de prioridades dinâmicas.

No escalonamento EDF, usando o SRP, os valores de π são definidos na ordem inversa dos deadlines relativos. Supondo as tarefas T_i e T_j ilustradas com dois casos de chegada na figura A.1. É tirado desta figura que π_i é maior que π_j pois $D_i < D_j$. No caso (a) da figura, a tarefa T_i que chega em t_2 consegue a preempção porque $\pi_i > \pi_j$ ($D_i < D_j$) e $p_i > p_j$ ($d_i < d_j$). No caso (b) da figura A.1 a preempção não ocorre por que uma das duas condições necessárias não se verifica: T_i que chega em t_2 não possui prioridade maior que T_j ($d_i > d_j$).

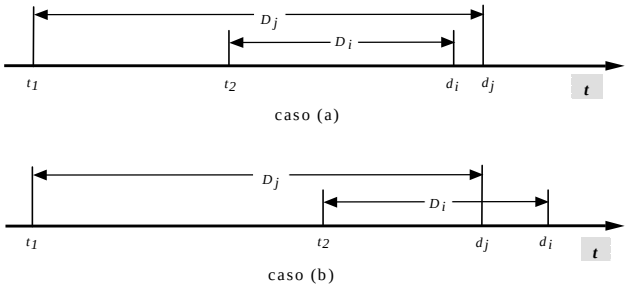


Figura A.1: Níveis de Preempção no EDF

Cada recurso compartilhado possui um valor máximo de unidades de recurso para alocação (*max* n_r). O SRP também, a exemplo do PCP, atribui a cada recurso compartilhado uma prioridade teto ($\lceil R_r \rceil$). A diferença é que, neste caso, este parâmetro é dinâmico: o *"ceiling"* corrente de R_r tem o valor determinado a partir do número de unidades de alocação disponíveis no recurso R_r. Se n_r representa o número de unidades de R_r disponíveis em um dado instante e, $\mu_{r,i}$ corresponde as necessidades de uma tarefa T_i em termos do recurso R_r então o *ceiling* é dado por:

$$\lceil R_r \rceil_{(n_r)} = max\left[\{0\} \cup \left\{\pi_i \mid n_r < \mu_{r,i}\right\}\right].$$

Na expressão acima, quando todas as unidades de R_r estão disponíveis, o valor

corrente de *"ceiling"* de R_r é mínimo ($\lceil R_r \rceil = 0$). Se as unidades de alocação de R_r não são suficientes para que algumas tarefas se executem, então o valor do *"ceiling"* assumirá o mais alto nível de preempção dentre estas tarefas ($\lceil R_r \rceil = \pi_j$). O *"ceiling"* do sistema (Π) é assumido como o máximo *"ceiling"* entre todos os *m* recursos compartilhados no sistema:

$$\Pi = max\left(\lceil R_r \rceil \,\middle|\, r = 1,2,\ldots,m\right).$$

No SRP a regra que garante a não ocorrência de múltiplas inversões de prioridade determina que uma tarefa T_i não pode se executar até que os recursos necessários para a sua execução estejam disponíveis, ou seja, $\pi_i > \Pi$. De uma maneira geral, uma tarefa T_i pode preemptar uma tarefa T_j em execução quando: (i) for a tarefa mais prioritária ($p_i > p_j$); (ii) o seu nível de preempção for maior do que a tarefa se executando ($\pi_i > \pi_j$); e por fim, (iii) se o seu nível de preempção for superior ao *"ceiling"* do sistema ($\pi_i > \Pi$).

Esse método impõe que, quando da ativação de uma tarefa T_i, fique disponível as informações de suas necessidades em recursos (ou seja, os seus valores de $\mu_{r,i}$ para cada recurso R_r) de modo a se determinar previamente os valores de *"ceiling"* de cada recurso R_r em diferentes situações de alocação. Quando as condições (i), (ii) e (iii) são verificadas, a tarefa T_i tem satisfeita as suas necessidades em recursos disponíveis e poderá, portanto, iniciar sua execução sem ser bloqueada por tarefas menos prioritárias. Muito embora, os recursos só venham a ser alocados quando requisitados durante a execução de T_i.

A figura A.2 e suas tabelas apresentam uma aplicação do *"Stack Resource Policy"*. Os parâmetros das tarefas T_1, T_2 e T_3 e suas necessidades de recursos são descritos nas tabelas ($\mu_{r,i}$). Os recursos R_1, R_2 e R_3 são mostrados com suas máximas unidades para alocação ($max\ n_r$). Os *"ceilings"* são determinados em tabela a partir das unidades de alocação disponíveis nos recursos.

Uma escala obtida sob o SRP é descrita na figura A.2. Em *t=2*, a tarefa menos prioritária T_3 começa a executar porque possui seu nível de preempção maior que o *"ceiling"* do sistema ($\Pi = 0$). T_3 entra na sua primeira seção crítica em *t =3* e ocupa uma unidade do recurso R_3. Com isto o *"ceiling"* do sistema passa ao valor de π_2, porque T_2 não consegue ter suas necessidades atendidas ($\mu_{3,2} = 2$). Logo, em *t = 4*, T_2 é bloqueada pelo teste de preempção e não consegue executar. A tarefa T_3 continua a sua execução e em *t = 4* entra em sua seção crítica aninhada ocupando uma unidade de R_2. Com esta nova seção de T_3, a execução de T_1 é postergada em *t=5* pela não disponibilidade de suas necessidades no recurso R_2 ($\mu_{2,1} = 3$). Como conseqüência, o *"ceiling"* do sistema cresce para π_1. Quando T_3 libera o recurso R_2, o *"ceiling"* do sistema torna se π_2 o que permite que T_1 interrompa T_3. T_1 é executada completamente porque seu nível de preempção é maior que o *"ceiling"* do sistema. Com a conclusão de T_1, T_3 reassume e quando libera o recurso R_3 em *t=15*, o *"ceiling"* do sistema baixa para $\Pi = 0$ e a tarefa T_2 pode interromper T_3.

R_r	*Max* n_r	$\mu_{r,1}$	$\mu_{r,2}$	$\mu_{r,3}$	$\lceil R_r \rceil_{(3)}$	$\lceil R_r \rceil_{(2)}$	$\lceil R_r \rceil_{(1)}$	$\lceil R_r \rceil_{(0)}$
R_1	**1**	**1**	**1**	-	-	-	**0**	π_1
R_2	**3**	**3**	-	**1**	0	π_1	π_1	π_1
R_3	**2**	-	**2**	**1**	-	0	π_2	π_2

T_1	T_2	T_3
....	....	....
pedido (R_1,1)	pedido (R_3,2)	pedido (R_3,1)
libera (R_1)	libera (R_3)	pedido (R_2,1)
pedido (R_2,3)	pedido (R_1,1)	libera (R_2)
libera (R_2)	libera (R_1)	libera (R_3)
....	....	....

Prioridades : $p_1 > p_2 > p_3$

Níveis de preempção : $\pi_1 > \pi_2 > \pi_3$ ($D_1 < D_2 < D_3$)

Semáforos : S_1 - S_2 - S_3 -

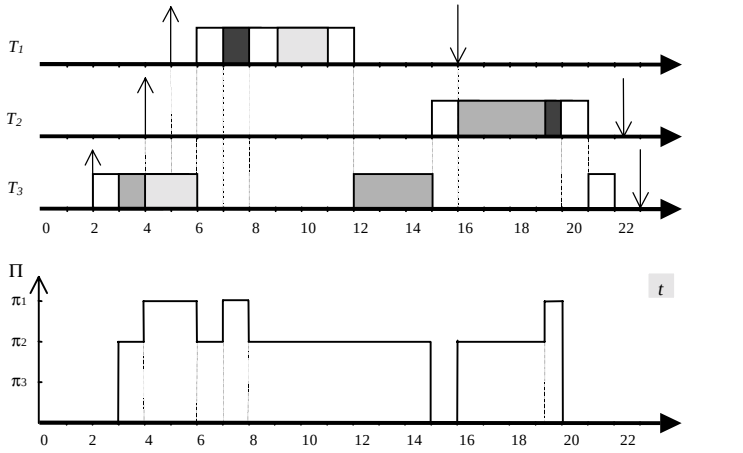


Figura A.2 : Exemplo de escala com o SRP

Extensões de Testes de Escalonabilidade Tomando como Base o SRP

O SRP foi concebido para ser usado com políticas de prioridades dinâmicas como o EDF. Em [Bak91], o teste de escalonabilidade do EDF (equação [3], capítulo 2) foi estendido considerando o SRP:

$$\sum_{j=1}^{i} \frac{C_j}{P_j} + \frac{B_i}{P_i} \leq 1, \quad \forall i, 1 \leq i \leq n.$$

O somatório da equação acima representa a utilização de T_i e de todas as tarefas T_j que possuam nível de preempção maior que o seu ($\pi_j > \pi_i$). Esta condição considera o instante crítico na origem o que implica nas tarefas ordenadas segundo valores decrescentes de seus níveis de preempção, ou seja, no sentido crescente de seus deadlines relativos. O termo B_i/P_i corresponde a utilização perdida com bloqueio por tarefa de nível de preempção menor que π_i.

Quando é assumido deadlines relativos menores que os correspondentes períodos no modelo de tarefas, o teste proposto por Baker toma a seguinte forma:

$$\sum_{j=1}^{i} \frac{C_j}{D_j} + \frac{B_i}{D_i} \leq 1, \quad \forall i, 1 \leq i \leq n.$$

Os testes acima, conseguidos a partir de extensões de [3] (capítulo 2), dependem apenas de parâmetros estáticos e portanto, podem ser verificados em tempo de projeto.

Outros testes mais precisos e gerais para políticas de prioridade dinâmica, apresentados neste *Anexo*, devem também ser modificado para expressar os bloqueios que tarefas podem sofrer de tarefas menos prioritárias. É o caso dos testes baseados em *demanda de processador* [Spu96], onde tarefas com deadlines relativos arbitrários e sofrendo de *release "jitters"*, quando compartilham recursos sob o SRP, podem ter suas escalonabilidades verificadas a partir de:

$$t \geq \sum_{D_j \leq t + J_j} \left(1 + \left\lfloor \frac{t + J_j - D_j}{P_j} \right\rfloor \right) C_j + \left(\left\lfloor 1 + \frac{t + J_i - D_i}{P_i} \right\rfloor \right) B_i \qquad \forall i,\ 1 \leq i \leq n$$

onde t é definido em $\Im$ (conjunto [A.7] indicado no item A.1.2.):

$$\Im = \left\{ d_{ik} \,\middle|\, d_{ik} = kT_i + D_i,\ d_{ik} \leq min(L, H),\ k \geq 0 \right\}.$$

No teste acima o somatório determina a demanda de carga de tarefas de nível de prioridade maior ou igual a i, ou seja, tarefas que apresentem $D_j - J_j \leq D_i - J_i$. O termo que envolve B_i corresponde ao pior caso de bloqueio imposto sobre T_i, durante o intervalo t, por tarefas de menor nível de preempção.

O máximo bloqueio B_i que uma tarefa T_i pode experimentar em uma ativação, deve corresponder a duração da maior seção crítica entre as que podem bloquear T_i. A exemplo do PCP, um bloqueio só pode ser caracterizado quando uma tarefa T_j menos prioritária executando em uma seção crítica de duração $D_{j,r}$, guardada pelo semáforo S_r, tiver o seu nível de preempção menor que o de T_i ($\pi_i > \pi_j$) e o recurso guardado apresentar o *"ceiling"* máximo igual ou maior que π_i. O *"ceiling"* máximo de um semáforo é assumido quando o número de unidades livres do recurso for nulo. O bloqueio máximo é dado então por:

$$B_i = \max_{j,r}\left\{ D_{j,r} \,\middle|\, \left(\pi_j < \pi_i\right) \wedge \left(\left\lceil R_{S_r} \right\rceil_{(0)} \geq \pi_i \right) \right\}$$

A.3 Escalonamento de tarefas aperiódicas com políticas de prioridade dinâmica

O escalonamento de tarefas aperiódicas e periódicas pode também ser feito sob atribuições dinâmica de prioridades. As tarefas periódicas neste caso são escalonadas usando o algoritmo *"Earliest Deadline First"* (EDF). A justificativa encontrada para a extensão do EDF de modo a permitir esquemas híbridos de escalonamento é fundamentada nos maiores limites de escalonabilidade desta política se comparada com as de prioridade fixa. Em [Spu96] a título de exemplo é apresentada uma carga periódica com utilização $U_p=0,6$. Se a atribuição de prioridades é feita pelo RM e o servidor de prioridade fixa usado é o SS, a máxima Utilização do servidor é dada por $U_S=0,1$. Se o EDF é usado a utilização do processador vai para 100% e a máxima utilização do servidor acaba sendo de $U_S=0,4$.

A.3.1 Servidores de Prioridade Dinâmica [SpB96]

Dentre as muitas abordagens introduzidas em [SpB96] para servidores de prioridade dinâmica (*"Dynamic Priority Exchange"*, *"Dynamic Sporadic Server"*, *"Improved Priority Exchange"*, etc.), uma grande parte é extensão dos servidores de prioridade fixa apresentados no item 2.8 (capítulo 2). Neste item, no sentido de evitar textos repetitivos, é apresentado como uma ilustração destes servidores o *"Dynamic Sporadic Server"* (DSS); os leitores que quiserem um estudo mais exaustivo sobre o assunto devem recorrer a [SpB96].

Servidor Esporádico Dinâmico

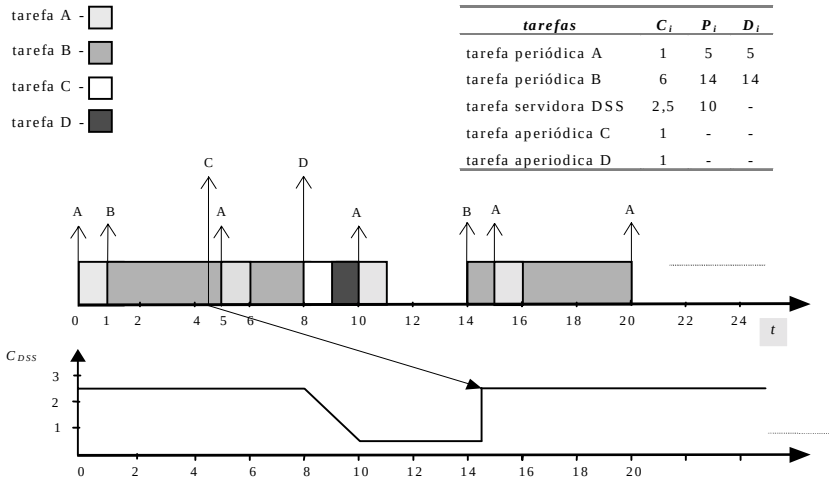
O *"Dynamic Sporadic Server"* (DSS) estende o algoritmo do SS para atuar com políticas de prioridade dinâmica – mais precisamente, o EDF. Esta abordagem híbrida abre espaço para execuções aperiódicas em escalas ordenadas pelo EDF também usando o conceito de servidor: uma tarefa servidora é então criada com capacidade C_S e período P_S. A servidora DSS preserva a sua capacidade enquanto não ocorrem requisições aperiódicas. Como a sua homônima de prioridade fixa a capacidade consumida não é preenchida no início do período da servidora DSS, mas no tempo de preenchimento RT correspondente.

A prioridade e o preenchimento da capacidade da servidora DSS são determinados pelas seguintes ações [SpB96, But97]:

- Quando a servidora é criada, a sua capacidade C_S é iniciada no seu máximo valor.
- O tempo de preenchimento e o deadline d_S da tarefa servidora são determinados quando temos $C_S > 0$ e existe uma requisição aperiódica pendente. Se t_a é o instante de tempo em que se verificam estas condições, então $RT = d_S = t_a + P_S$.
- A quantidade de capacidade a ser preenchida em RT é obtida quando a última requisição aperiódica é completada ou a capacidade C_S da servidora é totalmente consumida. Se no tempo t_f ocorre uma dessas duas situações, então a quantidade de capacidade a ser preenchida é igual a capacidade consumida em $[t_a, t_f]$.

A limitação imposta pela servidora DSS sobre a utilização da carga periódica do sistema é provado em [Spu96] como idêntica a de uma tarefa periódica de período P_S e tempo de computação C_S, ou seja: $U_p \leq 1 - U_S$.

A figura A.3 ilustra um exemplo de uso da técnica DSS – que é o mesmo utilizado quando da apresentação do servidor SS de prioridade fixa (seção 2.8.1). O conjunto de tarefas descrito na tabela da figura é para ser escalonado segundo uma atribuição EDF (dinâmica). a tarefa servidora DSS é também definida com capacidade $C_S=2,5$ e período $P_S = 10$.



tarefas	C_i	P_i	D_i
tarefa periódica A	1	5	5
tarefa periódica B	6	14	14
tarefa servidora DSS	2,5	10	-
tarefa aperiódica C	1	-	-
tarefa aperiodica D	1	-	-

Figura A.3: Algorítmo "Dynamic Sporadic Server"

Em $t = 0$, a tarefa A, na sua primeira ativação apresentando deadline absoluto mais próximo ($d_{A,1}= 5$), assume o processador. Na seqüência, quando A conclui, a tarefa periódica B com deadline em $t = 14$ passa a ser executada. Em $t = 4,5$, com a chegada

de uma requisição C a servidora DSS é liberada com o deadline absoluto em $t = 14,5$ ($d_S = RT = t_a + P_S$). No caso, t_a coincide com o tempo de chegada da requisição C porque $C_S > 0$ em $t = 4,5$. Embora a capacidade da servidora seja suficiente, a tarefa B não é interrompida porque esta última possui o deadline mais próximo ($d_{B,1}=14$). A tarefa A, por sua vez, interrompe B em $t = 5$, durante sua segunda ativação ($d_{A,2}= 10$). Em seguida, no término de A, a tarefa B reassume concluindo em $t=8$. No instante $t=8$, a servidora DSS começa a executar a tarefa C com o deadline absoluto em $t=14,5$. Em $t=8$, chega também uma requisição D que é executada em seguida pela tarefa servidora com o mesmo deadline ($RT=14,5$). Isto ocorre porque, quando liberado, o servidor DSS deve executar todas as requisições aperiódicas pendentes enquanto a condição $C_S > 0$ for mantida. Se compararmos a escala obtida na figura A.3 com a escala do servidor esporádico estático, para o mesmo conjunto de tarefas (figura 2.17, capítulo 2, seção 2.8.1), verificamos que o DSS apresenta um desempenho inferior ao seu similar estático. Em termos de resposta imediata o DSS também é pior.

A figura A.4 mostra o mesmo conjunto de tarefas mas com a servidora DSS projetada com capacidade $C_S=1$. Neste caso, a tarefa aperiódica C consome toda a capacidade da servidora em sua execução que inicia em $t=8$. A tarefa D terá a condição de $C_S > 0$ somente em $t=14,5$, portanto com um t_a diferente de seu tempo de chegada. Com isto o deadline da tarefa D passa a ser em $t=24,5$ e a sua execução sob o EDF ocorre em $t=14,5$, interrompendo a tarefa B na sua segunda ativação ($d_{B,2}=28$). A tarefa *A*, por sua vez, interrompe a requisição D em $t=15$ por apresentar deadline mais próximo ($d_{A,3}= 20$). A requisição D reassume e conclui em $t=16,5$. O DSS pode tratar com tarefas aperiódicas firmes e a análise para a garantia dinâmica não difere muito das usadas em servidores de prioridade fixa.

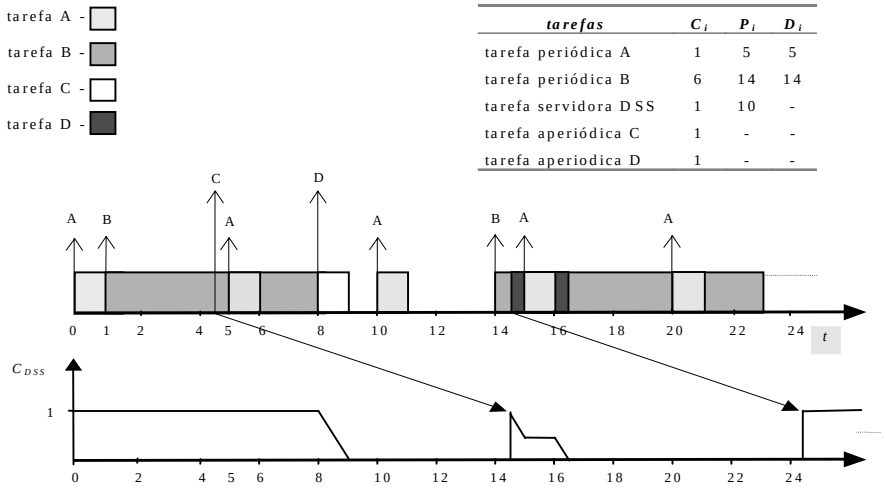


Figura A.4: "Dynamic Sporadic Server" com capacidade menor

ANEXO B

Sistemas Operacionais de Tempo Real na Internet

O mercado de sistemas operacionais de tempo real é fragmentado. Uma consequência direta disto é a grande quantidade de soluções disponíveis. Este anexo contém uma lista com cerca de 100 sistemas. Além do nome de cada sistema aparece na lista o nome do fornecedor e o endereço na Internet. É importante observar que esta lista inclui sistemas operacionais dos mais variados tipos, sendo que alguns não seriam considerados realmente de tempo real em um teste mais rigoroso. Entretanto, em todos eles, o fornecedor sugere o seu uso em aplicações de tempo real de algum tipo. Como o conteúdo da Internet é dinâmico, esta lista deve ser considerada como um ponto de partida para uma pesquisa sobre o assunto, e não a palavra final. Certamente no momento que este texto estiver sendo lido, a lista já estará desatualizada.

Listas similares podem ser encontradas em vários endereços da Internet. Em particular, a lista no "The IEEE Computer Society Technical Comite on Real-Time Systems Home Page", em http://www.cs.bu.edu/pub/ieee-rts/, aponta para os principais SOTR existentes.

Um excelente levantamento dos SOTR disponíveis pode ser encontrado na revista eletrônica "Dedicated Systems Magazine" (ex-"Real Time magazine", ela mudou de nome no início de 2000). O endereço da revista é

http://www.realtime-info.com/encyc/market/rtos/rtos_home.htm

e uma lista com mais de 100 SOTR aparece em

http://www.realtime-info.com/encyc/market/rtos/rtos.htm.

Aqui está a lista de Sistemas Operacionais de Tempo Real:

µC/OS-II fornecido por White Horse Design
http://www.uCOS-II.com
µITRON fornecido por TRON Association - ITRON Technical Committee
http://tron.um.u-tokyo.ac.jp/TRON/ITRON
AIX fornecido por IBM
http://www.austin.ibm.com/software/OS/index.html
AMX fornecido por KADAK Products Ltd
http://www.kadak.com/html/kdkp1010.htm

Ariel fornecido por Microware Systems Corporation
http://www.microware.com/ProductsServices/Technologies/ariel.html
ARTOS fornecido por Locamation
http://www.locamation.com/index.html
ASP6x fornecido por DNA Enterprises, Inc.
http://www.dnaent.com/c6x/press2.htm
Brainstorm Object eXecutive fornecido por Brainstorm Engineering Company
http://www.braineng.com/docs/boxhead.html
Byte-BOS fornecido por Byte-BOS Integrated Systems
http://www.bytebos.com/bytebos.htm
C Executive fornecido por JMI Software Systems Inc.
http://www.jmi.com/cexec.html
Chimera fornecido por The Robotics Institute Carnegie Mellon University
http://www.cs.cmu.edu/afs/cs.cmu.edu/project/chimera/www/home.html
ChorusOS fornecido por Sun Microsystems
http://www.sun.com/chorusos/
CMX fornecido por CMX Company
http://www.cmx.com/rtos.htm#rtx
CORTEX fornecido por Australian Real Time Embedded Systems (ARTESYS)
http://www.artesys.com.au
CREEM fornecido por GOOFEE Systems
http://www.goofee.com/creem.htm
CRTX fornecido por StarCom
http://www.n2.net/starcom/index.html
DDC-I Ada Compiler Systems (DACS) fornecido por DDC-I, Inc.
http://www.ddci.com/
Diamond fornecido por 3L
http://www.threel.co.uk
eCos fornecido por Cygnus Solutions
http://www.cygnus.com/ecos/
Elate(RTM) fornecido por Tao
http://www.tao-group.com
Embedded DOS 6-XL fornecido por General Software, Inc.
http://www.gensw.com/PAGES/EMBEDDED/EDOS6XL.HTM
EOS fornecido por Etnoteam S.p.A.
http://www.etnoteam.it/eos/eos.html
ERCOS EK fornecido por ETAS GmbH & Co.KG
http://www.etas.de/produkte/embedded_control/ercostext.htm
EspresS-VM fornecido por Mantha Software, Inc.
http://www.manthasoft.com
EUROS fornecido por Dr. Kaneff Engineering Consultants
http://www.directories.mfi.com/embedded/siemens/kaneff.htm
Fusion OS fornecido por Pacific Softworks
http:///www.pacificsw.com

Granada fornecido por Ingenieursbureau B-ware
http://www.b-ware.nl
Hard Hat Linux fornecido por MontaVista Software Inc
http://www.mvista.com
Harmony Real-Time Operating System fornecido por Institute for Information Technology, National Research Council of Canada
http://wwwsel.iit.nrc.ca/projects/harmony/
Helios fornecido por Perihelion Distributed Software
http://www.perihelion.co.uk/helios/
HP-RT fornecido por Hewlett-Packard
http://www.hp.com/go/hprt
Hyperkernel fornecido por Nematron Corporation
http://www.hyperkernel.com/
icWorkshop fornecido por Integrated Chipware
http://www.chipware.com
Inferno fornecido por Lucent Technologies
http://www.lucent.com/inferno
INTEGRITY fornecido por Green Hills Software, Inc.
http://www.ghs.com/html/integrity.html
INtime (real-time Windows NT), iRMX fornecido por Radisys Corp.
http://www.radisys.com/products/intime/index.html
IRIX fornecido por Silicon Graphics, Inc.
http://www.sgi.com/real-time/products.html#irix
iRMX III fornecido por RadiSys Corporation
http://www.radisys.com/products/irmx/irmx.html
ITS OS fornecido por In Time Systems Corporation
http://www.intimesys.com
Jbed fornecido por esmertec ag
http://www.esmertec.com
JOS fornecido por JARP
http://www2.siol.net/ext/jarp
Joshua fornecido por David Moore
http://www.moore160.freeserve.co.uk
LP-RTWin Toolkit fornecido por LP Elektronik GmbH
http://www.lp-elektronik.com/products/evxwin.htm
LP-VxWin fornecido por LP Elektronik GmbH
http://www.lp-elektronik.com/products/etoolkit.htm
LynxOS fornecido por Lynx Real-Time Systems
http://www.lynx.com/products/lynxos.html
MC/OS runtime environment fornecido por Mercury Computer Systems, Inc.
http://www.mc.com/Data_sheets/mcos-html/mcos.html
MotorWorks fornecido por Wind River Systems Inc.
http://www.wrs.com
MTEX fornecido por Telenetworks
http://www.telenetworks.com/

Nucleus PLUS fornecido por Accelerated Technology Inc.
http://www.atinucleus.com/plus.htm
OS-9 fornecido por Microware Systems Corp.
http://www.microware.com/ProductsServices/Technologies/os-91.html
OS/Open fornecido por IBM Microelectronics North American Regional Sales Office
http://www.chips.ibm.com/products/embedded/tools/osopen.html
OSE fornecido por Enea OSE Systems
http://www.enea.com
OSEK/VDX fornecido por University of KarlsruheInstitute of Industrial Information Systems
http://www-iiit.etec.uni-karlsruhe.de/~osek
PDOS fornecido por Eyring Corporation Systems Software Division
http://www.eyring.com/pdos/
PERC - Portable Executive for Reliable Control fornecido por NewMonics Inc.
http://www.newmonics.com/WebRoot/perc.info.html
pF/x fornecido por Forth, Inc.
http://www.forth.com/Content/Products/cFData.htm
PowerMAX OS fornecido por Concurrent Computer Corporation
http://www.ccur.com/product_info/index.html#anchor913878
Precise/MQX fornecido por Precise Software Technologies Inc
http://www.psti.com/mqx.htm
PRIM-OS fornecido por SSE Czech und Matzner
http://www.sse.de/primos
pSOS, pSOSystem fornecido por Integrated Systems, Inc.
http://www.isi.com/
PXROS fornecido por HighTec EDV Systeme GmbH
http://www.hightec-rt.com
QNX fornecido por QNX SOFTWARE SYSTEMS EUROPE
http://www.qnx.com/products/os/qnxrtos.html
QNX/Neutrino fornecido por QNX Software Systems, Ltd.
http://www.qnx.com/products/os/neutrino.html
Real-time Extension (RTX) for Windows NT fornecido por VenturCom, Inc
http://www.vci.com/products/vci_products/vci_products.html
Real-Time Software fornecido por Encore Real Time Computing Inc.
http://www.encore.com
REAL/IX PX fornecido por Modular Computer Services, Inc.
http://www.modcomp.com/c+c/cover2.html
REALTIME CRAFT fornecido por TECSI
http://www.tecsi.com
Realtime ETS Kernel fornecido por Phar Lap Software, Inc.
http://www.pharlap.com/html/body/8pager.htm#ANCH3
RMOS fornecido por Siemens AG
http://www.ad.siemens.de/support/html_00/index.shtml

Roadrunner fornecido por Cornfed Systems, Inc.
http://www.cornfed.com
RT-Linux fornecido por New Mexico Tech
http://www.rtlinux.org
RT-mach fornecido por Carnegie Mellon University
http://www.cs.cmu.edu/~rtmach
RTEMS fornecido por OAR Corporation
http://www.rtems.com
RTKernel-C fornecido por On Time Informatik GmbH
http://www.on-time.com
RTMX O/S fornecido por RTMX Inc.
http://www.rtmx.com/
RTOS-UH/PEARL fornecido por Institut fuer Regelungstechnik, Universitaet Hannover
http://www.irt.uni-hannover.de/rtos/rtos-gb.html
RTTarget-32 fornecido por On Time Informatik GmbH
http://www.on-time.com
RTX-51, RTX-251, RTX-166 fornecido por Keil Electronik GmbH
http://www.keil.com/products.htm
RTXC fornecido por Embedded System Products, Inc.
http://www.esphou.com
RTXDOS fornecido por Technosoftware AG
http://www.technosoftware.com
Rubus OS fornecido por Arcticus Systems AB
http://www.arcticus.se/rubus.htm
RxDOS fornecido por Api Software
http://www.rxdos.com
Smx fornecido por Micro Digital, Inc
http://www.smxinfo.com/smx/smx.htm
SORIX 386/486 fornecido por Siemens AG
http://www.siemens.de
SPOX fornecido por Spectron Microsystems, Inc.
http://www.spectron.com/products/spox/index.htm
SunOS, Solaris fornecido por Sun
http://www.spectron.com/products/spox/index.htm
Supertask! fornecido por U S Software
http://www.ussw.com
SwiftOS fornecido por Forth, Inc.
http://www.forth.com/Content/Products/SwiftX/SwiftX.htm
ThreadX - the high-performance real-time kernel fornecido por Express Logic, Inc.
http://www.expresslogic.com/products.html
Tics fornecido por Tics Realtime
http://www.concentric.net/~Tics/ticsinfo.htm
TNT Embedded Tool Suite fornecido por Phar Lap Software, Inc.
http://www.pharlap.com/html/tnt.html

Tornado/VxWorks fornecido por Wind River Systems Inc
http://www.wrs.com/products/html/tornado.html
TSX-32 fornecido por S&H Computer Systems, Inc
http://www.sandh.com/os.htm
velOSity fornecido por Green Hills Software, Inc.
http://www.ghs.com/html/velosity.html
Virtuoso fornecido por Eonic Systems
http://www.eonic.com
VRTX fornecido por Microtec Research
http://www.microtec.com/products/vrtx.html
Windows CE fornecido por Microsoft Inc.
http://WWW.eu.microsoft.com/windowsce/embedded/default.asp
XOS/IA-32 fornecido por TMO NIIEM
http://www.nexiliscom.com/osintro.html
XTAL fornecido por Axe, Inc.
http://www.axe-inc.co.jp

ANEXO C

Sintaxe e Semântica da Linguagem Esterel

C.1 Módulos e submódulos

Os programas Esterel são estruturados a partir das suas unidades básicas, os módulos. Um módulo tem um nome, uma declaração de interface e um corpo que é executável.

Module <nome> :
<declaração de interface>
<corpo>
end module

Um módulo pode utilizar submódulos que são módulos instanciados pela construção "**run**"; não pode haver recursividade sobre a instanciação.

C.2 Declaração de interface

A declaração de interface define os objetos que um módulo importa ou exporta. Ela contém *objetos de dados* declarados de forma abstrata em Esterel e implementados externamente e *objetos de interface reativa*.

C.2.1 Dados

As declarações de dados declaram os objetos que manipulam dados:

- **Tipos e operadores**

Os cinco tipos primitivos de Esterel são: **boolean**, **integer**, **float**, **double** e **string**. As operações são as usuais: igualdade "=" e diferença "< >" para todos os tipos, "**and**", "**or**" e "**not**" para o tipo **boolean** e "**+, -, *, /, <, < =, >, >=**" para tipos **integer**, **float**, **double**. O usuário pode definir seus próprios tipos declarando seus nomes; um tipo do

usuário é um objeto abstrato cuja definição será dada somente na linguagem hospedeira.

- **Constantes**

É possível declarar constantes de qualquer tipo. Quando o tipo é pré-definido, são declarados em Esterel nome, tipo e valor; quando o tipo é definido pelo usuário são apenas declarados nome e tipo, sendo que o valor é definido na linguagem hospedeira.

- **Funções**

A declaração de função contém a lista de tipos dos objetos que vai usar e o tipo do objeto de retorno: "**function <nome> (<lista de tipo de argumentos>) : <tipo do retorno>;**".

- **Procedimentos**

A declaração de um procedimento tem duas listas (opcionais) de argumentos de tipos arbitrários: lista de argumentos passados por referência e modificáveis pela chamada ("**call**"), lista de argumentos passados por valores e não modificáveis. A declaração se apresenta na forma: "**procedure <nome> (<lista de tipo de argumento-referência>) (<lista de tipo de argumento-valor>);**".

Procedimentos e funções são definidos na linguagem hospedeira e não apresentam efeitos colaterais.

- **Tarefas**

As tarefas são entidades de cálculo externo mas que não podem ser consideradas instantâneas. Elas são sintaticamente similares aos procedimentos e são executadas pela construção "**exec**" acoplada com o sinal "**return**" (ver a seguir)

C.2.2 Sinais e Sensores

Os sinais e sensores constituem a interface reativa do módulo. Os sinais são difundidos instantaneamente em todo o programa. O valor difundido de um sinal com valor ou de um sensor é único a cada instante.

- **Declaração de sinais de interface**

Os sinais de interface são de entrada "**input**", de saída "**output**", de entrada-saída "**inputoutput**" e de terminação de tarefas externas "**return**". Os sinais se dividem em sinais puros (por exemplo "**input <nome-sinal>;**") que tem apenas um estado de presença ("*presente*" ou "*ausente*") e sinais com valor que transportam também um

valor de tipo arbitrário (por exemplo "**output <nome-sinal> := <valor inicial> : <tipo-sinal>;**").

Um sinal com valor não pode ser emitido pelo programa duas vezes no mesmo instante e nem no mesmo instante que ele esta sendo recebido do ambiente. Ele é chamado de sinal com valor único. Para se livrar desta restrição, utiliza-se a palavra chave "**combine**" na declaração do sinal com valor; os sinais são chamado de sinais com valor combinados. Operadores (and, or, +, *) e outras funções declarados pelo usuário podem ser usados na combinação de sinais.

Existe apenas um sinal puro pré-definido "**tick**" que representa o relógio de ativação do programa reativo mas não precisa declara-lo. Seu estado tem o valor "*presente*" a cada instante.

- **Sensores**

Os sensores são sinais de entrada com valor mas sem a presença da informação de estado: "**sensor <nome-sensor> : <tipo-sensor>;**". O valor do sensor é acessado pelo programa através da construção "**?**", quando necessário

- **Relações de entrada**

A construção "**relation ...**" permite representar relações de entrada que indicam condições booleanas entre os sinais "**input**" e "**return**"; estas condições são supostamente garantidas pelo ambiente. As relações são de incompatibilidade ou exclusão entre os sinais ("#") ou de sincronização entre sinais ("=>").

- **Declaração de sinal local**

Sinais podem ainda ser declarados localmente pela declaração "**signal <lista de sinais> in p end signal**" sem aparecer na interface do módulo. A declaração dos sinais locais é feita da mesma forma que a dos sinais de interface e o escopo dos sinais locais é o corpo da construção **p**. Numa malha, um sinal local pode ser executado várias vezes no mesmo instante, criando a cada execução uma nova copia.

C.2.3 Variáveis

As variáveis são objetos aos quais valores podem ser atribuídos. A declaração das variáveis com seu nome, valor inicial e tipo é feita numa construção de declaração de variável local "**var <lista de variáveis> in p end var**". O escopo da declaração de variável e o corpo da construção **p**. A modificação da variável pode ser o resultado de atribuições, chamadas de procedimentos e execuções de tarefas externas ("**exec**"). Contrariamente ao sinal, uma variável pode tomar várias valores sucessivos no mesmo instante.

C.2.4 Expressões

As expressões possíveis em Esterel são:

- **Expressões de dados**

Elas combinam constantes, variáveis, sensores e sinais com valores usando operadores e chamadas de função. A sua avaliação é instantânea e envolve checagem de tipos. "**?S**" indica o valor corrente do sensor ou do sinal com valor **S**.

- **Expressões de sinais**

Elas são expressões booleanas ("**not**", "**and**", e "**or**") aplicadas sobre os estado de sinais (ou do sinal "tick"). Elas são utilizados em testes de presença ou em expressões de atraso.

- **Expressões de atraso**

Elas são utilizadas em construções temporais tais como "**await**" ou "**abort**" para expressar atrasos que começam quando inicia a construção temporal que as contém. Existem três tipos possíveis:

- atrasos padrões definidos por expressões de sinais e que nunca esgotam instantaneamente;
- atrasos imediatos definidos por "**immediate [<expressão de sinais>]**" e que esgotam instantaneamente;
- atrasos de contagem definidos por uma expressão de contagem inteira seguida por uma expressão de sinais: ("**<expressão-contagem> [<expressão-sinais>]**").

C.3 Construções do corpo

Todas as construções utilizáveis no corpo do módulo são descritas a seguir, a exceção da declaração do sinal local já descrito anteriormente.

- **Construções básicas de controle**

As construções básicas de controle são "**nothing**" que termina instantaneamente, "**pause**" que para e termina no próximo instante , "**halt**" que para para sempre sem nunca terminar.

- ## Atribuição

A atribuição "**X := e**" com a variável **X** e a expressão de dados **e** do mesmo tipo é instantânea.

- ## Chamada de procedimento

A chamada de procedimento tem a forma "**call P (X, Y) (e_1, e_2)**" onde **X**, **Y** são variáveis e $\mathbf{e_i}$ são expressões. A chamada é instantânea.

- ## Emissão de sinal

A emissão instantânea de sinal é realizada por "**emit S**" ou "**emit S(e)**" respectivamente no caso de um sinal puro ou de um sinal com valor resultante da avaliação da expressão **e**. Para um sinal único, somente um "**emit**" pode ser executado neste instante; para um sinal combinado, o valor emitido é combinado com os que são emitidos por outros "**emit**" neste instante, usando a função de combinação.

A emissão contínua de um sinal é realizada pela construção "**sustain S**" ou "**sustain S(e)**" que fica ativa para sempre e emite **S** ou **S(e**) a cada instante.

- ## Seqüência

O operador de seqüência "**;**" permite que a construção **q** de "**p; q**" inicia imediatamente após a construção **p** ter terminada, a menos que **p** contenha algum "**exit**" de um "**trap**".

- ## Malha

A construção "**loop p end loop**" representa a malha simples infinita. O corpo **p** é sempre re-iniciado imediatamente após seu término. Se construções "**trap**" fazem parte do corpo **p**, os "**exit**" são propagados instantaneamente e a malha para. Não é permitido que o corpo **p** de uma malha possa terminar instantaneamente quando iniciado.

A malha repetitiva executa seu corpo **p** um número finito de vezes. Não é permitido que o corpo **p** termina instantaneamente. A construção "**repeat e times p end repeat**" na qual **e** é do tipo "**integer**" e é avaliada somente uma vez no instante inicial. Se o resultado da avaliação for zero ou negativo, o corpo não será executado. A construção "**repeat**" é considerada como instantânea e não poderá ser colocada numa malha "**loop**" se não for precedido ou seguido por um atraso. Para garantir, neste caso, que o corpo será executado pelo menos uma vez, utiliza-se a construção "**positive repeat ...**" que permite realizar o teste para repetição somente após a primeira execução do corpo.

- **Testes**

O teste de presença binário tem a seguinte forma geral "**present e then p else q end present**"; uma das ramificações "**then**" ou "**else**" pode ser omitida e neste caso a omitida tem o significado de "**nothing**". O teste de presença múltiplo utiliza a construção "**case**" dentro do "**present**" da forma seguinte:

present
case e_1 **do** p_1
case e_2 **do** p_2
case e_3 **do** p_3
else q
end present

O teste condicional utiliza a construção "**if**" para testar expressões de dados booleanas. O teste condicional binário tem a seguinte forma "**if <condição> then p else q end if**". O teste condicional múltiplo não usa o "**case**" reservado para os teste de presença de sinal mas pode ser realizado em seqüência usando a palavra chave "**elseif**" da forma seguinte:

if **<condição 1> then** p_1
elseif <condição 2> then p_2
elseif <condição 3> then p_3
else q
end if

- **Atraso**

A construção temporal mais simples "**await S**" significa a espera por um atraso. Quando a construção é iniciada, ela entra em pausa até o atraso ser esgotado, instante no qual ela termina.

A construção "**await immediate S**" termina instantaneamente se o sinal for presente ou a expressão de sinal for verdadeira no instante inicial.

Pode se utilizar ainda a construção "**case**" dentro do "**await**"; o primeiro atraso que se esgota fixa o comportamento seguinte; se dois deles se esgotam simultaneamente, a prioridade é com o primeiro da lista; a cláusula "**else**" não é permitida. A contagem do número de sinais ou uma expressão de sinal pode também ser utilizada para expressar o atraso.

A construção "**await**" completa utiliza uma cláusula "**do**" para iniciar outra construção quando o atraso esgota: "**await <expressão-atraso> do p end await**".

- **Preempção**

A construção "**abort p when <expressão-atraso> do q**" realiza uma preempção

forte do corpo **p** mas não entrega o controle a este no instante de preempção.

A construção "**weak abort p when <expressão-atraso> do q**" realiza uma preempção fraca na qual o corpo **p** recebe o controle para um último instante no momento de preempção.

A claúsula "**do**" permite executar uma construção **q** se o atraso esgotar, imediatamente no caso de "**abort**" e após o último instante no caso de "**weak abort**".

A introdução da palavra chave "**immediate**" nestas construções na forma "**... when immediate <expressão-atraso> ...",** permite ao atraso de se esgotar imediatamente no instante de inicio se a expressão de atraso for verificada neste; o corpo **p** da construção não se executa no caso "**abort**" e se executa para um último instante no caso "**weak abort**".

Uma lista de "**case**" pode também ser introduzida nestas construções de preempção. Construções "**abort**" aninhadas são também possíveis e estabelecem prioridades; por exemplo J é prioritário sobre I se os dois ocorrem simultaneamente e q não será iniciado neste caso:

```
abort
    abort p
    when I do q
    end abort
when J
```

A construção "**suspend p when <expressão-sinal>**" tem um efeito de suspensão (do tipo ^Z do Unix). O corpo **p** é imediatamente iniciado quando a construção "**suspend**" inicia,. A cada instante, se a expressão de sinal for "*true*", o corpo **p** se suspende no seu estado atual e a construção "**suspend**" faz uma pausa neste instante; se a expressão de sinal for "*false*", o corpo **p** é executado neste instante. Para realizar o teste da expressão de sinal no primeiro instante é necessário utilizar "**suspend p when immediate <expressão-sinal>**".

- ## Malha temporal

As malhas temporais são infinitas e o único meio de termina-las é através de uma exceção ("**exit**" de um "**trap**"). Existem duas formas de declarar malhas temporais:

- "**loop p each d**" onde d é um atraso não imediato. O corpo **p** é inicializado ao instante inicial, e re-inicializado a cada vez que o atraso **d** esgota; se **p** termina antes do esgotamento de **d**, o re-inicio de **p** deverá esperar até o atraso **d** se esgotar. Esta construção é uma abreviação de:

```
loop
        abort
                p; halt
        when d
end loop
```

- "**every d do p end**" que difere do anterior pela espera inicial de **d** antes de iniciar o corpo **p**. Esta construção abrevia:

```
await d;
loop
        p
each d
```

Esta construção pode ser imediata "**every immediate d do p end**" e neste caso, no instante inicial, **p** inicia imediatamente se o atraso **d** esgota imediatamente.

- **Exceção e tratamento**

O mecanismo de exceção é implementado pela construção:

```
trap T in
        p
handle T do
        q
end trap
```

O corpo **p** inicia imediatamente quando a construção "**trap**" inicia. A sua execução continuará até o término de **p** ou a saída através da execução da construção "**exit T**" que leva o "**trap**" a terminar imediatamente, abortando **p** por preempção fraca. O tratamento da exceção (quando houver) é realizado pela construção "**handle**", que permite o inicio imediato de **q** após corpo **p** ter sido abortado pelo "**exit**" do "**trap**":

Quando se tem construções "**trap**" aninhadas, a mais externa tem a maior prioridade. Várias exceções podem ser declaradas numa única construção "**trap**"; todas essas exceções são concorrentes e tem o mesmo nível de prioridade; no caso de várias exceções ocorrer simultaneamente, seus tratadores de exceção serão executados em paralelo.

As construções "**trap**" podem ter valores como os sinais. Inicialização de valor e "**trap**" combinados são permitidos como no caso dos sinais. A passagem de um valor ao tratador de exceção é possível e é o resultado da expressão "**??S**" que se encontra somente neste tratador.

- **Paralelismo**

O operador de paralelismo "**ll**" (que pode ter qualquer aridade) coloca as construções que ele separa em paralelismo síncrono. Os sinais emitido por um dos ramos ou por outra parte do programa são difundidos instantaneamente a todas os ramos. Somente variáveis para leitura podem ser compartilhadas em todos os ramos da construção paralela.

Uma construção paralela, quando inicia, dispara instantaneamente uma "*thread*" por ramo. Ela terminará quando todos os ramos terão terminado, esperando eventualmente até o último terminar. Se o "**exit**" de uma exceção "**trap**" é ativada num ramo, ele é propagada em todos os outros ramos, levando a uma preempção fraca ("**weak abort**") de todos os ramos no mesmo tempo.

C.4 Instanciação de módulo

A instanciação de um módulo dentro de outro módulo é possível a partir da construção executável "**run <nome-módulo>**".A instanciação recursiva de submódulos é proibida.

Como todos os dados são globais em Esterel, as declarações de dados de um submódulo instanciado são exportados no módulo pai e vice-versa. As declarações de interface de sinais e de relações devem ser descartadas; as interface de sinais de um submódulo instanciado deve existir no módulo pai com o mesmo tipo.

A renomeação de objeto de interface é possível em tempo de instanciação de módulo conforme definido na construção "**run <nome-módulo> [X / Y; ...]**" que permite a renomeação de Y por X, desde que os tipos e operadores de tipos coincidam ("match"). O objeto renomeando X pode ser uma constante ou um operador ou um identificador; o objeto renomeado Y deve ser um identificador pertencente a interface de sinal ou a dados de módulo instanciado.

C.5 A execução de tarefa externa

O controle da execução de tarefas externas que levam tempo usa o mecanismo "**exec**". Essas tarefas se comportam como procedimentos a serem executados assincronamente. O programa Esterel se interessa apenas pelo inicio e pelo fim delas ou pela suspensão ou preempção das mesmas por outras construções Esterel.

A execução de uma tarefa é realizada por:

> "**exec Task (<parâmetros-referência>) (<parâmetros-valores>) return R**"

na qual **R** é o sinal de retorno. Quando deseja-se, simultaneamente, controlar várias tarefas, utiliza-se:

exec
case T_1 (...) (...) return R_1 do p_1
...
case T_n (...) (...) return R_n do p_n
end exec

Bibliografia

[ABR91] N. Audsley, A. Burns, M. F. Richardson, A. J. Wellings. *Hard Real-Time Scheduling: The Deadlin- Monotonic Approach.* Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software, pp. 133-137, May 1991.

[AnP93] C. André, M-A. Peraldi. *Synchronous Approach to Industrial Process Control*, Technical Report N^o 93-10, Laboratoire I3S, Université de Nice, March 1993.

[ARS91] K. Arvind, K. Ramamritham, J. Stankovic, *A Local Area Network Architecture for Communication in Distributed Real-Time Systems*. The Journal of Real-Time Systems, 3, pp. 115 – 147, 1991.

[AuB90] N. Audsley, A. Burns, *Real-Time System Scheduling*, on First Year Report Task B of the Esprit BRA Project 3092: Predictably Dependable Computing Systems, Chapter 2, vol2. of 3, May 1990.

[Aud93] N. Audsley, *Flexible Scheduling of Hard Real-Time Systems.* PhD Thesis, Department of Computer Science, University of York, UK, 1993.

[Bak91] T. P. Baker. *Stack-Based Scheduling of Realtime Processes.* The Journal of Real-Time Systems, Vol. 3, pp. 67-90, 1991.

[BCH95] E. Byler, W. Chun, W. Hoff, D. Layne. *Autonomous Hazardous Waste Drum Inspection Vehicle*, IEEE Robotics & Automation Magazine, March 1995.

[BCJ97] F. Balarin, M.Chiodo, A. Jureska, H. Hsieh, A. L. Lavagno, C. Passerone, A. Sangiovanni-Vincentelli, E. Sentovich, K. Suzuki e B. Tabbara. *Hardware-Software Co-Design of Embedded Systems: The Polis Appoach*, Livro a ser publicado por Kluwer Academic Press, 1997.

[BCL99] A. Benveniste, B. Caillaud, P. Le Guernic. *Compositionality in dataflow synchronous languages: specification & distributed code generation*, in Journal Information and Computation, 1999.

[BCN95] G.Bucci, M. Campanai, P. Nesi. *Tools for Specifying Real-Time Systems*, Journal of Real-Time Systems, pp. 173-198, 1995.

[BeB91] A Benveniste, G. Berry, *The Synchronous Approach to Reactive and Real-Time Systems*, Proceedings of the IEEE, vol 79 (9), pp1270-1282, sept. 1991.

[BeB97] G. Bernat, A. Burns, *Combining (n m)-Hard Deadlines and Dual Priority Scheduling*, In Proceedings. of the 18th IEEE Real-Time Systems Symp. , December 1997.

[BeC99] A. Benveniste, P. Caspi. *Distributing synchronous programs on a loosely synchronous, distributed architecture*, Rapport de Recherche Irisa, N°1289, Décembre 1999.

[BeG92] G. Berry, G. Gonthier, *The Esterel Synchronous Programming Language: Design, Semantics, Implementation*, Science of Computer Programming vol. 19, n°2, pp 87-152, 1992.

[Ber89] G. Berry, *Real-Time Programming: Special Purpose or General Purpose Languages*, In Information Processing 89, pp11-17, Ed. Elsevier Science Publishers, 1989.

[Ber92] G. Berry. *Esterel on Hardware*, on Philosophical Transactions Royal Society of London A, vol 339, pp. 87-104, 1992.

[Ber98] G. Berry, *The Foundations of Esterel*, In Proof, Language and Interaction: Essays in Honour of Robin Milner, G.Plotkin, C. Stirling and M.Tofte (editors), Ed. MIT Press, 1998.

[Ber99] G. Berry, *The Esterel v5 Language Primer*, Esterel Reference Manual. 1999.

[BNT93] A. Burns, M. Nicholson, K.W.Tindell, N. Zhang. *Allocation and Scheduling Hard Real-Time Tasks on a point-to-point Distributed System.* Proceedings of the Workshop on Parallel and Distributed Real-Time Systems, pp. 11-20, Dana Point, CA, 1993.

[BoS91] F. Boussinot, R. de Simone, *The Esterel Language: Another Look at Real-Time Programming*, Proceedings of the IEEE, vol. 79, pp 1293-1304, sept. 1991.

[BoS96] F. Boussinot, R. de Simone, *The SL Synchronous Language*, IEEE Transactions Software Engineering, vol. 22 (4), pp256-266, april 1996.

[Bud94] R. Budde. *Esterel Applied to the Case Study Production Cell*, in FZI Publication (chap. 4) intitled "*Case Study Production Cell: A Comparative Study in Formal Software Development*", Forschungszentrum Informatik Karlsruhe, 1994.

[But97] G.C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, Ed. Kluwer Academics Publishers, 1997.

[BuW97] A. Burns, A. Wellings, *Real-Time Systems and Programming Languages*, Second edition. Addison-Wesley,1997.

[CaB97] M. Caccamo, G. Butazzo, *Exploiting Skips in Periodic Tasks for Enhancing Aperiodic Responsiveness*, In Proc. of the 18th IEEE RTSS, Dec. 1997.

[CaK88] D. Callahan, K. Kennedy. *Compiling Programs for Distributed Memory Multiprocessors*, Journal Supercomputing, Vol. 2, pp. 151-169, 1988.

[CER92] E. Coste-Manière, B. Espiau, E. Rutten. *Task-level programming combining object-oriented design and synchronous approach*, in IEEE International Conference on Robotics and Automation, pp. 2751-2756, Nice, May 1992.

[CGP99] P. Caspi, A. Girault, D. Pilaud. *Automatic Distribution of Reactive Systems for Asynchronous Networks of Processors*, IEEE Transactions on Software Engineering, vol. 25, N^o 3, pp. 416-427, May/June 1999.

[CLL90] J.-Y. Chung, J. W. S. Liu, K. -J. Lin, *Scheduling Periodic Jobs that Allow Imprecise Results*, IEEE Transactions on Computer, 39(9), pp.1156-1174, 1990.

[Coo96] J. E. Cooling. *Languages for the Programming of Real-Time Embedded Systems - A Survey and Comparison*, Microprocessors and Microsystems, 20, pp. 67-77, 1996.

[Cos 89] E. Coste-Manière. *Utilisation dÉsterel dans un contexte asynchrone: una application robotique*, Rapport de Recherche INRIA N^o 1139, Dec 1989.

[CSR88] S. Cheng, J. A. Stankovic, K. Ramamrithan, *Scheduling Algirithms for Hard Real-Time Systems: A Brief Survey*. In Hard Real-Time Systems: Tutorial, Ed. J. A. Stankovic and K. Ramamrithan, pp. 150-173, IEEE Computer Society Press, 1988.

[DEL91] DELTA-4, *Real-Time Concepts*, on Delta-4 Architecture Guide, Cap.5, pp.102-124, 1991.

[DTB93] R. I. Davis, K. W. Tindell, A. Burns. *Scheduling Slack Time in Fixed Priority Pre-emptive Systems*. Proceedings of the IEEE Real-Time Systems Symposium, pp. 222-231, 1993.

[ENS99] J. Euler, M. do C. Noronha, D. M. da Silva, *Estudo de Caso: Desempenho do Sistema Operacional Linux para Aplicações Multimídia em Tempo Real*, Anais do II Workshop de Tempo Real, Salvador-BA, 25-28 de maio de 1999.

[FeC97] C. Fetzer, F. Cristian, *Integrating External and Internal Clock Synchronization*, The Real-Time Systems Journal, pp.123-171, dez 1997.

[Fid98] C. J. Fidge. *Real-Time Schedulability Tests for Preemptive Multitasking*. Journal of Real-Time Systems Vol 14. pages 61-93, 1998.

[GaJ79] M. R. Garcey, D.S.Johnson. *Computer and Intractability: a Guide to the Theory of the NP-Completeness*. W.H.Freeman and Company,1979.

[Gal95] B. O. Gallmeister. *POSIX.4 Programming for the Real World*. O'Reilly & Associates, ISBN 1-56592-074-0, 1995.

[GeR91] N. Gehani, K. Ramamritham. *Real-Time Concurrent C: a Language for Programming Dynamic Real-Time Systems*, Journal of Real-Time Systems, vol. 3, 1991.

[GLM94] T. Gautier, P. LeGuernic, O. Maffeis. *For a New Real-Time Methodology*, Rapport de Recherche Inria, No2364, Octobre 1994.

[GNM97] M. Gergeleit, E. Nett, M. Mock, *Supporting Adaptive Real-Time Behavior in CORBA*, Proceedings. of the First IEEE Workshop on Middleware for Distributed Real-Time Systems and Services. San Francisco, CA, Dec. 1997.

[Gus94] J. Gustafsson, *Calculation of Execution Times in Object-Oriented Real-Time Software – A Study Focused on RealTimeTalk*, PhD Thesis, Royal Institute of Technology, Suécia, 1994.

[Har94] M. G. Harnon, et al. *A Retargetable Technique for Predicting Execution Time of Code Segments*, The Journal of Real-Time Systems, Vol. 7, pp 157-182, 1994.

[HaR95] M. Hamdaoui, P. Ramanathan, *A Dynamic Priority Assignment Technique for Streams com deadline (m,k)-firms*, In IEEE Transactions on Computer, April 1995.

[Har87] D. Harel. *Statecharts: a Visual Approach to Complex Systems*, Science of Computer Programming, vol 8, pp231-274, 1987.

[HCR91] N. Halbwachs, P.Caspi, D. Pilaud. *The Synchronous Dataflow Programming Language Lustre*, Another Look at Real Time Programming, Proc. of the IEEE, vol. 79, sept. 1991.

[HeM96] C. Heitmeyer, D. Mandrioli (eds). *Formal Methods for Real-Time Computing*, Ed. Wiley – Trends in Software (5).

[HSP98] R. Hill, B. Srinivasan, S. Pather, D. Niehaus. *Temporal Resolution and Real-Time Extensions to Linux*. Technical Report ITTC-FY98-TR-11510-03, Information and Telecommunication Technology Center, Electrical Engineering and Computer Science Department, University of Kansas, 1998.

[JeS93] K. Jeffay, D. L. Stone. *Accounting for Interrupt Handling Costs in Dynamic Priority Task Systems.*Proceedings of the IEEE Real-Time Systems Symposium, pp. 212-221, December 1993.

[JLT85] E.Jensen, C. Locke, H. Tokuda. *A Time-Driven Scheduling Model for Real-Time Operating Systems*, Proceedings of the 6th IEEE RTSS, pp.112-122, Dec. 1985.

[JoP86] M. Joseph, P. Pandya. *Finding Response Times in a Real-Time System*. BCS Computer Journal Vol 29, Nº 5, pp. 390-395, 1989.

[Jos91] M. Joseph. *Problems, Promises and Performance: Some questions for real-time system specification*, on Proceedings of Rex Workshop on Real-Time: Theory in Practice, Lecture Notes in Computer Science Nº 600, June 1991, pp.315-324, Ed. Springer-Verlag.

[Kic97] G. Kiczales, et al. *Aspect-Oriented Programming*, Proceedings of the ECOOP'97, Spring-Verlag LNCS, No 1241, Finlândia, Junho de 1997.

[Kop92a] H. Kopetz, *Sparse Time versus Dense Time in Distributed Real-Time Systems*, on Proceedings of 12th International Conference on Distributed Computing Systems ICDCS'12, pp.460-467, June 1992, Yokohama (Japan).

[Kop92b] H. Kopetz. *Real-Time and Real-Time Systems*, on Proceedings of Advanced Course on Distributed Systems, July 1992, Estoril (Portugal).

[Kop92c] H. Kopetz, *Scheduling.* on Proceedings of Advanced Course on Distributed Systems, July 1992, Estoril (Portugal).

[Kop97] H. Kopetz. *Real-Time Systems. Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers,1997.

[KoS95] G. Koren, D. Shasha, *Skip-Over: Algorithms e Complexity for Overloaded Systems that Allow Skips*, In Proceedings of the 16th IEEE RTSS, Pisa, Italy, December 1995.

[KSt86] E. Kligerman, A. Stoyenko. *Real-Time Euclid: A Language for Reliable Real-Time Systems.* IEEE Transactions on Software Engineering, 12(9), September 1986.

[KuM97] T. Kuo, A. K. Mok, *Incremental Reconfiguration e Load Adjustment in Adaptive Real-Time Systems*, IEEE Trans. on Computers, Vol. 46, No. 12, Dec. 1997.

[LeL90] G. Le Lann. *Critical Issues for the Development of Distributed Real-Time Computing Systems*, Rapport de Recherche INRIA Nº 1274, Août 1990.

[LeR92] J. P. Lehoczky, S. Ramos-Thuel. *An Optimal Algorithm for Scheduling Soft-Aperiodic Tasks in Fixed-Priority Preemptive Systems*. Proceedings of the IEEE Real-Time Systems Symposium, pp. 110-123, 1992.

[LeW82] J. Y. T. Leung, J. Whitehead. *On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks*. Performance Evaluation, 2 (4), pp. 237-250, december 1982.

[Li95] G. Li, *An Overview of Real-Time ANSAware* 1.0, Document APM. 1285.01, March 1995.

[LiL73] C. L. Liu, J.W.Layland. *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. Journal of the ACM, Vol. 20, No. 1, pp. 46-61, january 1973.

[LSL94] J. W. S. Liu, W. Shih, K.-J. Lin, R. Bettati, J.-Y. Chung. *Imprecise Computing*. Proceedings of the IEEE, Vol. 82, N^{o} 1, pp. 83-94, January 1994.

[LLG91] P. Le Guernic, M. Le Borgne, T. Gauthier, C. Le Maire, *Programming Real Time Applications with Signal*, Another Look at Real Time Programming, Proc. of the IEEE, vol. 79, sept. 1991.

[LSD89] J. P. Lehoczky, L. Sha, Y. Ding. *The Rate Monotonic Scheduling Algorithm: Exact Characterization and Avarage-Case Behavior*. Proceedings of the IEEE Real-Time Systems Symposium, pp.166-171, Los Alamitos, CA, December 1989.

[LSS87] J. P. Lehoczky, L. Sha, J.K. Strosnider. *Enhanced Aperiodic Responsiveness in Hard Real-Time Environments*. Proceedings of IEEE Real-Time Systems Symposium, San Jose, CA, pp. 261-270, 1987.

[Mae87] P. Maes, *Concepts and Experiments in Computational Reflection*, Proceedings of OOPSLA'87, pp. 147-155, 1987.

[Maf93] O. Maffeis. *Ordonnancements de Graphes de Flots Synchrones; Application a la Mise en Oeuvre de Signal*, Tese de Doutorado, Universidade de Rennes I, France, Jan.1993.

[MFO99] C. Montez, J. Fraga, R. S. Oliveira, J-M. Farines, *An Adaptive Scheduling Approach in Real-Time CORBA*, 2nd IEEE International Symposium on Object-oriented Real-time Distributed Computing - ISORC'99, Saint-Malo, France, May 1999.

[Mil80] R. Milner, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science, vol 92, 1980, Ed. Springer-Verlag.

[Mot92] L. Motus, *Time Concepts in Real-Time Software*, on Proceedings of International Workshop on Real-Time Programming WRTP'92, June 1992, Bruges (Belgium).

[OlF97] R. S. Oliveira, J. S. Fraga, *Escalonamento de Tarefas com Relações Arbitrárias de Precedência em Sistemas Tempo Real Distribuídos*, 16° Simpósio Brasileiro de Redes de Computadores, SBC, Rio de Janeiro-RJ, 25-28 de maio de 1998.

[OMG98] OMG, *Realtime CORBA - Joint Revised Submission*, Object Management Group (OMG), Document orbos/98-10-05, October 1998.

[Ort99] S. Ortiz Jr, *The Battle Over Real-Time Java*, IEEE Computer, Vol. 32, No. 6, pp. 13-15, june 1999.

[Pin95] M. Pinedo, *Scheduling: Theory, Algorithms and Systems*. Prentice-Hall, 1995.

[POS97] A. Pascoal, P. Oliveira, C. Silvestre, A. Bjerrum, A. Ishoy, J.-P. Pignon, G. Ayela, C. Petzelt. *MARIUS: An Autonomous Underwater Vehicle for Coastal Oceanography*, IEEE Robotics & Automation Magazine, december 1997.

[Raj91] R. Rajkumar. *Synnchronization in Real-Time Systems: A Priority Inheritance Approach.* Kluwer Academic Publishers,1991.

[RaS94] K. Ramamrithan, J. A. Stankovic. *Scheduling Algorithms and Operating Systems Support for Real-Time Systems*. Proceedings of the IEEE, Vol. 82, N° 1, pp. 55-67, January 1994.

[Ray91] M. Raynal, *La Communication et le Temps dans les Réseaux et les Systèmes Répartis*, Cap.8 et 9, 1991, Ed. Eyrolles.

[RNS93] U. Rembold, B. O. Nnaji, A. Storr, *Computer Integrated Manufacturing and Engineering*, Addison-Wesley Publishing Company, ISBN 0-201-56541-2, 1993.

[Rus93] J. Rushby. *Formal Methods and the Certification of Critical Systems*, Technical Report CSL-93-7, disponivel em http://www.csl.sri.com, 1993.

[SBS93] S. Ramesh, G. Berry, R.K. Shyamasundar. *Communicating Reactive Processes*, in Proceedings of 20th ACM Conference on Principles of Programming Languages, 1993.

[SeR98] B. Selic, J. Rumbaugh. *Using UML for Modeling Complex Real-Time Systems*, artigo disponível em http://www.rational.com, 1998.

[SGW94] B. Selic, G. Gullekson, P. Ward. *Real-Time Object-Oriented Modeling*, Ed. Wiley – Wiley Professional Computing.

[SiG98] A. Silberschatz, P. B. Galvi, *Operating System Concepts*, Addison-Wesley, 5th edition, ISBN 0-201-59113-8, 1998.

[SpB96] M. Spuri, G.C. Buttazzo. *Scheduling Aperiodic Tasks in Dynamic Priority Systems.* Journal of Real-Time Systems Vol 10 N^o 2, 1996.

[SPG97] S. Shenker, C. Partridge, and R. Guerin, *Specification of Guaranteed Quality of Service*. RFC 2212, IETF Specification, 1997.

[SPH98] B. Srinivasan, S. Pather, R. Hill, F. Ansari, D. Niehaus. A Firm Real-Time System Implementation Using Commercial Off-The-Shelf Hardware and Free Software. Proc. of the Real-Time Technology and Applications Symposium, june 1998.

[Spu96] M. Spuri. *Analysis of Deadline Scheduled Real-Time Systems*, Technical Report, INRIA, França N^o 2776, Janeiro 1996.

[SRL90] L. Sha, R. Rajkumar, J. P. Lehoczky. *Priority Inheritance Protocols: An approach to Real-Time Synchronization*. IEEE Transactions on Computers, Vol. 39, No. 9, pp. 1175-1185, september 1990.

[SSL89] B. Sprunt, L. Sha, J. Lehoczky. *Aperiodic Task Scheduling for Hard-Real-Time Systems*. The Journal of Real-Time Systems, Vol. 1, pp. 27-60, 1989.

[Sta88] J. A. Stankovic, *Misconceptions about real-time computing*, IEEE Computer, vol 21 (10), October 1988.

[Sta96] J. Stankovic et al. *Strategic Directions in Real Time and Embedded Systems*, ACM Computing Surveys, Vol. 28, No 4, pp. 751-763, December 1996.

[STB96] E. Sentovich, H. Toma, G. Berry. *Latch Optimization in Circuits Generated from High-level Descriptions*, in Proceedings of International Conference on Computer-Aided Design (ICCAD), 1996.

[StR88] J. A. Stankovic, K. Ramamrithan, (Editors) *Tutorial on Hard Real-Time Systems*, 1988, IEEE Computer Society Press.

[StR90] J. A. Stankovic, K. Ramamrithan, *What is predictability for Real-Time Systems*?, The Journal of Real Time Systems, vol.2, pp.247-254, 1990, Ed. Kluwer Academic Publications.

[TaT92] K. Takashio, M. Tokoro, *DROL: An Object-Oriented Programming Language for Distributed Real-Time Systems*, Proceedings of OOPSLA'92, 1992.

[TaT93] K. Takashio, M. Tokoro, *Time Polymorphic Invocation: A Real-Time Communication Model for Distributed Systems*, In Proceedings of the IEEE WPDRTS'93, 1993.

[TaW97] A. S. Tanenbaum, A. S. Woodhull, *Sistemas Operacionais Projeto e Implementação*, Bookman, segunda edição, ISBN 85-7307-530-9, 1997.

[TBU98] M. Timmeman, B. V. Beneden, L. Uhres. *RTOS Evaluation Kick Off* Real-Time Magazine, 1998-Q33, http://www.realtime-info.be, (atualmente Dedicated Systems Magazine), 1998.

[TBW94] K. W. Tindell, A. Burns, A.J. Wellings. *An Extendible Approach for Analyzing Fixed Priority. Hard Real-Time Tasks*. Journal of Real-Time Systems 6(2). pages 133-151, 1994.

[TiC94] K. W. Tindell, J. Clark. *Holistic Schedulability Analysis for Distributed Hard Real-Time Systems.* Microprocessors and Microprogramming Vol. 40, 1994.

[Vah96] U. Vahalia, *Unix Internals - The New Frontiers*, Prentice-Hall, ISBN 0-13-101908-2, 1996.

[Vie99] J. E. Vieira. *LINUX-SMART: Melhoria de Desempenho para Aplicações Real-Time Soft em Ambiente Linux*. Dissertação de mestrado. Instituto de Matemática e Estatística, Universidade de São Paulo, Outubro de 1999.

[VRC97] P. Veríssimo, L. Rodrigues, A. Casimiro, *CesiumSpray: A Precise and Accurate Global Time Service for Large-scale Systems*, The Journal of Real-Time Systems, 12, pp.243-294, dez 1997.

[WaL99] Y.-C. Wang, K.-J. Lin. *Implementing a General Real-Time Scheduling Framework in the RED-Linux Real-Time Kernel*. Proc. of the Real-Time Systems Symposium, December 1999.

[Wur99] P. Wurmsdobler. *A Simple Control Application with Real-Time Linux*, Real-Time Linux Workshop, Vienna, 1999.

[XuP93] J. Xu, D. L. Parnas. *On Satisfying Timing Constraints in Hard Real-Time Systems*. IEEE Transaction on Software Engineering, Vol. 19, N^{o} 1, pp. 70-84, January 1993.

[YoB99] V. Yodaiken, M. Barabanov. *RT-Linux Version Two*, Real Time Linux Workshop, http://www.thinkingnerds.com/projects/rtl-ws/rtl-ws.html, Vienna, 1999.

[You82] S. J. Young, *Real-Time Languages Design and Development*, Ellis-Harwood Ed., 1982.

[ZhB94] S. Zhang, A. Burns. *Timed Proprieties of the Timed Token Protocol.* Technical Report YCS 243. University of York, UK, 1994.