RAFAEL VIDAL AROCA

Análise de Sistemas Operacionais de Tempo Real para Aplicações de Robótica e Automação

Dissertação apresentada ao Departamento de Engenharia Mecânica da Escola de Engenharia de São Carlos da Universidade de São Paulo para obtenção do título de Mestre.

Área de Concentração: Dinâmica das Máquinas e Sistemas

Orientador: Prof. Dr. Glauco Augusto de Paula Caurin

São Carlos

2008

Ficha catalográfica preparada pela Seção de Tratamento
da Informação do Serviço de Biblioteca – EESC/USP

Aroca, Rafael Vidal
A769a Análise de sistemas operacionais de tempo real para aplicações de robótica e automação / Rafael Vidal Aroca ; orientador Glauco Augusto de Paula Caurin. -- São Carlos, 2008.

Dissertação (Mestrado-Programa de Pós-Graduação em Engenharia Mecânica e Área de Concentração em Dinâmica das Máquinas e Sistemas) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2008.

1. Sistemas operacionais de tempo real. 2. Sistemas de tempo real. 3. Pior caso de resposta. 4. Latência de interrupções. I. Título.

FOLHA DE JULGAMENTO

Candidato: Bacharel **RAFAEL VIDAL AROCA**

Dissertação defendida e julgada em 31/10/2008 perante a Comissão Julgadora:

Aprovado

Prof. Associado **GLAUCO AUGUSTO DE PAULA CAURIN (Orientador)**
(Escola de Engenharia de São Carlos/USP)

APROVADO

Prof. ~~Dr.~~ **EVANDRO LUIS LINHARI RODRIGUES**
(Escola de Engenharia de São Carlos/USP)

APROVADO

Prof. Dr. **RÓMULO SILVA DE OLIVEIRA**
(Universidade Federal de Santa Catarina/UFSC)

Prof. Associado **JONAS DE CARVALHO**
Coordenador do Programa de Pós-Graduação em
Engenharia Mecânica

Prof. Associado **GERALDO ROBERTO MARTINS DA COSTA**
Presidente da Comissão da Pós-Graduação da EESC

DEDICATÓRIA

Ao pequeno e sereno Matheus, que sem saber, muito ajudou neste trabalho, e à Sílvia, minha querida esposa, pela compreensão e colaboração ao longo deste trabalho.

AGRADECIMENTOS

Aos meus pais José e Denisi, por todo apoio incondicional que deram ao longo de minha vida, que acabou tornando este trabalho possível, e pela compreensão devido a minha constante ausência durante a elaboração desta dissertação.

Mais uma vez, à minha esposa, Sílvia, pelas intermináveis correções e leituras deste texto.

Aos meus sogros Adonai e Maria Elisa, pelas discussões científicas de alto nível e pela compreensão durante a escrita desta dissertação.

Ao Rene Uriza da 3WT, que foi quem realmente tornou possível este trabalho junto ao Marcelo Akira que também agradeço. Também agradeço a diretoria da 3WT pelas dezenas de horas de trabalho cedidas para serem aplicadas neste projeto.

Ao Ricardo Ribeiro da 3WT, pelo auxílio com o empréstimo, durante mais de um ano, dos equipamentos utilizados nesta pesquisa.

Ao Glauco Caurin, orientador deste trabalho, que idealizou este projeto tendo em vista uma pesquisa acadêmica que também fosse útil para empresas, e conduziu o trabalho com observações valiosas.

Aos amigos e companheiros de laboratório sempre presentes Dalton M. Tavares e Leonardo M. Pedro, pelas valiosas revisões e sugestões deste texto, e pelo ótimo e produtivo trabalho realizado no *retrofit* do robô Scara.

Ao amigo Diego Fiori por estar sempre presente, tentando ajudar no que for possível, e pelas suas sábias sugestões.

Ao amigo Renan Prates, pelas diversas correções e sugestões pacientemente realizadas.

Aos colegas de laboratório Jorge Félix, pela ajuda com o desenvolvimento dos *device drivers* para o robô Scara, Jean Mimar, pelos relatórios técnicos da *Dedicated Systems* e Leandro Massaro, pelo auxílio com o *Windows CE.*

Ao Sergio D. Carvalho, pela colaboração de alto nível no projeto dos circuitos eletrônicos do Scara, e posterior montagem impecável dos novos equipamento no robô.

Ao Luis Augusto Rizzi da VisionBR, e ao pessoal da Dia System, pela compreensão e horas de trabalho cedidas para esta pesquisa.

Ao André Muezerie, da Microsoft, pelas diversas discussões produtivas sobre este trabalho.

Ao Paul Mertens da Wind River, pelo apoio técnico com o VxWorks e rápido auxílio junto ao departamento de *marketing* da Wind River para autorizar a divulgação dos resultados aqui presentes.

Aos autores e comunidades de *software* livre, pelos produtos gratuitos de altíssimo nível, como Linux, RTAI e LaTex, desenvolvidos com uma mentalidade futurista de informação e conhecimento livre para todos.

A todos amigos e colegas que eu possa ter esquecido de citar aqui, mas aos quais certamente sou grato pela paciência, sugestões e críticas.

RESUMO

AROCA, R. V. **Análise de sistemas operacionais de tempo real para aplicações de robótica e automação**. 2008. Dissertação (Mestrado) - Departamento de Engenharia Mecânica, Universidade de São Paulo, São Carlos, 2008.

Este trabalho apresenta um estudo sobre sistemas operacionais de tempo real (RTOS) utilizados na implementação da infraestrutura de controle digital para sistemas mecatrônicos, mas serve também como referência para outros sistemas que possuam restrições de tempo. Além de ter um caráter experimental, onde foram medidos e analisados dados como o pior tempo de resposta dos sistemas e a latência para tratamento de interrupções, este trabalho de pesquisa ainda contempla a implementação e uso de RTOS em situações práticas, bem como contempla a construção de uma plataforma geral de pesquisa que servirá de base para futuros trabalhos no laboratório de mecatrônica. Os sistemas analisados neste trabalho foram o VxWorks, QNX, Linux, RTAI, Windows XP, Windows CE e μC/OS-II. Outro produto gerado durante este trabalho foi um *Live* CD para auxiliar na implementação e ensino de conceitos e sistemas de tempo real.

Palavras-chave: sistemas operacionais de tempo real, sistemas de tempo real, pior caso de resposta, latência de interrupções

ABSTRACT

AROCA, R. V. **Analysis of real time operating systems for robotics and automation applications**. 2008. Thesis (Master) - Departamento de Engenharia Mecânica, Universidade de São Paulo, São Carlos, 2008.

This work presents a study about real time operating systems (RTOS) that are utilized as infrastructure to create digital control systems for mechatronics systems, and also for systems that have critical time constraints. Parameters like worst case response time and interrupt latency were measured for each operating system. This research project also covers the implementation and use of RTOS in practical situations. A general research platform for robotics and real time research was also developed and will be used for future works in the Mechatronics Laboratory. The tested systems were VxWorks, QNX, Linux, RTAI, Windows XP, Windows CE and μC/OS-II. Another product released during this work was a Live CD to aid the implementation and teaching of real time systems and concepts.

Keywords: real time operating systems, real time systems, worst case response time, interrupt latency

Sumário

Lista de Figuras

Lista de Tabelas

1 Introdução

Um grande crescimento na área de sistemas embarcados vem ocorrendo nos últimos anos graças a presença cada vez maior de microprocessadores e microcontroladores de baixo custo com alta capacidade computacional. Em termos percentuais, uma pesquisa realizada no ano 2000 indicou que 80% dos processadores existentes eram usados em algum tipo de aplicação embarcada [Tennenhouse, 2000]. É razoável afirmar que este número tenha aumentado, dado que hoje em dia pode-se comprar com grande facilidade microcontroladores de 32 bits operando com velocidade de 80MHz por menos de 10 dólares.

Uma evidência desta situação está na indústria automobilística, que é considerada a indústria que mais utiliza microcontroladores no mundo. Estima-se que 33% dos semicondutores usados em um carro são microcontroladores, sendo que um carro popular possui de 30 a 40 microcontroladores, enquanto um carro de luxo pode possuir mais de 70 microcontroladores [Sangiovanni-Vincentelli et al., 2007, Parab et al., 2007]. Dessa forma, os custos de fabricação de veículos estão atingindo proporções até então somente existentes na indústria da aviação, sendo 1/3 do custo do veículo gasto na carroceria, 1/3 no motor, e 1/3 em eletrônica [Wolf, 2007a].

Estima-se que em 2010, os softwares embarcados que são executados em um veículo irão possuir 100 milhões de linhas de código [Sangiovanni-Vincentelli et al., 2007]. Um telefone celular comum possui 2 milhões de linhas de código, sendo esperado que em 2010 possua 20 milhões de linhas de código [van Genuchten, 2007]. O que é preocupante é o fato de que em média para cada 1000 linhas de código escritas, existem cerca de 20 defeitos [Taurion, 2005]. Na verdade, em alguns casos, o número de defeitos a cada 1000 linhas de código pode chegar a 50, contudo na indústria, o *software* final entregue pode conter de 1 a 25 defeitos por 1000 linhas de código [McConnell, 2004]. Com base nestas afirmações, no ano de 2010, um veículo poderá possuir 2 milhões de defeitos em seu *software*.

Além disto, pesquisas demonstram que os projetos de sistemas embarcados têm sido entregues fora do seu prazo. Observou-se que em média 50% dos projetos têm seus cronogramas atrasados em pelo menos 4 meses e cerca de 11% dos projetos acabam sendo cancelados, além de pelo menos 30% dos projetos não atingirem 50% das especificações propostas inicialmente [Taurion, 2005]. Estes fatos demonstram a necessidade de se construir componentes de *software* confiáveis e reutilizáveis, para reduzir custos, prazos e aumentar a confiabilidade dos produtos.

Neste trabalho são discutidos sistemas operacionais de tempo real, que fornecem uma infraestrutura confiável para se construir sistemas de tempo real. Um sistema de tempo real nada mais é do que um sistema capaz de processar dados, e gerar uma resposta antes de um tempo máximo predeterminado para cada situação. Aplicações como controle de sistemas em computadores de bordo de veículos, pilotos automáticos de aviões, controladores de robôs e máquinas industriais, bem como a maioria dos sistemas de controle e aquisição de dados encaixam-se na categoria de sistemas de tempo real [Barabanov, 1997]. De acordo com Timmerman e Perneel, nestes sistemas o importante é a previsibilidade do sistema e não

a sua velocidade média [Timmerman e Perneel, 2001].

O termo *Tempo Real* muitas vezes é confundido com um sistema que é simplesmente rápido [Li et al., 1997]. Isto é um mito que é discutido em mais detalhes posteriormente neste trabalho. Na área da engenharia e computação, o sentido deste termo é usado de uma forma mais específica, significando o grau de determinismo de um sistema. Dessa forma, a execução de algumas tarefas pode ser lenta, mas deve ocorrer em um intervalo de tempo predeterminado, sendo que o não cumprimento deste tempo muitas vezes está relacionado com incidentes fatais para os sistemas e até mesmo para a vida humana.

Falhas em sistemas de tempo real podem ser catastróficas, como em um incidente que aconteceu em 1991 durante a guerra do Golfo, onde 27 soldados americanos morreram e 97 ficaram feridos devido a uma falha que ocorreu em um sistema de defesa anti-míssil. Esta falha ocorreu por um erro já conhecido pelos desenvolvedores no *software* de controle do sistema de defesa anti-míssil *Patriot*. O erro é chamado de *software aging* ou envelhecimento de *software*.

A falha ocorreu pois o cálculo da trajetória para atingir um alvo inimigo utilizava parâmetros como velocidade e o tempo. Quando um ataque ocorreu, o sistema estava ligado fazia muito tempo, e seus contadores de tempo tinham acumulando um erro de uma fração de segundos, que levou o sistema a errar o alvo, permitindo o sucesso do ataque inimigo. Isto demonstra que provavelmente o tempo é a entidade mais importante em um sistema de tempo real [Levi e Agrawala, 1990]. Sabendo do problema, os engenheiros do projeto *Patriot* avisaram o exército que o uso prolongado do sistema poderia impactar negativamente na localização dos alvos, recomendando que o sistema deveria ser reiniciado regularmente. Como os oficiais do exército acharam que o sistema nunca ficaria em operação tempo suficiente para este problema tornar-se eminente, eles não especificaram para os soldados a regularidade com que o sistema deveria ser reiniciado [Grottke e Trivedi, 2007].

Embora a idéia de reiniciar regularmente um sistema possa parecer um tanto quanto rudimentar e até incorreta, muitas vezes esta é a melhor solução quando um *bug* não pode ser encontrado. Este conceito de reiniciar o sistema periodicamente é chamado de "rejuvenação", e é usado em várias áreas da computação.

Um dos grandes problemas dos sistemas embarcados de tempo real hoje em dia, é como obter confiabilidade no *software* desenvolvido, já que atualmente não é possível desenvolver *softwares* confiáveis a um custo acessível [Henzinger et al., 2007]. Além de ser fundamentalmente difícil construir um sistema embarcado, ainda não entende-se completamente todas as características destes sistemas [Wolf, 2007b]. É um consenso que sistemas embarcados são difíceis de modelar, de projetar e de especificar [Ostroff, 1992].

Tendo em vista os argumentos discutidos, sabe-se que os softwares de tempo real podem conter muitos defeitos, sendo alguns nem mesmo conhecidos ou testados previamente. Isto torna interessante a construção de componentes de software confiáveis e reutilizáveis para se usar no desenvolvimento de sistemas de tempo real.

Um exemplo desta padronização também está na indústria automobilística. Dada a complexidade das unidades de controle eletrônico dos carros *(Eletronic Control Units* ou ECUs), o *software* tornou-se cada vez maior e mais complicado, de forma que alguns fabricantes de carros passaram a utilizar sistemas operacionais de tempo real para ter meios mais poderosos de atender as severas demandas de tempo real [won Lee; Sung-Ho Hwang; Jae Wook Jeon, 2006].

Um segundo exemplo está nos novos sistemas de aviação que concentram vários subsistemas de controle de um avião em um único computador, necessitando de sistemas operacionais com mecanismos de particionamento espacial e temporal [Krodel e Romanski, 2007]. A partição espacial refere-se ao isolamento das tarefas na memória, e a temporal ao escalonamento de tarefas, dividindo o tempo do processador. Estas partições permitem a um único processador executar várias tarefas ao mesmo tempo, sem que nenhuma delas cause interferências nos requisitos temporais das demais tarefas.

A execução de vários subsistemas em um mesmo computador, que antes eram executados em diferentes computadores é um desdobramento do grande poder de processamento disponível, que leva cada vez mais os projetistas a adicionarem mais recursos aos seus sistemas que possuem usualmente um único processador. Para atender a estas demandas de forma adequada, os sistemas operacionais acabaram tornando-se essenciais para simplificar o projeto de *software* em sistemas embarcados, deixando para o programador apenas a preocupação com o desenvolvimento de cada uma de suas aplicações.

Para sistemas em que alta confiabilidade é um requisito, os sistemas operacionais de tempo real ou *Real Time Operating Systems* (RTOS) devem ser utilizados por serem sistemas projetados e preparados para funcionarem de forma determinística, em contraposição aos sistemas operacionais convencionais (como Windows ou Linux) de propósito geral. Estes são utilizados usualmente nos computadores pessoais, sendo projetados para terem o menor tempo de resposta possível ao usuário, favorecendo a interatividade com o usuário e não o sistema controlado.

De acordo com Vetromille, o sistema operacional é o *software* mais importante em um sistema de tempo real embarcado [Vetromille et al., 2006]. De fato, o sistema operacional oferece a infraestrutura básica fundamental para construir um sistema de tempo real. Por outro lado, deve-se levar em consideração que ele aumenta o tempo de resposta do sistema [Kohout et al., 2003].

Mesmo com todos os problemas envolvidos, e diversas ferramentas e teorias de tempo real, ainda existe uma lacuna muito grande entre indústria e pesquisa na área de sistemas operacionais de tempo real, sendo que a indústria ainda tem usado a abordagem de tentativa e erro em várias ocasiões para construir sistemas de tempo real [Timmerman e Perneel, 2001].

No passado (e até mesmo atualmente), muitas aplicações de tempo real são desenvolvidas sem um sistema operacional. Contudo, com o baixo custo e alto poder computacional disponível atualmente, as aplicações estão cada vez mais complexas, criando uma tendência de se utilizar um sistema operacional para gerenciar a complexidade do *hardware* que está por trás do sistema [Bouyssounouse, 2005].

Entretanto, é sempre importante projetar e dimensionar o sistema, já que existem situações onde o uso de um sistema operacional pode não ser necessário, ou até mesmo proibitivo. Em muitos casos, um ambiente sem sistema operacional, com um *loop* ou uma arquitetura dirigida a interrupções pode levar a uma solução mais fácil de manter, desenvolver e testar [Ganssle, 2006].

De qualquer forma, em 1994 já era fortemente recomendada a compra de um RTOS para desenvolver um sistema embarcado [Jensen, 1994], dado que alguns desenvolvedores resolviam desenvolver todo sistema "do zero", incluindo o RTOS, e freqüentemente perdiam o prazo, além do produto ainda apresentar falhas em sua versão final.

Ainda é muito discutido nesta área se é necessário construir um sistema operacinal de tempo real "do zero" ou deve-se utilizar um sistema operacional já existente. Na verdade, esta análise deve ser realizada caso a caso [Laplante, 2004]. Erros comuns que certamente ocorreriam em um desenvolvimento do zero, não ocorrem em sistemas maduros e já testados em várias situações pelos fornecedores. Por outro lado, estes sistemas são bastante completos, freqüentemente oferecendo diversas opções além do esperado, causando *overhead* pelo gerenciamento das tarefas.

Muitas empresas especializaram-se no desenvolvimento e comercialização deste tipo de sistema, originando sistemas operacionais de tempo real consolidados no mercado, como o *VxWorks*, *LynxOS*, *VXRT*, *QNX* e *Integrity* dentre outros. Até mesmo a Microsoft oferece uma versão *Real Time* do Windows CE.

Um outro aspecto a ser abordado neste trabalho, consiste no uso e na análise de qualidade de ferramentas de *Software Livre* no desenvolvimento de aplicações confiáveis para utilização em aplicações críticas em situações reais. Este tipo de solução vem sendo procurada recentemente, especialmente devido ao alto custo das soluções proprietárias, e as possibilidades de personalização de sistemas livres de *royalty* e licenças como o Linux, graças à sua vasta documentação e comunidade [Lennon, 2001]. Versões de tempo real do Linux vem sendo amplamente utilizadas em anos recentes [Caccavale et al., 2005]. Embora a comunidade industrial ainda tenha um pouco de resistência, o Linux vem mostrando-se cada vez mais uma solução viável, flexível e acessível para os mais diversos casos de controle de tempo real [Proctor, 2002].

Neste trabalho são estudados sistemas operacionais de tempo real que estão presentes hoje em dia nas mais diversas aplicações sem nem mesmo serem percebidos. Muitos sistemas embarcados tem requisitos de tempo real, como sistemas anti-travamento de freios de veículos (ABS), sistemas de controle de aviões, de robôs, e até mesmo receptores de sinal de televisão por satélite, dentre muitos outros.

Além disso, a massificação da produção de processadores de propósito genérico (especialmente compatíveis com PC), vem permitindo cada vez mais desenvolver aplicações que utilizam processadores genéricos para controlar processos, em casos em que anteriormente equipamentos caros e complexos precisavam ser usados. Esta tendência é tão grande, que hoje é comum encontrar PCs da arquitetura x86 realizando o controle de robôs industriais amplamente vendidos, mesmo o PC não sendo um computador projetado

para atividades industriais e nem mesmo de tempo real. Contudo, acredita-se que os PCs podem efetuar tarefas de controle em robôs com mais flexibilidade e menor custo que computadores industriais tradicionais [González et al., 2003]. Dessa forma, observa-se um rápido crescimento do uso de PCs em aplicações de tempo real, onde antigamente computadores dedicados e projetados especialmente para determinadas funções eram utilizados [Puthiyedath et al., 2002].

Este trabalho foca na arquitetura e *software* de computadores compatíveis com o padrão IBM-PC, embora na computação de tempo real seja comum usar outras arquiteturas como PowerPC e Motorola 68K. Ao discutir uma arquitetura diferente da usada nos IBM-PC ao longo do texto, uma observação será realizada.

As arquiteturas dos computadores modernos oferecem técnicas para aumentar a performance média do processador, como várias unidades de execução, vários níveis de *pipelines*, predição dinâmica de saltos, execução especulativa, renomeação de registradores, *cache* no processador, dentre outros. Estas técnicas tornam difícil de se determinar o momento exato que uma tarefa será executada, tornando o sistema menos previsível [Kailas et al., 1997]. O fato é que estas técnicas são usadas para melhorar a velocidade média das máquinas, o que de fato ocorre, entretanto estas técnicas destroem o determinismo do sistema [Laplante, 2004]. Em alguns casos, um velho PC 486 ou um Motorola 68K podem fornecer melhores características de tempo real do que um moderno Athlon ou Pentium-III [Sohal, 2001].

Várias mudanças aconteceram no mercado de RTOS recentemente. Uma das empresas mais conhecidas do ramo, a Wind River, possui além do tradicional VxWorks, uma versão de Linux de tempo real [Wind River, 2006]. Além desta versão de Linux, a Wind River adquiriu em 2007 a propriedade do RTLinux, da FSMLabs [Systems, 2007], uma versão parcialmente gratuita de *Real Time* Linux muito utilizada na indústria. Outro sistema operacional muito conceituado, o Neutrino da QNX teve seu código fonte aberto e disponibilizado para *download* gratuitamente no final de 2007. Embora gratuito, uma licença comercial deve ser paga quando um sistema que usa Neutrino for comercializado.

> "Uma aplicação não pode ser mais segura que o sistema operacional onde ela roda" [Klingsheim et al., 2007].

Embora tal afirmação seja específica para segurança da informação, levando-se em conta a proteção fornecida pelo sistema operacional para que um aplicativo não obtenha dados da área de memória de outro aplicativo, pode-se estender esta afirmação para sistemas de tempo real, em que é desejável, que o mal funcionamento de um programa, como invasão voluntária ou involuntária da área de memória de outro programa, seja evitada e tratada pelo sistema operacional. Isto confirma-se em um artigo da IBM onde afirma-se que sistemas responsivos são tão bons quanto as plataformas que os suportam [McKenney, 2008].

É importante ressaltar que os sistemas operacionais de tempo real não são a panacéia dos sistemas de tempo real, pois por mais seguro que o sistema operacional seja, ele não pode evitar falhas de lógica e de

programação realizadas pelo programador, ou mesmo de operação incorreta do sistema pelos usuários.

Ao longo do texto, várias informações de diferentes sistemas operacionais de tempo real serão analisadas. De acordo com Laplante, a informação mais importante de um RTOS são os piores casos de resposta, sendo que estes dados normalmente não são divulgados pelos fabricantes, mesmo que sejam conhecidos, tentando evitar qualquer risco de desfavorecer seus produtos [Laplante, 2004]. Uma das analises realizadas neste trabalho é exatamente esta.

É importante observar que nem todos os aspectos e detalhes dos sistemas operacionais de tempo real são discutidos neste trabalho. Os itens discutidos aqui foram selecionados por terem relação com os testes e analises realizados ao longo desta pesquisa. A discussão de todas os mecanismos e características de um sistema operacional de tempo real necessitaria de uma discussão muito mais ampla.

1.1 Influência em experimentos

Outro aspecto importante e motivador deste trabalho, é a influência de sistemas operacionais em experimentos e medidas de laboratório. Enquanto antigamente eram utilizados *data loggers* e equipamentos dedicados para realizar experimentos e medidas de laboratório, atualmente observa-se cada vez mais uma tendência de utilizar-se diretamente o PC para realizar medidas e monitoramento em tempo real.

Um problema comum, é o fato de que a maioria dos pesquisadores e usuários de computadores estão acostumados a usar o Windows para realizar estes experimentos. De fato, Cinkelj verificou que é viável utilizar o Windows XP em tarefas de *soft real time*, como aquisição de dados a 10KHz. Neste experimento, verificou-se que o Windows apresentou um *jitter* máximo de 1ms [Cinkelj et al., 2005]. Isto demonstra que se o Windows pode ser usado com alguma segurança para fazer medidas que não necessitem precisão inferior a 1ms, contudo é importante certificar-se de que o Windows não esteja sobrecarregado durante as medidas.

Laplante, um dos mais ativos pesquisadores na área de tempo real, sugeriu recentemente que o uso de sistemas operacionais de tempo real para realizar experimentos e medidas em laboratórios é uma necessidade [Cedeno e Laplante, 2007]. Isto deve-se ao fato das medidas de laboratório estarem muitas vezes relacionadas a medidas entre interrupções, contudo o tempo entre a geração da interrupção e o momento em que o processador executa a rotina relacionada ao seu tratamento pode ser uma causa de erro em medidas [Bergman, 1991].

Com isto, muitos fabricantes possuem produtos que podem induzir usuários leigos a cometerem sérios erros de medidas. Talvez por conseqüência do termo tempo real estar na moda, muitos pesquisadores hoje utilizam ferramentas como *Matlab* ou *Labview* com placas de aquisição, acreditando que estão realizando medidas em tempo real. Infelizmente, tais medidas não são de tempo real, além de poderem estar completamente incorretas, especialmente se executadas em sistemas como o Windows. É importante ressaltar que tanto o *Labview* quanto o *Matlab* possuem soluções específicas de *hardware* e *software* para

implementar sistemas de tempo real com confiabilidade.

Um pico na utilização da rede ou a gravação de um CD, ou até mesmo o uso do mouse ou do teclado podem ter prioridade superior às interrupções da placa de aquisição, de forma que o momento em que a medida foi feita e o momento em que seu tempo foi determinado e ela foi armazenada podem ser completamente incoerentes.

A própria *National Instruments*, empresa líder em equipamentos de medição e entrada/saída, não apenas conhece este problema, como possui uma página explicativa alertando seus consumidores a respeito deste problema, e sugerindo usar um modelo especial de placa de aquisição que possui um *buffer* dedicado para as medidas. Mais informações podem ser encontradas no Tutorial de tempo real da *National Instruments* [National Instruments, 2007b]. Nesta placa, todas aquisições de dados são realizadas e armazenadas em uma memória especial da própria placa de aquisição, de forma que a garantia de tempo real é oferecida por um *hardware* dedicado de alta confiabilidade [National Instruments, 2007a]. Somente após a aquisição dos dados ser realizada para o *buffer*, é que os dados deste *buffer* são transmitidos para a memória principal do PC.

Dessa forma, em certas situações é aconselhável utilizar antigos *data-loggers*, que de fato são de tempo real (embora não coloquem isto em sua propaganda).

É importante notar que as ferramentas aqui mencionadas são de ótima qualidade, e os possíveis erros estão no mau uso da ferramenta.

1.2 Organização do texto

Nesta dissertação, primeiramente são apresentados conceitos de tempo real, de sistemas de tempo real e de sistemas operacionais. Em seguida, são discutidas algumas peculiaridades da programação de sistemas de tempo real e são apresentadas as ferramentas desenvolvidas ao longo deste trabalho. Na seqüência, apresenta-se a análise experimental realizada nos sistemas operacionais de tempo real selecionados, incluindo os materiais e métodos utilizados, e por fim, as considerações finais são realizadas incluindo tendências para a área de sistemas operacionais de tempo real e possibilidades de trabalhos futuros na área.

2 Objetivos

O objetivo deste trabalho é realizar uma análise quantitativa e qualitativa de alguns sistemas operacionais de tempo real, para verificar suas capacidades, vantagens e desvantagens na implementação de aplicações de robótica e automação.

Os sistemas selecionados para análise são o MS Windows XP, Linux, VxWorks, μC/OS-II, MS Windows CE, QNX Neutrino e o RTAI (uma extensão de *kernel* de tempo real para o sistema operacional Linux). A escolha destes sistemas deu-se devido a sua popularidade, e sua disponibilidade no laboratório onde os experimentos foram realizados.

De acordo com Taurion, comparar sistemas operacionais de tempo real não é uma tarefa fácil. Como eles diferem muito nas funcionalidades e implementação, nem sempre eles são diretamente comparáveis [Taurion, 2005]. Além disso, os especialistas da *Dedicated Systems* afirmam que não é possível medir características de um RTOS com confiabilidade sem o uso de *hardware* externo, pois durante os testes, é o próprio sistema operacional que gerencia os temporizadores (*timers)* da máquina em testes, afetando as medidas [Beneden, 2001]. A *Dedicated Systems* é uma empresa que possui como uma de suas especialidades, a execução de testes sistemáticos em sistemas operacionais.

A análise quantitativa realizada no decorrer desta pesquisa foi feita testando os sistemas operacionais através de um "teste de caixa preta", onde estímulos externos foram gerados, e equipamentos de medida externos determinaram o tempo de resposta para cada sistema. Após os testes, implementou-se sistemas robóticos baseados nestes sistemas, que foram utilizados como caso de testes, e tornaram-se ambiente de testes para futuras pesquisas na área.

Também foi necessário pesquisar o domínio das aplicações de robótica e automação para saber a partir dos resultados dos testes, se os sistemas operacionais testados são confiáveis e indicados para serem utilizados em sistemas de controle em tempo real. O grupo de usuários OMAC [1] foi criado em 1994, e já conta com 500 empresas de automação industrial e áreas correlatas associadas. A idéia da OMAC é de estabelecer regras e padrões a serem seguidos por seus membros que estabelecem limites de tempo para uma tarefa ser considerada de tempo real, bem como padrões para que equipamentos possam trocar informações entre si.

A equipe de desenvolvimento de *PowerTrain* [2] para carros da GM, em conjunto com a OMAC estudaram diversos tipos de sistemas de tempo real na indústria e concluíram que 95% deles requerem ciclos de 1ms ou mais [Microsoft, 2007a]. Esta informação é compatível com a realidade da mecatrônica, pois sistemas mecânicos como robôs tem freqüências entre 5 a 10Hz. Com isso, controladores trabalhando a 100Hz (ciclos de 10ms) seriam suficientes. A informação de 1ms será utilizada como referência de qualidade, sendo esperado que nesta pesquisa todos sistemas tenham um determinismo de tempo real inferior

[1] *Open Modular Architecture for Control.* Maiores informações em http://www.omac.org/
[2] Conjunto de todos componentes responsáveis por gerar potência e a transmitir para as rodas.

a 1ms.

3 Tempo Real

Neste capítulo serão discutidos conceitos sobre sistemas de tempo real e suas classificações, bem como a entidade mais importante de um sistema de tempo real, o tempo, e como ele é medido.

3.1 Sistemas de *Tempo Real*

Em uma época em que as operações realizadas por computadores levavam muitas horas ou dias (normalmente utilizava-se processamento em lote ou *batch*), alguns pesquisadores começaram a se interessar por sistemas que respondessem mais rapidamente às requisições dos usuários. O que os projetistas dos sistemas computacionais buscavam era que a resposta fosse suficientemente rápida para atender a a uma determinada solicitação sem que os usuários tivessem que esperar muito tempo.

Um dos primeiros livros sobre o tema, foi publicado em 1967 por James Martin [Laplante, 2004]. Neste livro, o autor não chega a discutir sistemas de controle, que hoje são exemplos típicos de sistemas de tempo real, mas ele limita-se a operações computacionais bancárias, de companhias financeiras e outros tipos de empresas. Naquela época os computadores já eram capazes de executar cálculos e tomar decisões em poucos segundos ou milisegundos, entretanto as operações de entrada e saída eram muito lentas, devido a necessidade de uso de fitas magnéticas e impressoras. Nesta mesma época, um grande leque de possibilidades de telecomunicação de dados e voz já estava surgindo, convergindo para possibilidades promissoras.

Na introdução de seu livro, Martin apresenta uma visão futurista, descrevendo a possibilidade de um cliente poder sacar seu dinheiro guardado em um banco através de "máquinas automáticas", instaladas em qualquer parte do mundo, sendo o dinheiro imediatamente debitado de sua conta na matriz do banco. Vendedores poderiam enviar pedidos para sua matriz em outro país, para serem processados já no próximo dia [Martin, 1967].

Após descrever diversos casos onde um sistema de tempo real seria interessante, como o caso de caixas automáticos de bancos, o autor conclui sua explanação com o termo: *"This is Real Time"* ou "Isto é Tempo Real".

Nesta obra clássica, o termo Tempo Real é definido como:

> "Um sistema computacional de tempo real pode ser definido como aquele que controla um ambiente recebendo dados, processando estes dados, e executando uma ação ou retornando resultados suficientemente rápido para afetar o funcionamento do ambiente naquele momento"

Mais de 40 anos depois esta definição ainda é atual e consistente com os sistemas de tempo real. Existem diferentes tempos de resposta para sistemas de tempo real. Equipamentos de bordo em aeronaves devem responder solicitações em poucos microsegundos, enquanto telefones móveis podem responder em alguns

milisegundos, ou mesmo em um caixa automático de bancos onde as respostas não devem ultrapassar 100 milisegundos (para não irritar os clientes).

Como será discutido neste capítulo, o termo tempo real é utilizado freqüentemente de uma forma inadequada. Mesmo na época de Martin, onde sistemas de tempo real eram pouco difundidos, já existiam algumas divergências entre os especialistas sobre a definição do termo tempo real. Alguns diziam que era um sistema que poderia dar uma resposta em poucos segundos, enquanto outros definiam como um sistema que poderia controlar um ambiente em um modo "minuto a minuto" ou "hora a hora". [Martin, 1967].

Isto leva a definição mais aceita atualmente de um sistema de tempo real:

> "Um sistema de tempo real é aquele que deve satisfazer explicitamente restrições de tempo de resposta podendo ter conseqüências de risco ou falha não satisfazendo às suas restrições" [Laplante, 2004]

A norma alemã DIN 44300 especifica que um sistema de tempo real é aquele onde os resultados das entradas estão sempre disponíveis em períodos de tempo predeterminados [Amianti, 2008, Timmerman, 2001].

A IEEE também possui uma definição para sistemas de tempo real que descreve tal sistema como aquele onde os resultados da computação podem ser usados para controlar, monitorar ou responder a um evento externo em tempo [IEEE, 1990].

Com estas definições, já pode-se concluir que um sistema de tempo real não precisa ser necessariamente rápido. Se sua função é encher latas de um produto alimentar a cada 2 minutos, o sistema deve encher as latas a cada 2 minutos, caso contrário pode haver perdas financeiras para a empresa ou mesmo a lata não estar no local correto. Um mito comum que não é verdadeiro, é dizer que um sistema de tempo real é rápido. Isto está normalmente associado ao fato dos sistemas de tempo real serem usados em aplicações militares e de aviação que precisam de respostas rápidas, contudo nem todas situações aplicam-se a estes requisitos. Computação de tempo real não quer dizer execução rápida, mas sim cumprimento de prazos [Wolf, 2007b]. Dessa forma, um sistema é dito de tempo real se o sinal de controle é gerado suficientemente rápido para suprir as necessidades do processo [Dupré e Baracos, 2001].

Geralmente, aplicações de tempo real realizam operação repetitivas com um intervalo de tempo definido. O fator mais importante nestas tarefas é o determinismo [Relf, 2007]. Um sistema é considerado de tempo real, se a amplitude de erro de tempo do intervalo de uma tarefa é pré-definida, podendo ser sempre repetida [Relf, 2007]. Este fator mede o quanto o erro varia, tratando-se do *jitter*.

O *jitter* é definido pela IEEE como uma variação abrupta e falsa relacionada a algum período de tempo [Cinkelj et al., 2005]. O *jitter* pode ocorrer por vários motivos, dentre eles variação no relógio, *branches* (saltos) no código e decisões de escalonamento [Cinkelj et al., 2005].

O uso do termo "tempo real" sozinho pode não dar uma informação completa do tipo de sistema sendo controlado. Um erro no sistema de tempo real de uma usina nuclear seria catastrófico, enquanto em uma máquina de encher latas de produtos alimentares poderia causar danos financeiros, mas não chegaria a causar uma catástrofe. Dessa forma, os sistemas de tempo real podem ser classificados em três categorias:

- *Hard Real Time*: Consiste de um sistema de tempo real onde sua falha pode acarretar conseqüências desastrosas, tanto financeiramente, quanto com relação a perda de vidas humanas. Um exemplo é o sistema de suporte a vida humana em um equipamento médico;
- *Soft Real Time*: É um sistema de tempo real em que sua falha ou perda de prazos, não causa nenhum dano, contudo o produto ou sistema deixa de cumprir sua função. Um exemplo é um aparelho reprodutor de vídeos em DVD;
- *Firm Real Time*: É um sistema intermediário entre os outros dois, onde um acúmulo de muitas falhas em um sistema *Soft Real Time* pode transformá-lo em um sistema *Hard Real Time*.

Sistemas de tempo real são por natureza multidisciplinares, envolvendo diversas áreas da computação e da engenharia. A figura 1 mostra a natureza interdisciplinar de sistemas de tempo real.

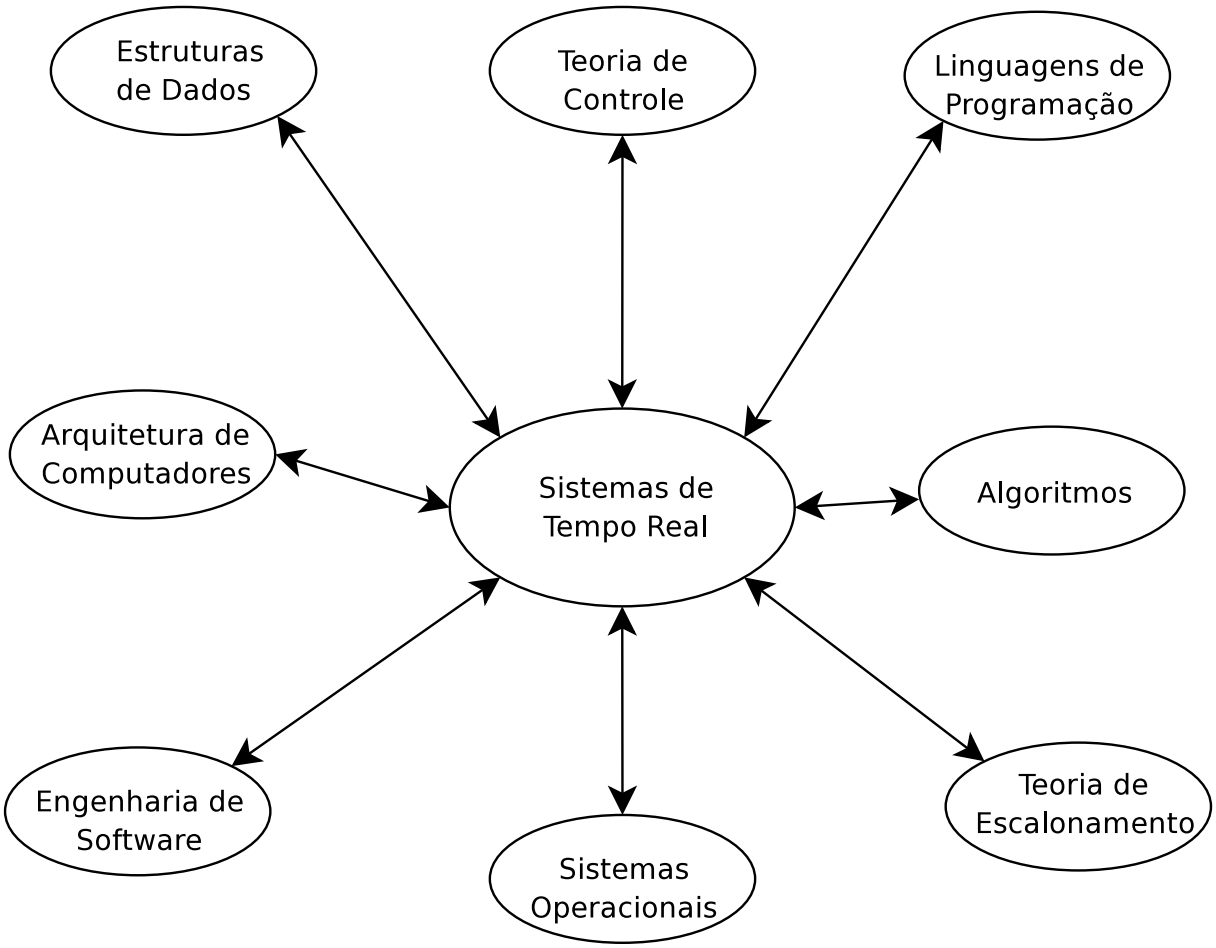


Figura 1: Áreas correlatas aos sistemas de tempo real. Adaptado de [Laplante, 2004]

Como apontado anteriormente, os sistemas de tempo real são cercados de diversos mitos, como a afirmação de que sistemas de tempo real são sinônimos de sistemas "rápidos" e a crença de que existem metodologias universais amplamente aceitas para especificação e projeto de sistemas de tempo real [Laplante, 2004]. Estes mitos também são apresentados em [Stankovic, 1988]. Em sua discussão, Stankovic ainda enfatiza o amplo uso incorreto do termo *real time* feito por pesquisadores, empresas e governos, além do fato de muitos pesquisadores não considerarem a pesquisa em tempo real científica. O autor convida pesquisadores a olharem para sistemas de tempo real de um ponto de vista mais científico, para tentar suprir as demandas e necessidades de tempo real nos anos por vir.

É interessante o fato de 20 anos terem se passado desde a publicação deste artigo e os mitos continuarem fortes e presentes em empresas e até entre programadores qualificados. Em resposta ao clássico artigo de Stankovic, Kurki-Suonio publicou novos conceitos mal usados na área de tempo real com uma abordagem mais formal e com contrastes entre modelos que envolvem tempo real e que não envolvem tempo real [Kurki-Suonio, 1994].

No início do ano de 2008, o site http://www.real-time.org foi criado em mais uma tentativa de esclarecer mal entendidos e mitos sobre o termo *real time*. E. Douglas Jensen, seu criador, tem larga experiência na área de sistemas de tempo real e trabalha em projetos do departamento de defesa dos Estados Unidos.

3.2 Aplicações de sistemas de tempo real

Dependendo do ponto de vista, e das restrições de tempo, qualquer sistema computacional pode ser considerado um sistema de tempo real. Até mesmo um editor de textos pode ser considerado um sistema de tempo real (*Soft Real Time*). Podemos citar algumas aplicações que utilizam-se de sistemas de tempo real:

3.2.1 Eletrônica de consumo

Grande parte dos produtos eletrônicos e eletrodomésticos vendidos hoje em dia possuem algum tipo de processamento. Os fornos microondas devem temporizar corretamente o tempo em que as bobinas de microondas aquecem os alimentos, enquanto aparelhos reprodutores de DVD devem reproduzir os vídeos com fidelidade e sem saltos indesejados. Telefones celulares, devem codificar a voz em formato digital e transmiti-lá através de diversos protocolos sensíveis a um compartilhamento temporal do canal de comunicação, como no caso das redes TDMA (*Time Division Multiple Access*).

3.2.2 Indústria automotiva

No caso da indústria automobilística, onde já se tem softwares embarcados cuidando de tarefas críticas de tempo real, como frenagem e aceleração, falhas podem não apenas causar perdas materiais, mas

também perdas de vidas.

Muitos veículos populares já utilizam o conceito de *drive by wire*, um termo inspirado no *fly by wire* da aviação. No caso dos veículos com injeção eletrônica, não é mais um cabo de aço que conecta o pedal do acelerador ao motor, mas sim um sinal elétrico que é enviado do acelerador para um processador que é responsável por repassar a aceleração desejada para o motor. Isto permite controlar melhor acelerações bruscas, evitando que o motor falhe, gaste muito combustível ou seja forçado a trabalhar fora das especificações.

Os freios anti-travamento de rodas ABS também funcionam de forma coordenada por meio de um processador que recebe comandos de frenagem, os processa e repassa aos atuadores de freio nas rodas. Nos veículos de luxo, já existem sistemas de suspensão ativa, onde sensores inteligentes detectam o tipo de terreno e suas irregularidades, informando estes dados a um processador que controla a posição e altura de cada roda do veículo a cada momento.

3.2.3 Medicina

Equipamentos médicos que medem pressão arterial, batimento cardíaco, fluxo de oxigênio e outros dados devem ser precisos. Todas estas medidas normalmente tem uma base de tempo, como por exemplo "batidas por minuto" ou "litros por minuto". Estes dados também podem ser processados e gerar alarmes para médicos ou equipes de enfermagem.

Outra área em constante evolução na medicina, é a robótica aplicada a medicina [Versweyveld, 1999], para conduzir intervenções cirúrgicas remotamente [Ghodoussi et al., 2002], onde cirurgiões especializados podem controlar um braço robótico a distância para operar um paciente em locais de difícil acesso, ou onde não existe um especialista da área. Esta tecnologia envolve diversos problemas de tempo real, pois não apenas o controle robótico precisa possuir uma precisão milimétrica, como pequenas latências ou problemas nos enlaces de comunicação poderiam comprometer o ato médico.

3.2.4 Telecomunicações

A área de telecomunicações é uma das maiores utilizadoras de sistemas de tempo real. Em telecomunicações é muito comum se falar de QoS (*Quality Of Service*) que são parâmetros de qualidade da rede, como baixa latência, alta velocidade, dentre outros. Para reduzir custos, é prática comum entre as companhias telefônicas trafegar dezenas de ligações telefônicas convencionais através de suas redes de dados IP (*Internet Protocol*) evitando a necessidade de canais dedicados típicos da telefonia convencional. Este processo é chamado de Voz sobre IP ou *Voice over IP* (VoIP). Atualmente o VoIP já chegou aos usuários finais com programas como o *Skype* e outros. VoIP é uma aplicação de *soft real time*. Como a Internet (rede mundial de computadores) não oferece garantias de tempo, é comum observar altas latências na comunicação VoIP, como por exemplo o interlocutor do outro lado da ligação demorar diversos segundos

para escutar o que acaba de ser dito. Isto ocorre porque os limites de tempo não são cumpridos. Outro exemplo da perda de requisitos temporais é a voz robotizada.

Novas tecnologias como televisão digital e vídeo conferência também impoem requisitos ainda mais rigorosos às redes de comunicação.

3.2.5 Robótica/Automação

Outra área que faz grande uso de sistemas de tempo real é a área de robótica e automação. Máquinas industriais, como tornos CNC, devem executar operações com base em tempos pré-definidos, sendo o cumprimento dos tempos essencial para que o resultado seja preciso.

Os robôs possuem vários subsistemas. Os sistemas de mais baixo nível que realizam os laços de controle, como malhas PID (Proporcional-Integral-Derivativo) devem ser executados a cada 100, 10 ou 1ms. Operações de cálculo como integrais ou derivadas são executadas com base no tempo, de forma que a precisão das malhas de controle está diretamente relacionada com a precisão de medir o tempo dos computadores que estão executando os sistemas. Em níveis mais altos, existem requisitos de tempo real mais brandos, como em enlaces de comunicação e interface com usuário.

3.2.6 Indústria militar e Aeroespacial

Hoje em dia a cabine de controle dos aviões vêm sendo chamadas de *glass cockpit* [Sweet, 1995] (cockpit de vidro) pois ao invés de se ver os equipamentos que eram visto antes, ao olhar para uma cabine de avião, só se vê *displays* e telas de computadores.

Os sistemas aviônicos (uma contração de *aviation* e *eletronics*) atuais substituíram os cabos de aço, de forma que os atuadores que controlam as superfícies, como *flaps*, profundor e leme são controlados por um sistema chamado *fly by wire*. Neste sistema, todos os comandos feitos no *joystick*, manche, pedal, ou qualquer controle do piloto são transformados em dados digitais, enviados para um computador, que processa os dados e os envia eletronicamente para os atuadores [Knight, 2007]. Tal sistema impede que um avião seja forçado fora de seus limites, filtrando manobras inadequadas para movimentos seguros. O *fly by wire*, aliás, já era usado em 1984 no primeiro sistema aviônico do ônibus espacial através de um barramento de 1MHz [Spector e Gifford, 1984].

No caso militar, sistemas de detecção de inimigos e armas com alto poder de fogo devem obedecer a regras rígidas e o momento de disparo deve ser exato, para que o alvo não seja incorreto.

3.2.7 Experimentos científicos

Outra área que utiliza bastante sistemas de tempo real é a científica, tanto para realizar medidas de experimentos com precisão, quanto para controlar ensaios acionando atuadores nos momentos corretos durante a condução dos experimentos. Aceleradores de partículas, telescópios, experimentos químicos e

físicos são alguns exemplos de ambientes onde os sistemas de tempo real deveriam ser utilizados.

Os exemplos acima são apenas uma pequena amostra dos domínios onde os sistemas de tempo real estão presentes. Certamente, esta lista é bem mais ampla. Os sistemas de tempo real podem consistir de sistemas embarcados ou não. Neste caso, um sistema embarcado, na verdade consiste de um computador de tempo real integrado a um sistema maior [Baskiyar e Meghanathan, 2005, Shaw, 2003].

De acordo com Ganssle [Ganssle, 2004], tempo real e sistemas embarcados são tratados como diferentes áreas no mundo acadêmico, muito embora aplicações embarcadas normalmente sejam de tempo real.

3.3 A medida do tempo

Não se pode falar em sistemas onde o tempo é um fator crítico, sem se falar no tempo e sua medida. Para garantir que tempos e prazos sejam cumpridos, é preciso saber o momento exato da ocorrência de certos eventos com segurança para se tomar determinadas ações. Evidentemente, isto implica na necessidade de realizar medidas de tempo com qualidade e precisão. Além disto, o tempo é um componente fundamental de outras fórmulas matemáticas e funções físicas. É possível, por exemplo, obter-se tamanho, temperatura e massa usando funções relacionadas ao tempo.

Parte da discussão a seguir sobre as formas de se medir o tempo foi adaptada do livro "*From Sundials to Atomic Clocks*" [Jespersen, 1999].

Dentre as grandezas de medida, a medida de tempo é a que possui a natureza mais volátil. A medida de um valor de tensão elétrica, de uma resistência, ou mesmo de uma distância permanecem sempre as mesmas em condições normais. Entretanto, a medida do tempo só é válida no instante em que sua leitura é feita, de forma que o tempo entre a leitura do tempo e o uso desta leitura também pode afetar o verdadeiro significado do tempo.

Desde as primeiras civilizações, a noção de tempo e sua medida já era discutida, dada sua importância para sincronizar eventos e compromissos entre pessoas, bem como acompanhar eventos e acontecimentos naturais. Inicialmente, o tempo era medido por eventos astronômicos, como por exemplo, quantas vezes a Lua ou o Sol apareciam no céu.

Antes de 1965 um segundo era calculado com base no dia solar, e era chamado de segundo solar. Hoje sabe-se que ao longo do ano a duração do dia varia com a mudança das estações, tornando o segundo solar uma medida de tempo insegura devido a sua variação ao longo do ano. Desde então, várias propostas e formas de se medir o tempo e o segundo vem sendo estudadas. O padrão aceito internacionalmente hoje em dia é o segundo atômico. Um segundo atômico consiste de 9.192.631.770 oscilações de um atómo de Césio sem perturbação.

Observando que as entidades podem vibrar, logo chegou-se a conclusão que se estas vibrações fossem periódicas e sempre constantes, seria possível contar o número de vibrações e com isto a passagem do tempo. Para gerar uma freqüência usada para medir o tempo é necessário um oscilador. Um oscilador

ideal seria aquele que após ter um impulso inicial, iria oscilar/vibrar de forma igual indefinidamente. Entretanto sabe-se que esta situação é impossível na natureza devido ao atrito, por exemplo.

Alguns osciladores são melhores que outros. Uma forma de medir a qualidade dos osciladores, é através do fator de qualidade "Q". Q é definido como o número de oscilações que um oscilador realiza até sua energia diminuir para uma pequena percentagem da energia fornecida pelo impulso inicial. O Q normalmente está relacionado ao atrito, de forma que um relógio mecânico tem o Q da ordem de 100, e um relógio de pesquisa (normalmente atômico) tem um Q da ordem de milhões.

Alguns dos primeiros relógios utilizaram o fluxo de água ou de areia, mas os relógios mais eficientes baseavam-se na freqüência de vibração de um pêndulo. Em 1929, foi criada uma pequena cápsula de metal com um cristal de quartzo interno, que vibra ao receber energia elétrica por meio do efeito pizoelétrico. A freqüência fornecida pelo cristal de quartzo é contada, dando origem a maior parte dos relógios modernos. Como será discutido em seguida, não somente os relógios convencionais, mas também os relógios usados em computadores usam o mesmo tipo de cristal de quartzo.

Hoje existem relógios de alguns dólares até relógios de milhões de dólares. É graças a estes relógios mais caros que temos uma referência confiável de tempo para ajustar os relógios menos precisos. Mesmo os relógios de baixo custo já são bastante precisos. Os relógios de quartzo, por exemplo, atualmente têm um erro de 1 minuto por ano. Os melhores relógios baseados em cristais tem um erro inferior a 1ms por mês, enquanto os piores podem desviar 1ms em alguns dias. O erro provém de variações na temperatura, e efeitos de longo prazo, como contaminação do cristal com impurezas, e mudanças internas no cristal devido a vibração, ou outros aspectos do envelhecimento do componente.

Contudo, os relógios (*clocks*) digitais usados em PCs possuem uma grande imprecisão. A maioria dos PCs mede o tempo de duas formas. A primeira é o RTC (*Real Time Clock*) que é um sub componente do PC que mantém a hora através de um cristal de quartzo em conjunto com um pequeno circuito e uma bateria, que em conjunto mantém a hora enquanto o computador está desligado. A outra forma consiste na forma que o computador atualiza seu horário enquanto está ligado. Na prática o PC incrementa seu relógio interno a cada interrupção [Beck et al., 1997] gerada por um *timer*. Ou seja, o relógio é implementado via *software*, o que pode causar problemas devido as características assíncronas das interrupções, e ao estado do computador no momento de cada interrupção dos *timers*. A medida do tempo nos computadores pessoais do tipo PC será discutida em maiores detalhes na seção 3.4.

Uma das grandes vantagens de relógios com alto valor de Q, é que não é necessário perturbar o oscilador constantemente para inserir mais energia para manter o sistema oscilando. Conseqüentemente, sua freqüência natural de ressonância é pouco afetada, e a precisão aumenta, já que as perturbações necessárias para fazer o sistema oscilar nem sempre são da mesma freqüência natural do sistema. Outro fator interessante, é que osciladores de Q alto, somente oscilam perto de sua freqüência natural, aumentando sua estabilidade.

Tipo	Fator Q
Relógio de pêndulo	1000
Relógio de garfo de sintonização	2000
Relógio de Quartzo	10^5- 10^6
Relógio de Rubídio	10^6
Relógio de Césio	10^7- 10^8
Relógio de MASER de hidrogênio	10^9

Tabela 1: Fator Q de relógios. Adaptado de [Jespersen, 1999]

A sofisticação de alguns relógios é tão grande que alguns osciladores possuem inclusive um sistema de malha fechada, com realimentação do erro da freqüência para ter uma precisão ainda maior. Osciladores construídos com um feixe de césio em um tubo têm o fator Q acima de 100 milhões, causando um atraso de 1 segundo em 10.000.000 anos. A tabela 1 mostra o fator Q de alguns relógios.

Um fato curioso, é o fato de muitos relógios elétricos de baixo custo serem alimentados não apenas pela rede elétrica, mas também pela freqüência de 60Hz da rede elétrica, sendo esta a referência para a medida do tempo nestes relógios. Isto pode causar atrasos e adiantamentos nestes relógios, de acordo com as condições da rede de fornecimento de energia elétrica.

3.4 A medida do tempo nos computadores

A medida de tempo é necessária para qualquer tipo de computador, de forma que, desde os primeiros computadores, já existia algum mecanismo de medida de tempo. É importante considerar que embora o tempo seja contínuo no mundo real, ele é altamente segmentado e discretizado na computação digital [Levi e Agrawala, 1990]. Sistemas de navegação, por exemplo, integram duas vezes a aceleração ao longo do eixo do tempo, para obter a posição. Claramente, a discretização do tempo pode ter fortes efeitos nos resultados destas integrações. Dessa forma, como será discutido ao longo deste capítulo, é fundamental uma boa qualidade na medida do tempo para um sistema de tempo real [Wang e Lin, 1998].

Um dos primeiros mecanismos de medida de tempo nos computadores foi implementando no ENIAC em 1946. Este computador possuía um gerador de pulsos que podia ser direcionado para qualquer unidade da máquina [Grier, 2007a], como por exemplo, um acumulador para contar segundos. Outro relógio curioso foi o relógio dos computadores do sistema americano de defesa aérea, que possuía relógios construídos com tubos de vácuo, podendo contar até 34 minutos e 8 segundos, e voltando para zero em seqüencia [Grier, 2007a].

Percebe-se pela história, que os mecanismos para se medir e armazenar o tempo nos computadores nunca foram ideais, o que inclui o problema do "BUG do ano 2000", que custou cerca de 3 trilhões de dólares às empresas americanas [Grier, 2007a].

Talvez pelo fato da medida do tempo nos computadores também não ser ideal, com o passar do tempo

novos mecanismos foram sendo adicionados, fazendo com que os computadores tenham diversas formas de medir o tempo.

3.4.1 Gerador de pulsos

Um aspecto inerente ao projeto dos PCs, é que a maioria dos *timers* do PC executam com base em um oscilador construído a partir de um cristal pizoelétrico. Este cristal reage à eletricidade aplicada a seus pólos, gerando pulsos em uma freqüência predeterminada para alimentar os *timers* do computador. Entretanto, os cristais utilizados em placas mães de PC são de baixo custo, possuindo precisão de 100ppm (partes por milhão) e uma variação de +/- 200ppm por grau célsius podendo causar uma variação de 18ms por hora [Holden, 2004]. Além da variação de freqüência com a temperatura, o cristal ainda pode variar devido a outros fatores, como variação na tensão elétrica ou interferência eletromagnética.

A análise de Windl mostra que os erros na variação de freqüência podem causar um erro de 50ms por hora, e um erro de 1 segundo ou mais por dia, além do valor da freqüência variar em cerca de 11ppm pelo simples fato de se ligar o computador (provavelmente devido a variação na temperatura) [Windl et al., 2006].

Computadores industriais especialmente projetados para tarefas de tempo real, normalmente possuem fontes de freqüência e relógios mais precisos disponibilizados através de componentes dedicados a esta tarefa, com maior precisão do que aqueles tradicionalmente utilizados em PCs.

3.4.2 A medida de tempo via interrupções nos PCs

A forma mais utilizada para os RTOS medirem o tempo é através de interrupções periódicas provenientes de um *timer* [Kohout et al., 2003]. Este *timer* poderia ser o RTC que pode ser ajustado para gerar uma interrupção a cada 1ms aproximadamente. Entretanto, outras formas de gerar interrupções periódicas estão disponíveis nos PCs.

Uma interrupção de *hardware* consiste de em evento externo, normalmente na forma de um pulso gerado em um dos pinos do processador, que interrompe a execução da CPU para que alguma rotina associada à interrupção seja executada. Esta rotina chama-se rotina de tratamento de interrupção ou ISR (*Interrupt Service Routine).*

Dessa forma, usando o RTC como exemplo, a cada 1ms a CPU é avisada através de uma interrupção que 1ms se passou, e o *software* (neste caso, o sistema operacional) deve executar rotinas de atualização do relógio do sistema, para depois voltar a executar o que estava sendo executado anteriormente. A figura 2 mostra o diagrama de tempos e eventos para atualizar o *clock* do sistema através de interrupções provenientes de um *timer*. Percebe-se por esta figura, que o tempo entre o momento da interrupção de *clock* e o momento que o novo tempo é armazenado é perdido [Levi e Agrawala, 1990].

Além disso, sabe-se que os *timers* externos que geram as interrupções para o *clock tick*, e conseqüen-

temente para a hora do sistema, podem atrasar ou adiantar durante sua operação, levando a medidas incorretas de tempo [Kailas e Agrawala, 1997].

Como pode ser visto na figura 2, os principais fatores que influenciam no *clock* são:

1. Variação na freqüência dos pulsos gerados pelo *timer* devido a desvios na freqüência fornecida pelo cristal;
2. A latência das interrupções, ou seja, o intervalo de tempo entre o momento em que a interrupção ocorre, e o momento em que ela começa a ser tratada;
3. O tempo de execução da rotina de *software* responsável por tratar a interrupção (ISR), que neste caso atualiza o relógio do sistema.

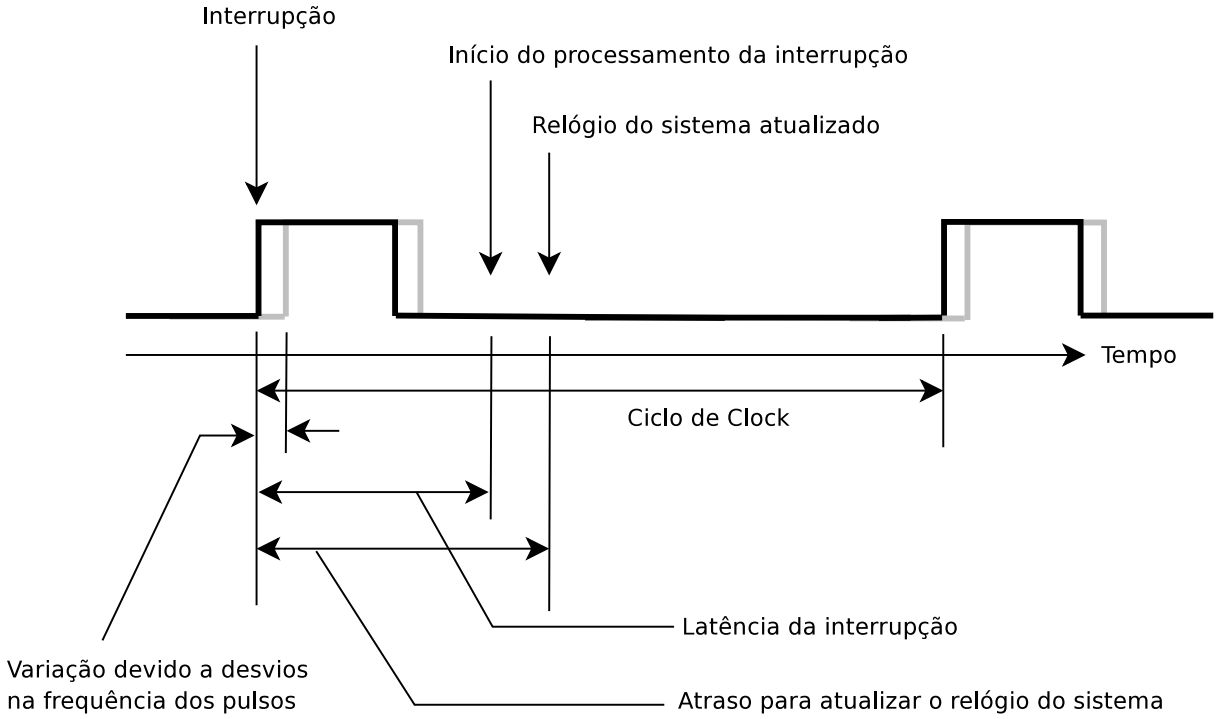


Figura 2: Diagrama da seqüência de atualização do *clock*. Adaptado de [Kailas e Agrawala, 1997]

A figura 3 mostra como a medida do tempo é dividida entre diversos componentes de um computador x86 neste tipo de relógio. Uma fonte de pulsos, normalmente um cristal, controla o *timer* que gera um estouro quando uma quantia de pulsos preprogramada é atingida. Neste estouro, uma interrupção é gerada, e a rotina de tratamento de interrupções atualiza uma região da memória com o relógio do sistema, que é baseado em *clock ticks* [Kailas e Agrawala, 1997].

O *clock tick* é uma variável especial que armazena o número de interrupções de *clock* desde que o computador foi ligado. Estes *ticks* podem ser gerados a diversas freqüências pelo *timer*. Um valor bastante comum de *clock tick* para sistemas operacionais fica entre 10ms e 100ms [Ganssle, 2004].

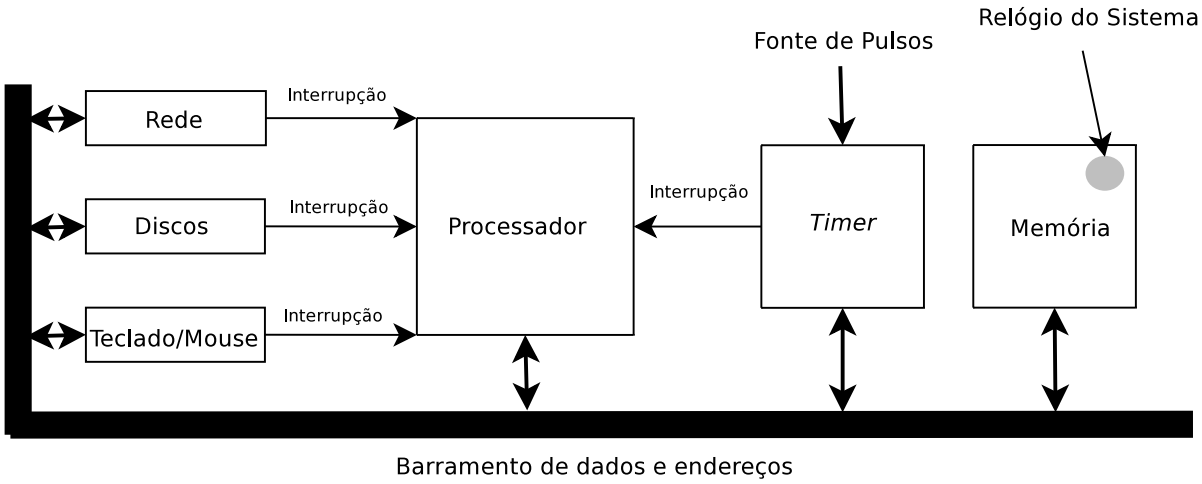


Figura 3: Interconexões que fazem o *clock* de um computador funcionar. Adaptado de [Kailas e Agrawala, 1997]

É importante dimensionar bem o tipo da variável que armazena o *clock tick* em memória, para evitar problemas como o que ocorreu nas primeiras versões do Windows 95 e Windows 98, onde o *clock tick* era registrado em um contador de 32 bits (indo de 0 a 4294967296). No caso do Windows, o *clock tick* ocorre a cada 55ms e após o sistema ficar exatamente 49,7 dias ligado (4294967 segundos) o Windows parava de responder completamente, pois não era mais possível incrementar este contador, levando o Windows a apresentar a conhecida tela azul de erro fatal. De acordo com o *NY Times*, a Microsoft levou 4 anos para descobrir a existência do problema [Gleick, 1999].

A figura 3 também demonstra que existe uma concorrência no processador entre as interrupções para medida do tempo e as de outros dispositivos, como da placa de rede, do controlador de discos e até mesmo do teclado e do *mouse*. Dessa forma, outras interrupções podem acabar causando atrasos no tratamento da interrupção de temporização. Outro fator que pode influenciar na medida do tempo neste sistema, é o fato de alguns sistemas operacionais desativarem as interrupções por curtos períodos de tempo como mecanismo de segurança para executar tarefas críticas. Esta característica dos sistemas operacionais será discutida com mais detalhes no capítulo 4.

A arquitetura para medir o tempo através de interrupções é limitada a uma granularidade da ordem de milisegundos, pois para cada atualização do *clock*, o sistema tem o *overhead* de interromper a tarefa em execução no sistema operacional, atualizar o relógio do sistema, e voltar a executar esta tarefa.

Entretanto, as interrupções de *clock tick* tem uma outra função extremamente importante. A cada quantia predeterminada de *clock ticks*, o sistema operacional executa diferentes tarefas internas para manter o bom funcionamento do sistema como um todo. Em alguns sistemas, por exemplo, a cada 10 *clock ticks*, o escalonador do sistema operacional é executado para verificar se alguma tarefa está

consumindo mais tempo de CPU do que o necessário, podendo tomar decisões de escalonamento, como terminar este processo, ou dar mais tempo para que ele tente terminar o que estava fazendo.

Em outras palavras, a qualidade do escalonamento depende diretamente do mecanismo de temporização do computador. O escalonamento é o processo através do qual o sistema operacional consegue executar vários programas de forma simultânea, dando fatias de tempo (*time slices*) a cada programa que precisa ser executado. O escalonamento dos sistemas operacionais será discutido em mais detalhes no capítulo 4.

Independente da origem dos *clock ticks*, é importante notar que o *clock tick* e a contagem do tempo nos computadores estão bastante relacionados com interrupções, podendo causar influências na contagem do tempo, como a latência, e o *jitter*[3].

Se um sistema estiver com um sensor quebrado, gerando interrupções espúrias, ou excessivas, dependendo da atribuição de prioridades das interrupções de hardware para o sistema operacional, a contagem de tempo poderia ser prejudicada, sendo imprecisa e em alguns casos até incorreta. Isto é reforçado pelo fato do *chip* controlador de interrupções mais usado em PCs (o 8259) não detectar interrupções perdidas. Caso o sistema operacional desative as interrupções, os *ticks* perdidos não serão detectados e nem recuperados [Holden, 2004].

O que minimiza o impacto deste possível problema é o fato dos PCs convencionais possuírem prioridades de interrupções fixas. Como a interrupção de *clock tick* normalmente é conectada diretamente a linha de requisição de interrupção 0 (IRQ0 - *Interrupt Request Line*), esta interrupção tem a maior prioridade do sistema, sendo tratada mesmo antes de interrupções externas que possam vir a ser mais importantes. A execução das rotinas de atualização do relógio devem ser executadas o mais rápido possível, pois sua latência contribui para o aumento da latência geral do sistema, tornando-o menos responsivo. De acordo com os especialistas da *Dedicated Systems,* isto é um dos motivos pelo qual arquitetura de *hardware* que tem a interrupção de *clock* como maior prioridade no sistema são a escolha menos comum para implementar sistemas de tempo real [Dedicated Systems, 2002e].

A discussão realizada nesta seção envolveu a contagem do tempo a partir de interrupções periódicas. As várias formas de se gerar interrupções periódicas no PC serão discutidas a seguir.

3.4.3 RTC (Relógio de Tempo Real)

Em geral, todos computadores possuem um relógio chamado RTC (*Real Time Clock* ou relógio de tempo real). É importante observar que embora este relógio tenha este nome, ele tem pouca relação com sistemas de tempo real. Ele é um relógio de baixo custo e relativa baixa precisão. O RTC possui uma bateria interna que é constantemente carregada e mantém um pequeno circuito que funciona de forma autônoma independente do computador estar ligado ou desligado, para manter a data e a hora

[3] Variação da latência, já que a latência não é sempre a mesma.

atualizadas.

O RTC tem um erro estimado de 1 segundo por dia [Windl et al., 2006], mas este erro varia com a temperatura e tensão. A implementação da leitura de dados do RTC é feita através de uma seqüência de chamadas as instruções de entrada e saída *inb* e *outb*, que são bastante lentas, levando de $1\mu s$ a $4\mu s$ para serem executadas a cada chamada. Maiores detalhes da temporização destas instruções serão discutidos no capítulo 7. Outro problema do acesso ao RTC é que as instruções *inb* e *outb*, além de lentas, também bloqueiam o processador enquanto estão sendo executadas, "paralisando" o sistema por alguns microsegundos.

Devido a estas limitações, o RTC é normalmente utilizado nos sistemas operacionais modernos apenas para ajustar a hora do sistema quando o computador é ligado. A partir deste momento, o relógio principal do sistema passa a ser um outro mecanismo de gerenciamento de tempo controlado pelo sistema operacional que é baseado em interrupções de *hardware*.

A interação do computador com o RTC depois da inicialização do sistema operacional é esporádica, em momentos em que o sistema operacional decide acertar a hora do RTC que freqüentemente pode ter desvios. Isto pode ocorrer a cada 11 minutos, como no *kernel* do Linux, ou apenas na hora em que o computador for desligado.

O RTC é um *chip* programável, podendo ser programado para fornecer pulsos de saída em um intervalo predeterminado, podendo ser dinamicamente direcionados para unidades do PC de forma a fornecerem uma entrada de pulsos para algum subsistema que necessite de contagem de tempo. Em geral, este recurso não é usado. Neste modo obtém-se a melhor resolução do RTC, onde ele pode gerar interrupções (na IRQ8) a cada $976\mu s$, ou seja, com uma resolução de aproximadamente 1ms [Sato e Asari, 2006].

A resolução e a qualidade do relógio do computador, dependem da fonte de *clock* utilizada. O RTC é uma destas fontes. A seguir, são descritos outras fontes de *clock*.

3.4.4 Temporizador Programável de Intervalos (PIT)

O temporizador programável de intervalos é muitas vezes chamado de PIT *(Programmable Interval Timer* ou de *Programmable Interrupt Timer*, dependendo do autor).

Este relógio possui uma qualidade relativamente baixa, com probabilidade de erro de vários segundos por dia e uma resolução de até aproximadamente 1ms. Ele existe até os dias atuais para manter compatibilidade e permitir que *softwares* mais antigos possam rodar nos PCs mais atuais. Sua origem remonta ao PC-XT.

Como a freqüência do PIT pode ser configurada para possuir intervalos menores, é possível obter maiores resoluções para sistemas onde a demanda de medida de tempo com maior precisão existe. Contudo, existe um ponto de equilíbrio entre a carga da CPU e a freqüência do PIT. Toda vez que o PIT gera uma interrupção, a CPU precisa parar o que estava fazendo, salvar o contexto da tarefa atual, e

executar a rotina de tratamento de interrupção para atualizar o relógio interno. Depois o contexto deve ser restaurado e a tarefa anterior voltar a ser executada. Quanto maior a freqüência do PIT, maior o número de vezes que o processador é interrompido para atualizar o tempo, degradando a performance de tarefas sendo executadas naquele momento.

O PIT, da mesma forma que o RTC, também é programado através de instruções *outb* e *inb*, tornando sua programação e ajustes muito lentas para o sistema operacional. As interrupções geradas pelo PIT normalmente são roteadas para a IRQ0 [Piccioni et al., 2001].

O PIT possui dois modos de operação. No modo aperiódico (também chamado de *One-Shot*) ele pode ser programado para gerar uma interrupção em algum momento específico, como por exemplo, daqui a 5ms ou daqui a 37ms, gerando uma única interrupção. No modo periódico, ele pode ser programado para gerar interrupções constantemente em períodos predefinidos. Este modo é freqüentemente utilizado para gerar o *clock tick* dos sistemas, e acionar os escalonadores, especialmente pelo fato do PIT precisar ser programado apenas na inicialização do sistema.

Alguns sistemas de tempo real, como o RTAI, podem utilizar o PIT no modo *One-Shot* para garantir maior precisão na execução de tarefas de tempo real. No caso do RTAI, o PIT é reprogramado toda vez que o laço de uma tarefa termina, dando precisão de milisegundos para o momento em que a tarefa vai voltar a ser executada.

O PIT é implementado no Circuito Integrado 8254, com base na sua versão anterior, o 8253. O 8253 era usado no PC-XT e o 8254 era usado nos PC-AT. Os PCs modernos não possuem mais este componente, mas todas as suas funções estão incorporadas no *chip* conhecido como *chipset*, que realiza o trabalho de vários componentes.

O PIT recebe uma onda quadrada de 1,19318 MHz em sua entrada, e possui 3 canais de saída. O canal 0 é o responsável por atualizar o *clock* do sistema. O canal 1 controla *refresh* da memória RAM, já que a memória DRAM utilizada nos PCs é barata, e suas células de memória devem ser periodicamente atualizadas para manterem os dados. Um terceiro canal (canal 2) é utilizado para gerar a freqüência que vai para o alto falante do PC (*PC Speaker*).

3.4.5 Controlador Programável de Interrupções Avançado (APIC)

O Controlador Programável de interrupções - *Advanced Programmable Interrupt Controller* (APIC) é uma evolução do controlador programável de interrupções (PIC) 8259, oferecendo mais linhas de interrupções e novos recursos. O APIC possui um *timer* interno, e pode gerar interrupções independente da necessidade de outros componentes e *timers*. O LAPIC (Local APIC) é incluído dentro da pastilha do processador, tornando o sistema mais integrado. A resolução do APIC é de boa qualidade, pois seu *timer* realiza a contagem com base no *clock* do barramento do processador [Intel, 2007].

O APIC não é muito usado pois só foi disponibilizado em processadores mais recentes, além de

existir situações em que o APIC pode parar de contar devido a estados de gerenciamento de energia do processador. Outro papel importante do APIC é para o uso em sistemas com vários processadores. Na verdade o APIC foi inserido na arquitetura dos PCs, para facilitar a implementação de sistemas multiprocessados, uma vez que vários LAPICs podem trocar informações entre si.

3.4.6 Interface avançada para configuração e alimentação (ACPI)

A interface avançada para configuração e alimentação - *Advanced Configuration and Power Interface* (ACPI) na verdade é um sistema criado para gerenciamento de energia nos PCs, permitindo desligar partes do PC que não estão em uso para economizar energia.

O ACPI possui um *timer* programável chamado de *PM Timer (Power Management Timer)*, que embora não tenha sido projetado para auxiliar sistemas operacionais, pode ser usado em algumas situações. No Linux, existe a opção de usar o ACPI como fonte do relógio do sistema, utilizando uma opção durante a inicialização do sistema operacional.

3.4.7 Contador de timestamp (TSC)

Como o gerenciamento do tempo é essencial em sistemas de tempo real, existe a possibilidade de integrar a medida do tempo diretamente dentro do processador [Kailas e Agrawala, 1997]. O TSC (*Time Stamp Counter*) consiste na contagem de ciclos do processador desde que o computador foi ligado. Isto é feito através de um registrador chamado TSC presente em processadores modernos a partir do Pentium. Utilizando instruções especiais (como o RDTSC), é possível ler o valor do TSC.

O TSC é um registrador de 64 bits, com resolução igual a 1/Hz, onde Hz é a freqüência de operação do processador [Piccioni et al., 2001]. Este registrador é incrementado automaticamente a cada ciclo de *clock*.

Dessa forma, um processador de 1GHz (1×10^9) possui a resolução de contagem de 1ns (1×10^{-9}). Como a velocidade de cada processador pode variar um pouco de acordo com sua fabricação, na hora da *boot*, o sistema operacional realiza um processo para verificar a velocidade do processador. Isto é feito ajustando o PIT para gerar uma interrupção depois de 50ms, por exemplo, e verificando quantos ciclos de *clock* passaram-se depois desta interrupção. Sabendo a velocidade do processador, o sistema poderá calcular o tempo em segundos [Chelf, 2001]. Após a calibração, o sistema pode contar o tempo com mais precisão sempre que necessário usando o TSC.

Com isto, caso algum problema de energia esteja ocorrendo durante o *boot* do computador, todas as medidas de tempo enquanto o sistema estiver ligado poderão estar erradas. Além disto, processadores que mudam sua velocidade para economizar energia automaticamente podem prejudicar esta contagem se um tratamento especial não for realizado [VMWare White Paper, 2005]. Isto é especialmente importante em processadores Intel com a tecnologia *SpeedStep*. Esta tecnologia varia a velocidade de funcionamento do

processador, de acordo com demandas instantâneas do computador, para reduzir o consumo de energia, o que é necessário hoje em dia com aplicações móveis e portáteis. Caso a velocidade do processador seja reduzida em 50% para economizar energia (com *SpeedStep*), e o tempo for baseado em TSC, a medida de tempo ficará incorreta.

Uma abordagem interessante discutida por Kailas em [Kailas e Agrawala, 1997], propõe o uso de vários temporizadores em conjunto para obter-se uma medida de tempo com acurácia de nano-segundos. Por exemplo, utilizar o PIT em conjunto com o TSC.

3.4.8 Temporizador de Eventos de alta precisão (HPET)

Finalmente, uma última forma de medir o tempo em busca de maior precisão foi criada. Trata-se do HPET (*High Precision Event Timer*), entretanto como ele consiste de um *hardware* novo e presente em poucas arquiteturas, até recentemente praticamente nenhum sistema operacional utilizava seus recursos [VMWare White Paper, 2005].

Sua origem deu-se em 2000 com especificação do *Multimedia Timer* que teve o nome mais recentemente alterando para HPET. Tal *timer* só começou a estar disponível em máquinas comerciais em meados de 2004. Suas funções incluem sincronização para maior qualidade de vídeos de tempo real (s*oft real time*), suporte ao escalonamento do sistema operacional, além de um *timer* de referência para sincronia de sistemas com vários processadores [Intel, 2004].

Espera-se que os *timers* do HPET tenham em um período de 1ms um erro de 0,05% (500ppm). Para períodos de 100μs o *clock* pode ter um erro de mais ou menos 200ns, representando um erro menor que 0,2% (2000ppm) [Intel, 2004].

A resolução mínima do HPET é de 0,1μs (100nS) [Schmidt, 2007], tornando-se a escolha certa para novas aplicações que precisam de *timers* com alta precisão. Internamente, o HPET tem *ticks* da ordem de femto-segundos (10^{-15} segundos). Para se ter uma idéia de sua dimensão, um femto-segundo é para um segundo o que um segundo é para 32 milhões de anos.

A decisão de se construir o HPET surgiu mais especificamente para atender demandas multimídia. O HPET foi feito para substituir totalmente o PIT e a função de gerar interrupções periódicas do RTC. O HPET pode ter até 32 *timers*. Atualmente, apenas o Windows Vista e novas versões do Linux vem usando o HPET.

3.4.9 Sincronização do tempo via rede

Embora a maioria dos sistemas embarcados não estejam diretamente conectados a Internet ou a uma rede, para aqueles que estão, existe a possibilidade de utilizar-se uma fonte bastante precisa de tempo via rede chamada de NTP (*Network Time Protocol).* Os servidores de NTP normalmente utilizam várias fontes de alta precisão, como relógios atômicos e receptores de GPS (*Global Positioning System*) para ter

um horário bastante preciso.

Como a Internet possui latências completamente indeterminadas e não previsíveis, o tempo de sincronização poderia causar um erro na sincronia dos relógios [Selig, 2006]. Por exemplo, uma requisição de sincronia é enviada a um servidor NTP conectado a um relógio atômico, e o servidor responde que a hora atual é 22:34:13. Esta resposta vai trafegar pela Internet, e chegar ao PC que fez a requisição, mas por questões de atrasos indeterminados e imprevisíveis da Internet, a informação pode demorar vários segundos para chegar. Se ela demorar por exemplo 10 segundos, a atual hora será 22:34:23 mas será ajustada como 22:34:13.

O NTP possui ferramentas gratuitas com complexos algoritmos que medem a latência e variação de latência da rede até o servidor de data, e realizam uma grande troca de informações para medir a latência da rede. Dessa forma, a latência pode ser adicionada a hora recebida do servidor para que a hora ajustada seja a mais precisa possível. O NTP poderia atingir resoluções de 1ns quando utiliza-se *hardware* externo, como um GPS, para manter a hora. A simples sincronia do relógio de um PC via rede oferece resoluções de 1ms a 10ms. Para tanto, um *software* deve estar em constante execução, tanto no servidor de hora, quanto no cliente sincronizando o tempo. Uma discussão mais completa sobre NTP pode ser consultada em [Windl et al., 2006].

3.4.10 Resolução do relógio

Um parâmetro importante para sistemas de tempo real, é a resolução do relógio do sistema, ou seja, o menor incremento de tempo que pode ser adicionado ao relógio para contar o tempo. A qualidade e resolução do relógio dependem da origem das interrupções de *clock* utilizadas pelo sistema operacional.

Nos computadores da arquitetura Alpha da DEC, a interrupção de *clock tick* ocorre a cada 976,5625 microsegundos, dando uma resolução de aproximadamente 976μs. Se uma aplicação precisa de um *timer* ou uma medida de tempo que é um múltiplo da resolução do relógio (976,5625), esta medida será precisa. Caso contrário um pequeno erro irá ocorrer, como por exemplo, ao ajustar um *timer* periódico que gera eventos a cada 50ms, porém na prática, este *timer* irá na verdade gerar eventos a cada 50,78ms [Laplante, 2004].

Da mesma forma, isto ocorre com os sistemas operacionais de propósito geral que têm interrupções geradas a cada 10ms. Se um programador solicitar um *sleep* de 15ms, ele irá conseqüentemente durar 20ms. Em sistemas operacionais de tempo real, normalmente um relógio de hardware é programado para disparar uma interrupção daqui a 15ms, para garantir a precisão [de Oliveira, 2003]. No caso do Windows 98, a função *sleep* com parâmetro de 10ms não iria durar 10ms, mas sim 55ms, já que a granularidade deste sistema é de 55ms.

Nem sempre um sistema operacional deve ser usado em um sistema de tempo real. Muitas vezes, o requisito de resolução temporal é tão crítico, que é necessário utilizar um *hardware* analógico especialmente

projetado para tal requisito. Em outros casos, pode ser preciso escrever um código em linguagem de máquina *(assembly)* específico para determinada situação. A figura 4 mostra a hierarquia de tempos para ser usada em sistemas de tempo real.

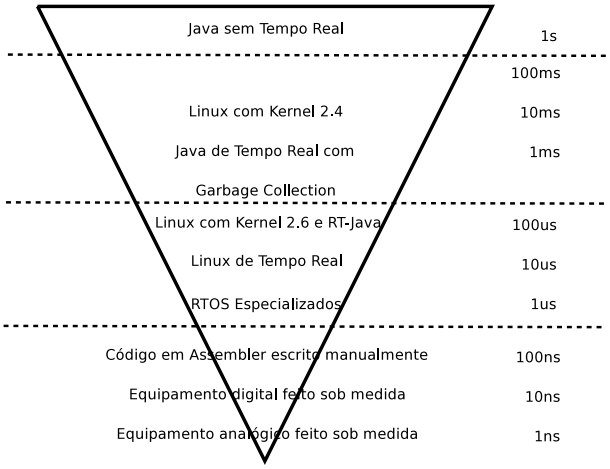


Figura 4: Triângulo invertido de capacidades de tempo real. Adaptado de [McKenney, 2007]

3.5 Tempo real na Mecatrônica

O tempo real para sistemas mecatrônicos não requer demandas extremamente restritas de tempo. Um bom exemplo está na área espacial, onde o ciclo de controle é executado a 25Hz. Os orbitadores da NASA usam um sistema operacional chamado FCOS (*Fligth Computer Operating System*) que executa sua tarefa mais importante de controle a 25Hz (a cada 40ms) em computadores IBM AP-101 redundantes, cada um com cerca de 420Kb de memória e velocidade de 450KHz, além do *software* do sistema todo não ultrapassar 500Kb [Carlow, 1984]. O FCOS usa cerca de 35% da memória do computador de propósito geral AP-101 [Spector e Gifford, 1984].

Como discutido na introdução, robôs industriais operam com laços de controle PID em geral de 10ms, e 1ms quando utiliza-se controle de força. Esta informação foi reforçada pela OMAC e GM.

Neste capítulo foram discutidas características do tempo e suas medidas, com foco em como os computadores modernos realizam a medida do tempo. Estas informações serão úteis ao longo deste trabalho para uma melhor compreensão dos sistemas operacionais de tempo real. No próximo capítulo serão discutidos aspectos construtivos dos sistemas operacionais, e em seguida uma análise experimental comparando vários sistemas será apresentada.

4 Sistemas Operacionais

4.1 Breve história

Os primeiros computadores não possuíam sistemas operacionais, de forma que era necessário que um operador carregasse manualmente na memória do computador cada programa a ser executado, além de fornecer os dados de entrada, anotar o resultado do processamento, e carregar o próximo programa. Este tipo de operação era demorada, repetitiva e poderia levar a noite toda. No início da década de 60 começaram a surgir os primeiros sistemas operacionais, que inicialmente eram capazes apenas de executar uma seqüência de programas, evitando a necessidade do operador carregar os programas manualmente.

A medida que a capacidade dos computadores foi aumentando, os sistemas operacionais acompanharam este crescimento, e no final da década de 60 foi lançada a primeira versão do Unix. O sistema era tão avançado que seus conceitos ainda são utilizados até hoje em sistemas operacionais. Em 1970 o Unix possibilitava que um computador executasse várias tarefas ao mesmo tempo, dividindo o tempo do processador em pequenas fatias e oferecendo algumas fatias para cada programa. O Unix também já possuía proteção de memória, para evitar que um programa mal elaborado causasse uma pane no sistema todo. Também oferecia o recurso de multiusuário, onde diversas pessoas podiam usar o mesmo computador ao mesmo tempo através de terminais remotos. Ken Thompson e Dennis Ritchie, os autores do Unix, possuíam uma visão tão ampla da computação, que além do Unix, também criaram a linguagem C, para facilitar a portabilidade de programas entre diversos hardwares e arquiteturas, permitindo que um único código pudesse ser compilado e executado sem alterações em diferentes tipos de computadores.

Outro grande evento ocorreu na década de 60 quando a missão Apollo conseguiu ir com sucesso para a Lua e voltar para o planeta Terra. Alguns consideram que o computador de bordo da nave foi o primeiro sistema embarcado de tempo real moderno construído. O computador possuía 8Kb de memória RAM (*Random Access Memory*) e 64Kb de memória ROM (*Read Only Memory*). Apenas para acessar estas memórias o processador levava $12\mu s$. Mesmo com estas limitações, este computador possuía um sistema operacional de tempo real desenvolvido pelo MIT, que além de dividir o processador entre as várias tarefas críticas, também tratava interrupções e gerenciava a memória, já que a memória era insuficiente para alocar todos os programas de forma simultânea.

4.2 Considerações de hardware

Antes de discutir os sistemas operacionais, será realizada uma breve apresentação de componentes e características fundamentais do *hardware* e da arquitetura dos microcomputadores PC que permitem que um sistema operacional funcione. A arquitetura de interligação de componentes e periféricos em um computador pode influir no funcionamento e na qualidade do sistema operacional. Um diagrama geral da arquitetura de um computador compatível com a arquitetura PC pode ser visto na figura 5. Este

diagrama será usado ao longo da discussão realizada neste capítulo, sendo que a seguir serão apresentados alguns aspectos presentes neste diagrama que podem influenciar em sistemas de tempo real.

4.2.1 Acesso Direto a Memória (DMA)

O acesso direto a memória, ou *Direct Memory Access (DMA)* é um recurso que permite que periféricos de um computador troquem dados diretamente com a memória sem a necessidade dos dados passarem pelo processador. Usualmente, qualquer operação que envolva cópia de dados necessita que o processador acesse a informação em um determinado local. Dessa forma, operações simples, como a cópia de um CD ou a aquisição de 500.000 amostras de um conversor analógico-digital roubariam um tempo precioso do processador.

O DMA permite que blocos de dados de um CD sejam copiados diretamente para a memória, sendo que o processador só é avisado através de uma interrupção quando a cópia do bloco terminou. No caso de dispositivos de conversão analógico para digital, o processador só seria avisado depois das 500.000 amostras serem feitas e copiadas para a memória.

O problema em sistemas de tempo real, está no fato do barramento para troca de dados entre os dispositivos ser único. Dessa forma, enquanto dados estão sendo copiados para a memória, o processador pode fazer cálculos, mas suas operações são limitadas, pois o processador não pode usar aquele barramento para obter dados ou controlar portas de entradas e saída.

Quando deseja-se pensar no pior caso de execução para um programa de tempo real, o roubo de ciclos executado pelo DMA deve ser considerado como ocorrendo a cada oportunidade, aumentando bastante os tempos de busca de instruções.

4.2.2 Memórias Cache

As memórias *cache* são memórias de alta velocidade, normalmente localizadas dentro do processador, de forma que seus tempos de acesso são extremamente baixos (da ordem de nano-segundos). Como estas memórias são caras, é usual os computadores possuírem apenas alguns megabytes deste tipo de memória. Ela funciona guardando dados que são mais freqüentemente utilizados pelo processador. Através de algoritmos especiais, que levam em consideração, por exemplo, dados mais freqüentemente utilizados pelo processador, ou mais recentemente usados, a memória *cache* mantém uma cópia destes dados mais utilizados que estão na memória RAM. Dessa forma, o processador economiza tempo acessando estes dados por não precisar recorrer ao barramento de acesso a memória RAM.

Quando o processador tenta acessar um dado que já está na *cache*, evitando o acesso a RAM, ocorre um evento chamado de CACHE HIT. Caso o dado não esteja na *cache*, e seja necessário recorrer a memória RAM, ocorre um CACHE MISS.

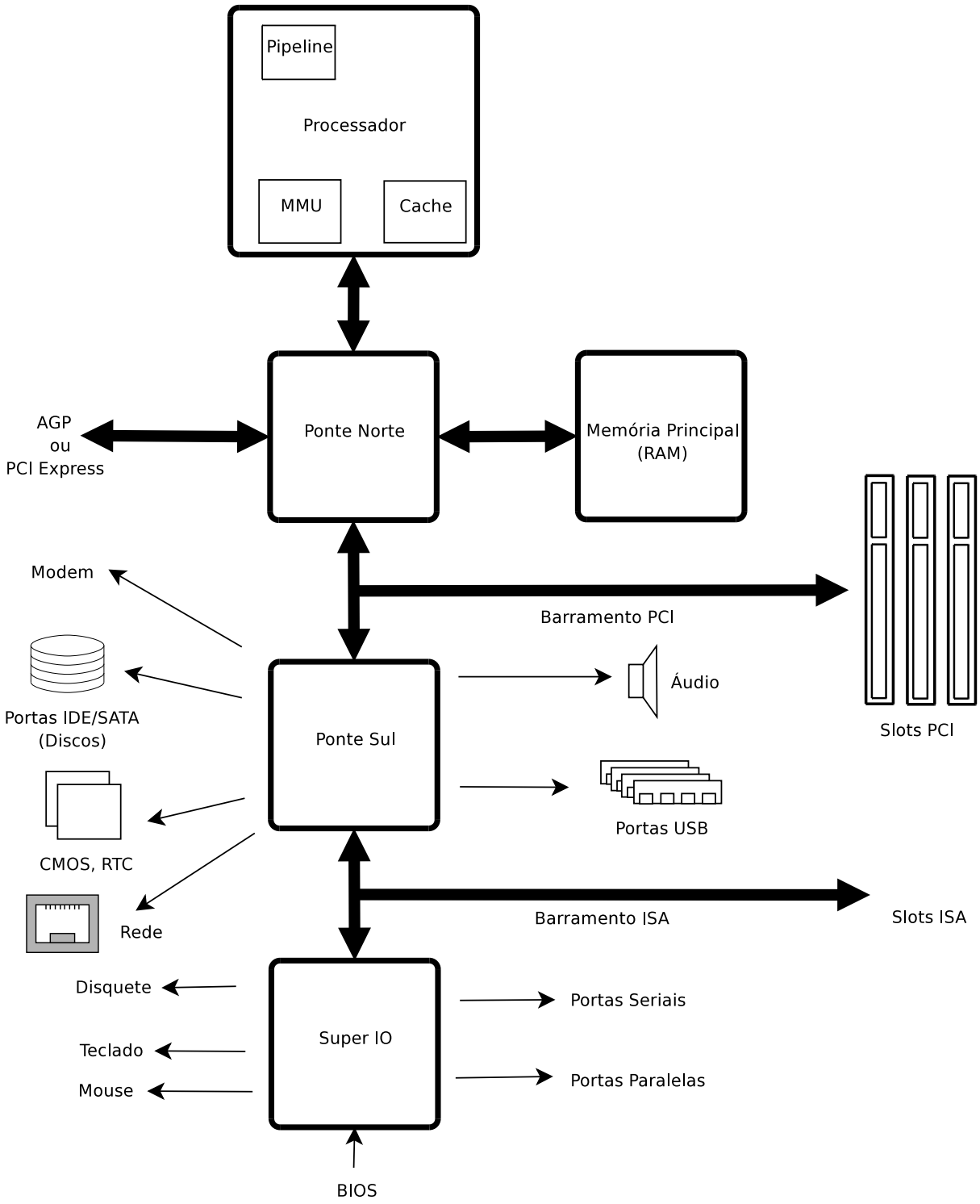


Figura 5: Arquitetura interna de um PC

Sabe-se que a memória *cache* reduz o tempo de acesso a memória principal de forma impressionante. Em um exemplo, o tempo de acesso à memória *cache* é de 30ns e à memória principal é de 100ns. Isto significa que se o *cache hit* for de 60%, o tempo médio de acesso à memória torna-se 58ns. Em contrapartida, é muito difícil predizer de forma confiável a performance da memória *cache* para uma tarefa. Dessa forma, o uso de *cache* em sistemas de tempo real é desencorajado [Basumallick e Nilsen, 1994].

O problema é que as memórias *cache* são muito pequenas, contendo apenas os dados mais usados pelo processador em um determinado momento. Dessa forma, é necessário constantemente fazer trocas do conteúdo da memória *cache* com a memória principal, para que a *cache* sempre tenha os dados que o processador mais utiliza. Uma abordagem segura para prever o tempo na troca do conteúdo de *cache*, é supor o pior caso, ou seja, considerar que sempre que ocorrer uma troca de contexto, a *cache* toda deverá ser recarregada. Embora seguro, em muitos casos isto não irá acontecer, levando a estimativas que subutilizam o processador [Sebek, 2001].

Tratando-se de processadores de uso genérico, quanto mais rápido o processador, maior o efeito da memória *cache*, e mais difícil é executar uma tarefa de tempo real [Proctor e Shackleford, 2001]. Como memórias *cache* introduzem comportamentos temporais que são difíceis de prever, muitos sistemas de tempo real desabilitam a memória *cache* [Vera et al., 2003], diminuindo muito o poder de processamento do sistema, tendo em vista que os processadores são muitos mais rápidos que as memórias, e a *cache* preenche esta lacuna.

Num cenário de pior caso, deve-se sempre levar em conta que as instruções não estão na *cache*. Em um estudo experimental realizado por Weiss [Weiss et al., 1999], a *cache* aumentou a performance do sistema analisado em 40%, entretanto a conclusão obtida foi que a memória *cache* não deve ser usada em sistemas de tempo real *hard* que necessitam de determinismo.

4.2.3 Pipelines

Os processadores dos PCs utilizam uma arquitetura interna chamada de arquitetura de Von Neumann, que normalmente tem 4 passos:

1. Obter instrução a ser executada na memória
2. Decodificar a instrução
3. Executar a instrução
4. Gravar o resultado na memória

O que todo processador faz é ficar executando este ciclo infinitamente de instrução em instrução. Esta seqüência pode ser vista na figura 6.

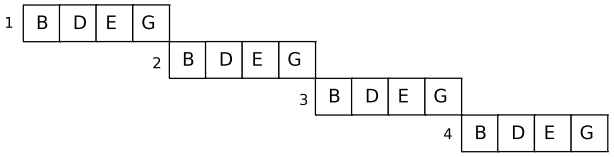


Figura 6: Execução de 4 instruções sem o uso de *pipeline*. Legenda: B: Busca D: Decodificação E: Execução G: Gravação

Um mecanismo criado para aumentar o desempenho chama-se *pipeline*. O *pipeline* consiste em paralelizar estas 4 operações. Enquanto uma unidade do processador está executando uma instrução, outra unidade já está decodificando a próxima, e outra unidade já está buscando a instrução seguinte. Enquanto 4 instruções seriam executadas na forma tradicional, 13 instruções poderiam ser executadas com o paralelismo via *pipeline*, como pode ser visto na figura 7.

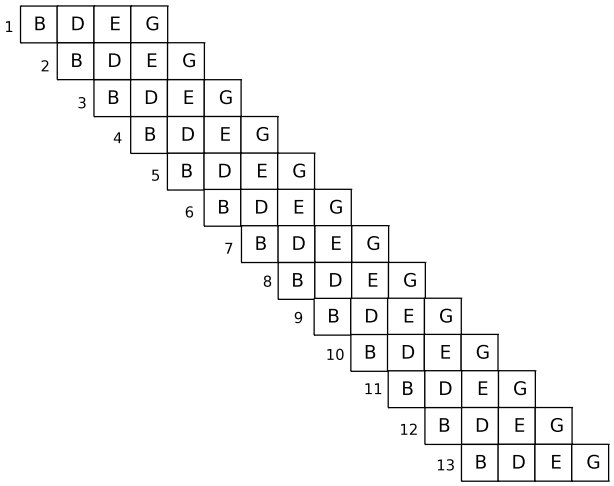


Figura 7: Execução com o uso de *pipeline*. Legenda: B: Busca D: Decodificação E: Execução G: Gravação

Assim como a *cache*, os *pipelines* aumentam a velocidade média do sistema, mas deteriora o determinismo, pois o tempo de execução de instruções pode não ser sempre fixo. Isto ocorre, pois o processador alimenta o *pipeline* pegando uma seqüência continua de instruções da memória. Muitas vezes, uma destas instruções pode ter saltos incondicionais ou saltos condicionais. No caso de um salto, todo *pipeline* é perdido, de forma que o *pipeline* necessita começar a ser preenchido novamente. No caso dos *pipelines*, também deve-se considerar o pior caso, como se o *pipeline* estivesse vazio.

Arquiteturas com *cache* e *pipelines* criam situações onde um mesmo código é executado em diferentes intervalos de tempo [Sohal, 2001], já que nunca se sabe quando o *pipeline* será descartado devido a um salto condicional, ou quando o conteúdo da memória *cache* será utilizado ou não devido ao histórico de uso e alocação das tarefas do sistema.

4.2.4 Unidade de Gerenciamento de Memória (MMU)

A unidade de gerenciamento de memória ou *Memory Management Unit (MMU)* é uma unidade que nos primeiros projetos de computadores consistia de um componente externo ao processador, como no caso do Motorola 68020 que usava o *chip* MC68852 como MMU. A partir do AT80286 da Intel, a MMU já vinha integrada ao próprio *chip* do processador, como pode ser visto na figura 5.

A MMU desempenha dois papéis importantes: o mapeamento de memória virtual para memória real, e a proteção da memória. A proteção é um recurso importante para aumentar a segurança em sistemas operacionais de tempo real.

Toda vez que o sistema operacional pára a execução de uma tarefa no processador, e inicia a execução de outra tarefa, registradores especiais relacionados a MMU são ajustados. Eles especificam o intervalo de posições de memória que o programa a ser executado pode acessar.

Caso o programa/tarefa em execução tente acessar uma região de memória fora de sua área permitida, a MMU irá detectar esta violação e causar uma interrupção, que vai fazer o processador parar a execução do programa, e voltar a executar o sistema operacional, que poderá tomar a decisão do que deve ser feito com a tarefa que ocasionou o erro.

Sem a MMU, toda a memória do computador pode ser acessada por qualquer programa a qualquer momento, de forma que falhas em uma tarefa podem acabar corrompendo outras tarefas totalmente independentes.

4.2.5 Ponte Norte

A ponte norte também é freqüentemente chamada de *chipset* da placa mãe, nome popular para a placa eletrônica base onde são conectados todos componentes de um PC. Seu papel é fundamental nas arquiteturas modernas de PCs, já que ela é o componente que conecta o processador e a memória principal do computador. Em um processador Pentium 4 a velocidade de comunicação entre a ponte norte e o processador pode chegar a 3,2Gbits por segundo.

As recentes demandas de processamento de imagens, especialmente para jogos e aplicações gráficas 3D, criaram uma demanda de dispositivos de vídeo de alta velocidade. A solução encontrada pela indústria foi ligar placas de vídeo diretamente à ponte norte. Este tipo de conexão é encontrada no barramento da Porta Gráfica Acelerada - AGP (*Accelerated Graphics Port*) e mais recentemente no *PCI-Express.* O AGP pode atingir velocidades de pouco mais de 2Gb/s enquanto o *PCI-Express* pode ultrapassar 4Gb/s.

Em aplicações de tempo real, não é recomendado utilizar aceleração gráfica ou placas gráficas AGP ou *PCI-Express,* já que este tipo de porta gráfica demanda grande largura de banda da ponte norte, causando latências e perda de determinismo na arquitetura x86.

Existem muitas variações possíveis de configuração e tipos de ponte norte. Aliás, este nome se dá em decorrência deste componente normalmente ficar na parte superior das placas-mãe de PCs, próximas ao

processador. Em produtos recentes da AMD, partes da ponte norte foram incorporadas ao processador. Também existem situações, onde tanto a ponte norte quanto a ponte sul são integradas em um único *chip*.

Como as pontes são bastante acopladas aos processadores, é comum os fabricantes lançarem junto com seus processadores, pontes de referência, com especificação aberta, para que outros fabricantes possam implementar suas pontes.

4.2.6 Ponte Sul

A ponte sul é conectada a ponte norte e é responsável por gerenciar dispositivos de velocidade relativamente baixa, como portas USB (*Universal Serial Bus)*, unidades ópticas (CDs e DVDs), discos rígidos, entrada e saída de som, dentre outros. A ponte sul e suas conexões podem ser vistas na figura 5. Em algumas versões, esta ponte também pode controlar rede e modem. Um dispositivo importante já discutido, o RTC (*Real Time Clock)* normalmente é conectado a esta ponte.

A ponte sul também pode disponibilizar um barramento ISA (*Industry Standard Architecture*) que hoje em dia não é mais comum em computadores. Quando este barramento é disponibilizado, normalmente isto é feito por questões de compatibilidade com dispositivos legados, já que o barramento ISA é bastante lento. Uma outra ponte, que pode ser integrada no mesmo *chip* da ponte sul, chamada de *Super IO* oferece acesso a dispositivos importantes para automação, como portas seriais e paralela.

Dessa forma, um acesso a porta serial ou paralela deve sair do processador, passar pela ponte norte, em seguida pela ponte sul, depois pelo dispositivo *Super IO* para finalmente atingir a porta de destino. Este caminho pode causar influências negativas e tempos diferentes para cada acesso a estas portas.

4.2.7 Barramento ISA e LPC

Para poder eliminar o barramento ISA das novas arquiteturas, mantendo compatibilidade, a Intel criou o barramento LPC (*Low Pin Count*) que apresenta as mesmas características e endereços do padrão ISA para o *software*, entretanto conexões físicas diferentes. Isto possibilita a execução de *software* legado em computadores modernos sem o barramento ISA. O *chip* IT8705F, que implementa este barramento, é usado atualmente nas placas mãe para controlar dispositivos de baixa velocidade, como disquetes, teclado, dentre outros [Intel, 2002].

4.2.8 Gerenciamento de Energia e do Sistema

Como discutido na seção 3.4.7, a velocidade do processador pode ser alterada para economizar energia, afetando contadores de tempo como o TSC.

Outro subsistema que tem impacto negativo em sistemas de tempo real é um modo de operação dos processadores modernos chamado de Modo de Gerenciamento do Sistema ou SMM (*System Management*

Mode). O SMM é responsável por gerenciar aspectos importantes do sistema, como ligar/desligar um ventilador de resfriamento, ou aumentar/diminuir a velocidade destes ventiladores. O SMM também é capaz de desligar automaticamente um computador caso o processador fique em uma alta temperatura por muito tempo, havendo risco de queimar. O SMM também trata erros de memória ou das pontes norte ou sul.

O grande problema do SMM é que ele interrompe a execução de tudo que está sendo realizado para ser executado, incluindo o próprio sistema operacional. Dessa forma, *clock ticks* e interrupções são perdidas enquanto o SMM está em operação.

Em um experimento de medidas de tempo, foi notado que em intervalos irregulares, variando entre 300 e 400 segundos, os dados medidos tinham um pico de 40 a 50 microsegundos. Ao investigar o problema, concluiu-se que era o SMM em operação [Windl et al., 2006].

O SMM pode ser acionado por interrupções chamadas de SMI (*System Management Interrupt*). Como o SMM foi projetado para não precisar ser desabilitado, não existem opções na configuração dos computadores, como na BIOS (*Basic Input Output Sytem*), por exemplo, para desativar este recurso. Em algumas pontes é possível desabilitar o SMM e o SMI escrevendo uma seqüência de números em registradores específicos. Contudo, é preciso tomar muito cuidado ao desabilitar o SMM, pois alguns mecanismos de auto proteção do sistema estarão desativados.

O SMI é considerado um dos vilões que dificultam atingir um bom determinismo e performance de tempo real em PCs. Para amenizar o problema, o RTAI possui a opção de desativar o SMI ao ser carregado no computador.

4.2.9 Interrupções

Embora a noção de interrupções seja um conceito básico na área de computação, sistemas de tempo real podem utilizar ou não interrupções. Como a maioria dos sistemas as utiliza, é interessante analisar alguns aspectos da natureza das interrupções nos PCs.

As interrupções são bastante úteis quando o processador está esperando por dados. Por exemplo, se não houvesse uma interrupção para avisar que dados chegaram pela rede, o processador precisaria ficar lendo constantemente registradores da placa de rede para verificar se algum dado novo chegou. Este processo é chamado de *polling.*

Uma interrupção é normalmente um pulso elétrico aplicado a um pino especial do processador responsável por receber interrupções. Ao receber este pulso, o processador automaticamente interrompe a operação atualmente em execução, e verifica uma região da memória chamada vetor de interrupções. Este vetor possui os endereços de onde estão as rotinas que devem ser executadas para cada tipo de interrupção. A rotina responsável por tratar uma interrupção é chamada de Rotina de Tratamento de Interrupção ou ISR (*Interrupt Service Routine).*

Também existe uma categoria especial de interrupções que podem ser geradas por *software*, através de instruções especiais, ou como resultado de erros ou falhas em operações sendo executadas. O PC possui um vetor de interrupções que suporta 256 interrupções diferentes. Normalmente 16 são de *hardware*, e as demais de *software*, ou excessão. A tabela 2 mostra interrupções de um processador da arquitetura Intel x86 (PC).

As interrupções de 0 a 7 (com excessão da 2) são normalmente geradas pelo processador ao detectar falhas, chamadas de excessões. Estas interrupções são bastante importantes para manutenção de um sistema estável, já que erros em um *software,* como uma divisão por zero gera uma interrupção, interrompendo o programa que efetuou a operação ilegal e passando o controle para a rotina especificada no vetor de interrupções (normalmente o sistema operacional).

É comum existir uma confusão entre os termos IRQ e INT. O termo INT é diretamente uma abreviação para interrupção e refere-se a qualquer tipo de interrupção que possa ocorrer. IRQ é uma abreviação para requisição de interrupção (*Interrupt Request*). As IRQs são associadas a eventos externos causados pelo *hardware*, como o pulso do relógio de tempo real (RTC), por exemplo. A tabela 2 mostra a associação entre as INTs e IRQs. Normalmente, o núcleo do processador em si tem apenas 2 entradas de interrupção, sendo uma delas a entrada de interrupção não mascaravel - NMI (*Non Maskable Interrupt)* que é detectada como INT2.

Para conectar as várias interrupções existentes, o processador utiliza um controlador programável de interrupções - PIC (*Programmable Interrupt Controller)*, que recebe várias interrupções, e as repassa para o processador. Na arquitetura original do PC, era utilizado apenas um PIC com 8 entradas, mas posteriormente foi adicionado um outro PIC em cascata [Reis, 1992], para que fosse possível se ter 16 IRQs. Atualmente, o PIC faz parte do processador ou de uma das pontes.

Para evitar qualquer mal entendido, deve-se saber que o controlador programável de interrupções, apesar de ser chamado PIC, não tem absolutamente nenhuma relação com microcontroladores PIC. Normalmente utiliza-se o *chip* 8259A para o controlador de interrupções.

Pela tabela 2, também pode-se ver a ordem de prioridade das interrupções. Nos PCs, as prioridades das interrupções são fixas de acordo com seus números. Dessa forma, a IRQ de maior prioridade no sistema é a do *timer* do sistema. Caso uma interrupção de teclado ocorra porque um usuário apertou uma tecla ao mesmo tempo que uma de *timer,* o sistema irá tratar primeiro a de *timer* para depois tratar a de teclado. É importante notar que a IRQ2 do primeiro PIC é a entrada cascateada do segundo PIC, de forma que todas interrupções ligadas ao PIC 2 têm maior prioridade que as interrupções 3 até 7 do primeiro PIC.

As IRQs podem ser habilitadas ou desabilitadas a todo momento escrevendo informações de configuração nos registradores do(s) PIC(s). Esta programação é realizada em um registrador chamado de IMR *(Interrupt Mask Register).* Quando uma interrupção é mascarada, o PIC não avisa o processador que ela

Entrada do PIC	PIC	Interrupção no processador	Dispositivo/Evento
N/A	N/A	INT 0	Divisão por zero
N/A	N/A	INT 1	Depuração: próximo passo
N/A	N/A	INT 2	NMI
N/A	N/A	INT 3	Depuração: *breakpoint*
N/A	N/A	INT 4	Estouro matemático
N/A	N/A	INT 5	BIOS *Print Screen*
N/A	N/A	INT 6	Reservado
N/A	N/A	INT 7	Reservado
IRQ 0	1	INT 8 (0x8)	Timer do sistema (exclusiva)
IRQ 1	1	INT 9 (0x9)	Teclado (exclusiva)
IRQ 2	1	INT 10 (0xA)	PIC 2 (cascateamento)
IRQ 3	1	INT 11 (0xB)	Porta Serial 2 / Livre
IRQ 4	1	INT 12 (0xC)	Porta Serial 1 / Livre
IRQ 5	1	INT 13 (0xD)	Porta paralela 2 e/ou PS/2 / Placa de Som
IRQ 6	1	INT 14 (0xE)	Controlador de disquete
IRQ 7	1	INT 15 (0xF)	Porta paralela 1 / Livre
-	-	(0x10 - 0x6F)	Interrupções de Software
IRQ 8/0	2	(0x70)	Real Time Clock (RTC) (exclusiva)
IRQ 9/1	2	(0x71)	Livre
IRQ 10/2	2	(0x72)	Livre
IRQ 11/3	2	(0x73)	Livre
IRQ 12/4	2	(0x74)	Mouse PS/2 / Livre
IRQ 13/5	2	(0x75)	Unidade de Ponto Flutuante (exclusiva)
IRQ 14/6	2	(0x76)	Controlador de Disco Rígido IDE / Livre
IRQ 15/7	2	(0x77)	Controlador de Disco Rígido IDE secundário
-	-	(0x78 - 0xFF)	Interrupções de Software

Tabela 2: Interrupções em um PC. Adaptado de [Hyde, 2003]

ocorreu, de forma que o sistema nunca saberá da ocorrência desta interrupção específica.

A excessão às IRQs são as NMIs que nunca podem ser mascaradas, sendo necessário sempre realizar algum tratamento. Exemplos de NMIs são erros de paridade, erros na memória, ou erros graves em algum subsistema do hardware.

Após tratar uma interrupção, o programa do usuário deve enviar um código para o PIC chamado de EOI *(End Of Interrupt)*. Enquanto isto não for feito, outras interrupções não podem ocorrer. Normalmente o próprio sistema operacional responsabiliza-se por enviar os EOIs aos controladores de interrupções.

Como as interrupções podem ocorrer a qualquer momento, elas podem interromper a execução de tarefas importantes e até prejudicar a sincronia entre processos críticos em execução. Em muitos casos, é comum desabilitar as interrupções em momentos críticos para que uma tarefa seja executa por completo, de maneira atômica. Isto pode aumentar consideravelmente a **latência das interrupções**, pois novas interrupções só serão detectadas quando as interrupções estiverem habilitadas novamente. Por isto é recomendável que os ISRs sejam sempre o mais rápido e simples dentro do possível.

De acordo com Laplante, a latência de interrupções é o tempo entre o momento que a interrupção ocorre, e o momento que a primeira instrução associada à rotina de tratamento de interrupção é executada

[Laplante, 2004]. Esta informação é um dos parâmetros mais importantes dos sistemas operacionais de tempo real [Sohal, 2001] e vários de seus aspectos serão discutidos ao longo deste trabalho.

Na maioria dos casos, a forma como os programas são feitos é responsável por 99% da latência das interrupções, e não o que a CPU está fazendo em um determinado momento. Como isto é difícil de ser previsto na hora do projeto e desenvolvimento de um sistema, somado a falta de métodos formais, são necessárias técnicas empíricas para gerenciar a latência [Ganssle, 2004]. Dessa forma, a latência de tratamento de interrupções depende da implementação de cada sistema operacional. Se a técnica de desabilitar interrupções para manter sincronização é usada muito pelo SO, as latências aumentam.

Outro aspecto importante das interrupções é o fato de que processadores mais rápidos não vão causar uma redução significativa na latência das interrupções. Em um estudo realizado para o grupo de *PowerTrain* da GM, a latência mínima de tratamento de interrupções de 2 máquinas diferentes é praticamente a mesma, apesar de uma máquina ter o dobro da velocidade da outra. A velocidade do *clock* da CPU não tem grande influência nas medidas de latência mínima, já que este valor está mais relacionado com a velocidade do barramento [RadiSys, 1998].

Embora as interrupções sejam muito úteis e práticas, o tratamento delas dificulta a previsibilidade dos sistemas de tempo real [Timmerman, 2001].

De acordo com Ganssle [Ganssle, 2004], mesmo com décadas de pesquisa, métodos formais para provar que um *software* funciona corretamente ainda são impraticáveis para sistemas reais, além de não existir uma forma real e útil de calcular-se a previsibilidade de um sistema. Ganssle ainda afirma que é difícil prever o comportamento de um programa quando eventos assíncronos alteram o fluxo de execução do programa, de forma que a multitarefa e as interrupções levam a problemas impossíveis de serem analisados.

Discussões recentes em *newsgroups* e conferências de sistemas embarcados chegaram a discutir o abandono do uso de interrupções, o que tornaria os sistemas mais fáceis de analisar. De fato, é bastante fácil analisar um sistema de tempo real que possua apenas *Polled Loops* [Laplante, 2004].

Na verdade, a programação de sistemas de tempo real sem interrupções é freqüentemente usada em alguns casos usando *polling*, quando deseja-se aumentar o determinismo e previsibilidade do sistema. Previsibilidade refere-se ao fato de um sistema cumprir os prazos de todas suas tarefas, em qualquer ocasião [Stankovic e Ramamritham, 1990].

Porém, a possibilidade de usar interrupções oferece flexibilidade ao sistema e facilidade para seu desenvolvimento. Dessa forma, existem alternativas para se usar com qualidade interrupções em sistemas de tempo real. Uma abordagem interessante proposta em [Zhang e West, 2006] faz com que o sistema operacional leve em consideração as prioridades dos processos, antes de atender uma interrupção, fazendo com que interrupções associadas a processos de baixa prioridade, não penalizem processos de maior prioridade que deveriam estar sendo executados.

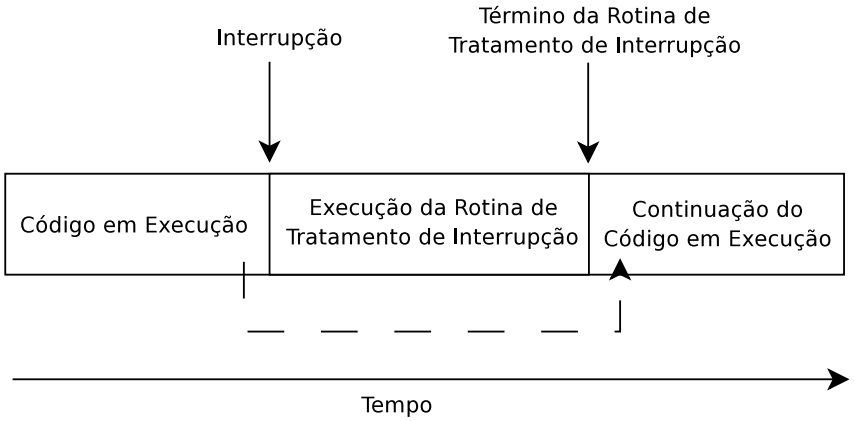


Figura 8: Tratamento imediato da interrupção

Esta abordagem é bastante interessante já que os sistemas operacionais de tempo real, normalmente são baseados em resposta a eventos externos, sendo reativos por natureza. Além disso, normalmente a resposta a eventos externos é feita via interrupções [Lamie e Carbone, 2007]. Em teoria, um sistema operacional perfeito nunca desabilita as interrupções [Dougan e Mwaikambo, 2004]. A influência das interrupções pode ser tão grande, que existem esforços para tratar-se interrupções consideradas de tempo real de forma diferenciada, proporcionando menos latência para as tarefas de tempo real [del Foyo et al., 2006].

Existem várias formas de se tratar uma interrupção. Cada sistema utiliza a que mais lhe é conveniente. A mais simples consiste em tratar a interrupção no momento em que ela ocorre, e depois continuar executando o que estava sendo executado anteriormente do ponto em que parou. O fluxo deste modelo de tratamento de interrupção pode ser visto na figura 8.

Já na figura 9, o sistema operacional recebe a interrupção e avisa o PIC que ela já foi tratada, para que outras interrupções possam ocorrer. Entretanto, ao invés de tratar a interrupção, o sistema operacional agenda o momento mais adequado para tratar esta interrupção, que pode ser após a tarefa atual (como mostrado na figura 9) ou até mesmo posteriormente. Esta abordagem é interessante, pois algumas interrupções podem não precisar de atenção imediata, como o aviso que uma operação de DMA entre disco e memória terminou.

A figura 10 mostra uma possibilidade que pode ocorrer no esquema proposto na figura 9 quando o sistema operacional é preemptivo. Os sistemas operacionais não preemptivos esperam que as tarefas liberem o processador voluntariamente, para que outras possam usar o processador. A maioria dos sistemas operacionais de tempo real são preemptivos, de forma que a qualquer momento eles podem interromper o

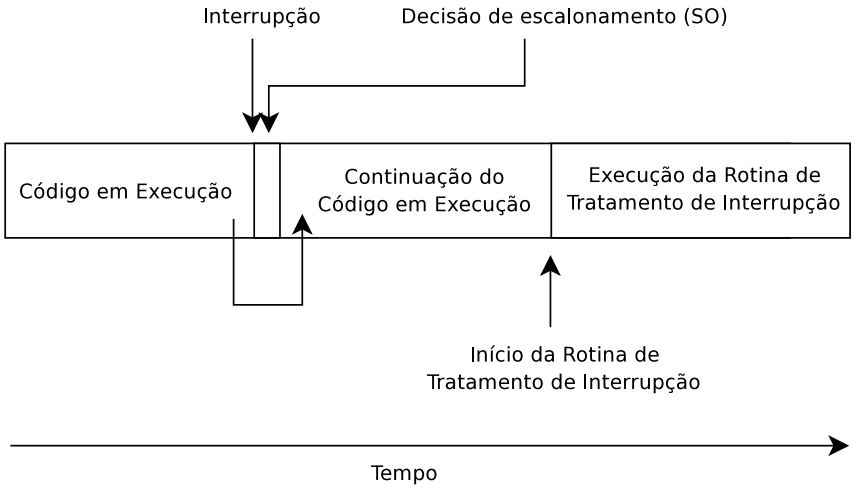


Figura 9: Tratamento da interrupção é agendado pelo escalonador do sistema operacional

que está sendo executado para permitir que uma operação de maior prioridade seja executada. No caso da figura 10, o sistema operacional parou a execução da rotina de tratamento de interrupção para executar uma tarefa mais importante. Em seguida voltou a executar a rotina de tratamento de interrupção de onde ela havia sido preemptada.

Idealmente a rotina de tratamento de interrupção deve detectar a interrupção, e agendar uma tarefa para tratar esta interrupção posteriormente, para que o ISR seja o mais rápido possível [Baskiyar e Meghanathan, 2005] e reduzir a latência para tratar interrupções. Esta situação é a descrita nas figuras 9 e 10.

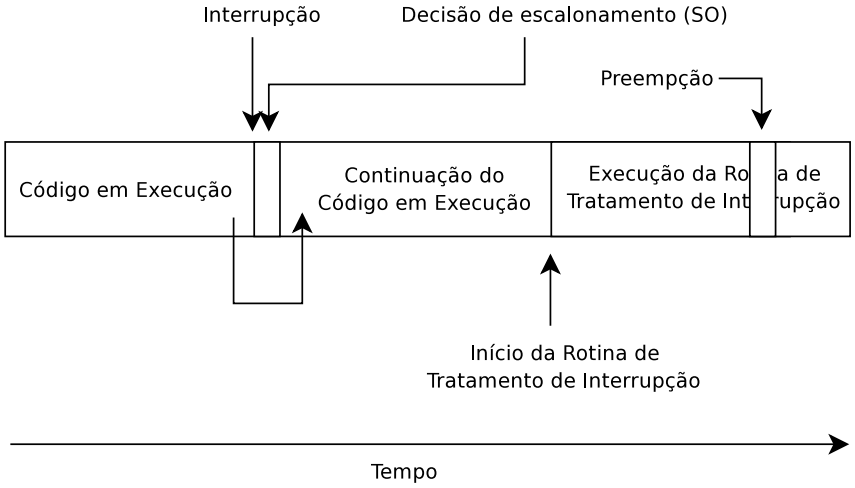


Figura 10: Tratamento da interrupção é agendado pelo escalonador do sistema operacional com possibilidade de preempção

4.2.10 Clock Tick

Embora o *clock tick* já tenha sido apresentado e algumas de suas características discutidas no capítulo 3, é conveniente e importante mencionar mais alguns detalhes sobre ele. O *clock tick* é uma interrupção especial que ocorre periodicamente. Quanto maior a freqüência de *ticks*, maior o *overhead* do sistema [Labrosse, 2002], pois o processador irá gastar muito tempo tratando as interrupções de *clock tick*, sobrando menos tempo para as tarefas dos usuários.

O *clock* do sistema é o mecanismo básico que faz com que a multitarefa e a confiabilidade dos sistemas operacionais tornem-se realidade. Para cada interrupção de *clock tick*, a rotina de tratamento de interrupções executa diversas tarefas para manter o sistema em ordem. A listagem abaixo, adaptada do livro *"The Design of The Unix Operating System"* [Bach, 1986], descreve algumas destas funções:

- Reiniciar o *timer* de hardware: Como já foi discutido, os *timers* podem ser programados para operarem em um modo repetitivo ou no modo *One-Shot*. Algumas vezes, o modo *One-Shot* é usado para aumentar a precisão, entretanto a cada interrupção o *timer* precisa ser reprogramado;

- Verificar a execução de tarefas associadas a *timers*: Muitas tarefas solicitam ao sistema operacional que certas operações sejam executadas a partir de *timers* do sistema operacional. O sistema operacional mantém tabelas chamadas de *call out tables*, que possuem contadores de cada tarefa associada a um *timer*, de forma similar a uma contagem regressiva. Quando a contagem chega em zero, o sistema operacional dispara a execução do código desejado;

- Atualizar informações de uso de processador e memória: Toda vez que o *clock* interrompe a execução da CPU, estatísticas são coletadas e acumuladas sobre o estado atual do sistema com relação a uso do processador e memória;

- Atualizar a data e hora: O sistema operacional atualiza uma variável a cada *clock tick* desde o momento do *boot* do sistema. Uma outra variável, que é atualizada com menos freqüência, guarda a data e hora no formato que é compreendido pelas pessoas;

- Controlar o escalonamento de processos: O escalonador decide qual será a próxima tarefa a ser executada e se alguma tarefa em execução deve ser interrompida em detrimento da execução de tarefas com maior prioridade;

- Ativar o processo de *swapping*: Quando um computador precisa executar mais programas do que sua memória suporta, o sistema operacional precisa realizar um processo chamado de *swap* que consiste em colocar partes da memória em disco, e trazer de volta para memória conforme a demanda dos programas. Como esta operação é lenta e custosa devido ao uso de discos, é pouco recomendável para sistemas de tempo real.

Alguns destes itens são executados a cada *clock tick*, enquanto outros são executados a cada quantia predeterminada de *clock ticks*. Como a interrupção de *clock* possui uma das mais altas prioridades do sistema, a rotina de tratamento de interrupção de *clock* deve ser muito rápida, para evitar que interrupções externas sejam perdidas enquanto a rotina de tratamento de interrupção do *clock* é executada [Bach, 1986].

Como o processador é apenas um, seria impossível executar no mesmo instante de tempo tanto instruções do sistema operacional, quanto do aplicativo em execução. Dessa forma, quando o sistema operacional passa o controle para um programa, este programa tem total controle do processador com algumas restrições. As restrições são impostas pelo processador e pela MMU, que não permitem que um programa comum execute instruções privilegiadas (como escrever diretamente numa porta de saída ou reiniciar o computador), ou acessar endereços de memória fora de seu escopo.

Se o programa não terminar seu processamento antes do *timer* pré-programado pelo sistema operacional gerar uma interrupção, a interrupção será gerada, e o sistema operacional voltará a assumir o controle, evitando que uma tarefa com problemas trave o sistema por não "devolver" o controle do processador [Stallings, 2003].

Dessa forma, para o escolanador funcionar bem em um sistema operacional de tempo real, é preciso um *timer* de alta resolução [Aarno, 2004]. Entretanto, deve-se tomar cuidado com o *overhead*. Por exemplo, se um *timer* que gera uma interrupção a cada 1 milisegundo (1000 vezes por segundo) tiver um custo de processamento de 100 microsegundos para a CPU a cada interrupção gerada, o computador irá gastar 10% de toda sua capacidade de processamento apenas atualizando o relógio [Shaw, 2003] e realizando tarefas do sistema operacional. Assim, *ticks* muito curtos fazem com que a CPU gaste uma larga fração do seu tempo com o sistema operacional [Piccioni et al., 2001].

Normalmente, o tempo e a medida do tempo estão relacionados com interrupções, sendo necessário que uma análise de tempos máximos seja executada para softwares governados por interrupções [Brylow e Palsberg, 2004].

Por mais que alguns sistemas operacionais prometam capacidades de tempo com resoluções baixas, isto nem sempre é possível devido ao *hardware*. Normalmente o PC utiliza o PIC ou PIT para gerar suas interrupções de *timer*, sendo que a sua maior capacidade é de 8192Hz, gerando uma interrupção a cada $122\mu s$. Contudo, com esta alta freqüência de interrupções, o processador terá que ficar tratando interrupções a todo momento, mesmo sem necessidade, aumentando a carga de processamento. É por este motivo que normalmente os sistemas operacionais configuram o *timer* para 100Hz, levando a intervalo de 10ms entre cada interrupção. Esta seria a resolução mínima de execução de uma tarefa. Entretanto usando o novo recurso APIC, é possível se obter uma resolução de $0{,}1\mu s$, mas ao se descartar o tempo gasto para troca de contexto e *house-keeping*[4], seria possível obter uma resolução de $1\mu s$. [Oberle e

[4] Tarefas de rotina executadas pelo sistema operacional para manter o sistema confiável.

Walter, 2001].

4.2.11 RISX X CISC

Com relação ao conjunto de instruções disponibilizado pelos processadores, existem dois principais conjuntos: o conjunto restrito de instruções - RISC (*Restricted Instruction Set)* e o conjunto complexo de instruções - CISC (*Complex Instruction Set).*

Os processadores de arquitetura CISC (por exemplo: 80486, Pentium e Athlon) possuem *hardware* bastante complexo, com instruções que são capazes de multiplicar matrizes ou comparar *strings* de texto completas via *hardware.* Já processadores da arquitetura RISC (por exemplo: PowerPC, MIPS e ARM) possuem poucas instruções, que são bastante simples, como somar, adicionar e comparar. Com base nestas instruções, programas mais complexos podem ser construídos. Com isto, o *hardware* de um processador RISC é mais simples.

Como as interrupções só ocorrem entre duas instruções (entre o momento que uma termina e a próxima será executada), a arquitetura RISC oferece latências de interrupção muito menores, já que cada instrução só leva um ciclo de *clock* para ser executada. Dessa forma, é comum, e preferível o uso de processadores RISC para implementar sistemas de tempo real.

No desenvolvimento de um sistema com restrições temporais críticas, o tempo para completar a execução das instruções também deve ser computado, já que dependendo da instrução os computadores podem levar diferentes tempos para completar a execução destas [Laplante, 2004]. Freqüentemente não se dedica muita atenção a esta análise, levando a problemas misteriosos [Laplante, 2004].

4.2.12 Uso de PCs no controle de sistemas de tempo real

Como já comentado, o *hardware* e sua arquitetura têm papel fundamental no suporte ao funcionamento do sistema operacional. Talvez um dos recursos mais importantes fornecido pelo *hardware* ao sistema operacional são as interrupções. Graças as interrupções, é possível disparar eventos de forma assíncrona, sem a necessidade de realizar *polling.*

Quando várias interrupções estão pendentes, o *hardware* do computador seleciona automaticamente a de maior prioridade. Em alguns casos, interrupções podem ser interrompidas por outras interrupções de maior prioridade. Tempos dos ISRs são constantemente pesquisados e estudados por projetistas de sistemas operacionais, sendo cuidadosamente medidos e publicados por fabricantes de *software* e usuários [Shaw, 2003].

Entretanto, todas as características discutidas nesta seção do texto, indicam uma arquitetura projetada para ter a maior velocidade possível e não o melhor determinismo possível. Todas estas melhorias nas arquiteturas modernas de computadores tornam o sistema impossível de ser analisado do ponto de vista de performance [Laplante, 2004]. Por outro lado, vem observando-se cada vez mais o uso de PCs

genéricos para controle de sistemas de tempo real. Isto de fato é possível utilizando-se algumas técnicas e cuidados.

Mantegazza e Dozio [L. Dozio, 2003] propõem o uso de computadores *desktop* (GPCPU - *General Purpose* CPU) para implementar tarefas de controle de alta freqüência a baixo custo. Utilizando-se um RTOS, além de se poder implementar esta plataforma de controle de alta performance a baixo custo, ainda ganha-se a vantagem de uso dos recursos disponíveis em computadores *desktop*, como as unidades para cálculo de ponto flutuante.

De acordo com Cawlfield, é possível implementar sistemas de controle baseados em PCs com alta confiabilidade usando técnicas de tolerância a falhas. Além disto, nenhum sistema operacional para PCs pode ser certificado como "livre de erros" [Cawlfield, 1997].

Outro caso de sucesso recente da possibilidade de uso da arquitetura genérica x86 em sistemas de tempo real, foi sua utilização em aplicações críticas militares em um sistema de defesa desenvolvido pela IBM para a marinha dos Estados Unidos [McKenney, 2008], com diversas restrições temporais.

Em uma unidade de controle de experimentos com plasma, o RTAI substituiu um sistema baseado em *LynxOS*, com mesma confiabilidade e adicionando ganho de performance [Centioli et al., 2004], além de reduzir custos, trocando um *hardware* especializado baseado em VME por um hardware genérico baseado em PC.

Embora a arquitetura PC nem sempre seja a ideal para sistemas embarcados, devido a problemas de *cache* e outros recursos que foram feitos para aumentar a performance prejudicando o determinismo, muitos sistemas de tempo real utilizam a arquitetura PC, devido a seu baixo custo graças a sua economia de escala. Um exemplo é o robô autônomo sub-aquático da universidade de McGill no Canadá, que usa placas de um PC/104[5] para o controle do robô chamado de *Aqua* [Theberge et al., 2006]. Neste robô, um outro conceito interessante é utilizado. Tanto o sistema operacional de tempo real QNX quanto o Linux convencional (não de tempo real) são utilizados simultaneamente em dois processadores diferentes, para beneficiar-se das vantagens de cada um dos sistemas operacionais [Dudek et al., 2007].

4.3 *Kernels*

A tradução direta do inglês da palavra *kernel* significa núcleo. De fato o *kernel* é o núcleo de qualquer sistema operacional, provendo uma interface entre *hardware* e *software*. Como pode-se ver na figura 11, o *kernel* é o único meio através do qual as bibliotecas e aplicativos podem acessar o *hardware*. Da mesma forma, informações que chegam do mundo externo através do *hardware* são recebidas pelo *kernel* e em seguida encaminhas para o *software* associado a esta informação. Também pode ser visto na figura, que normalmente os aplicativos finais utilizam bibliotecas para fazer chamadas de sistema (*system calls)* que são chamadas de funções que fazem com que o *kernel* seja invocado para realizar alguma tarefa que só o

[5] Um computador x86 completo de pequenas dimensões, projetado para uso em sistemas embarcados. Maiores informações em http://www.pc104.org/

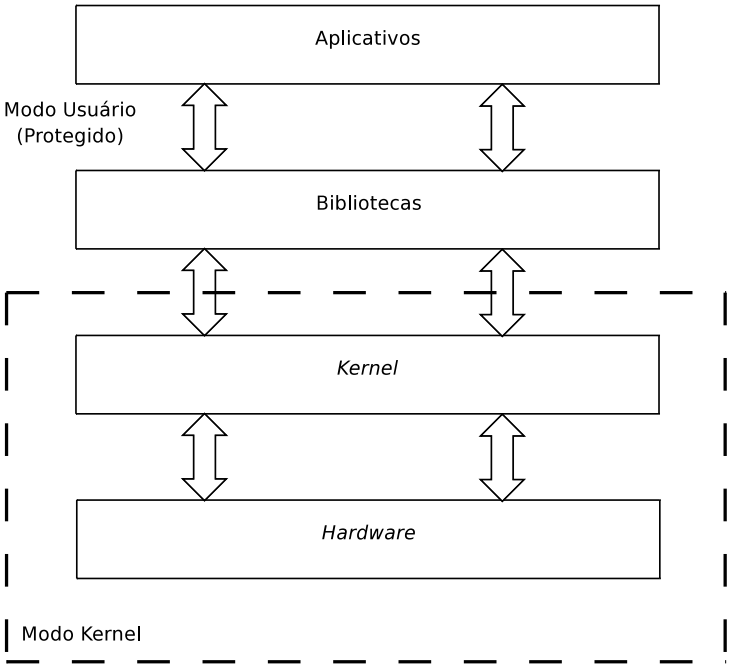


Figura 11: Interfaces de um *kernel*

sistema operacional pode fazer, como abrir um arquivo, ou obter uma informação da rede.

De acordo com Ganssle [Ganssle, 2004], um *kernel* deve oferecer os seguintes serviços :

1. Permitir dividir uma aplicação em tarefas;
2. Disponibilizar um relógio (*clock tick*) que permita suspender tarefas até um determinado momento ou para impor tempos limite;
3. A possibilidade de tarefas ou rotinas de tratamento de interrupção enviarem mensagens para outras tarefas;
4. Sempre executar a tarefa de maior prioridade que está pronta para execução. O escalonador é executado toda vez que ocorre um evento e existe a possibilidade de uma tarefa mais importante ser executada;
5. Realizar a troca de contexto, que é usada para salvar o estado atual de uma tarefa e restaurar o estado de uma tarefa mais importante.

Os sistemas operacionais podem ser bastante simples, não oferecendo serviços como rede ou acesso a arquivos, mas apenas escalonamento de processos, ou mais complexos, oferecendo diversos serviços para as bibliotecas e aplicativos. Muitos sistemas operacionais consistem simplesmente de um *kernel*.

Um dado muito importante e freqüentemente questionado, refere-se a sobrecarga imposta por um sistema operacional em um computador. Em geral um sistema operacional de tempo real aumenta o uso

do processador de 2% a 4% [Labrosse, 2002], porém muitas vezes um *hardware* melhor pode não ajudar por limitações no *software*. De acordo com Weiss [Weiss et al., 1999], existem situações em que 77% do tempo de execução de uma tarefa é consumido pelo RTOS, de forma que o RTOS torna-se um fator limitante da performance do sistema.

Independente do seu tipo, um *kernel* faz com que o tempo de resposta de tarefas de alta prioridade seja virtualmente inalterado ao adicionar novas tarefas de menor prioridade [Ganssle, 2004]. Isto é muito importante para construir sistemas escaláveis e confiáveis.

Normalmente, o *kernel* utiliza recursos disponibilizados pelo processador e MMU para separar de forma segura o núcleo do sistema e as tarefas em execução, como pode ser visto na figura 11. Quando o *kernel* inicia a execução de uma tarefa, ele coloca o processador em um modo chamado de modo protegido (ou modo usuário), reduzindo o número de instruções possíveis e posições de memórias que podem ser acessadas, para que uma tarefa não possa afetar em nenhum aspecto o estado de outras tarefas. Como o modo protegido é bastante limitado, as tarefas precisam fazer pedidos para o *kernel*, como acessar a rede, usar uma porta de entrada e saída, ou ler e escrever no disco. Quando funções internas do *kernel* estão sendo executadas, o código em execução tem acesso irrestrito a memória e *hardware*. Este modo é o modo *kernel,* que é o modo em que o *kernel* é executado.

Durante a execução em modo protegido, qualquer operação ilegal executada pelos programas vai gerar uma interrupção de *software* sinalizando o erro, que irá imediatamente interromper a execução da tarefa que causou o erro, e avisar o sistema operacional, para que uma decisão sobre o que deve ser feito seja tomada.

Um *kernel* pode ser não preemptivo ou preemptivo. A maioria dos *kernels* atuais são preemptivos, de forma que o *kernel* sempre tenta executar a tarefa de maior prioridade que esteja pronta para execução. Embora um *kernel* possa ser preemptivo com relação as tarefas que ele esteja controlando, deve-se observar o fato de freqüentemente o *kernel* em si, não ser preemptivo. Isto quer dizer que se um *kernel* estiver executando alguma tarefa interna do *kernel* e tiver desativado todas interrupções até terminar esta tarefa crítica, por mais importante que seja o evento ou tarefa que aconteça neste intervalo de tempo, ela só será detectada e tratada quando a execução da tarefa crítica atual terminar.

O próprio *kernel* do Linux não pode ser "preemptado" em alguns momentos, pois ele desabilita as interrupções enquanto está executando operações críticas [Aarno, 2004]. Na verdade, partes do *kernel* do Linux podem ser preemptadas e algumas partes não, dependendo da sua importância e necessidade de execução atômica.

Outra ocasião em que as interrupções podem acabar sendo desabilitadas por segurança está na execução de funções reentrantes. Uma função reentrante basicamente utiliza seus dados de forma que mesmo sendo executada várias vezes de forma simultânea (por exemplo, sendo chamada a partir de várias tarefas), sua resposta é correta. A melhor opção para tornar uma determinada porção de código reentrante,

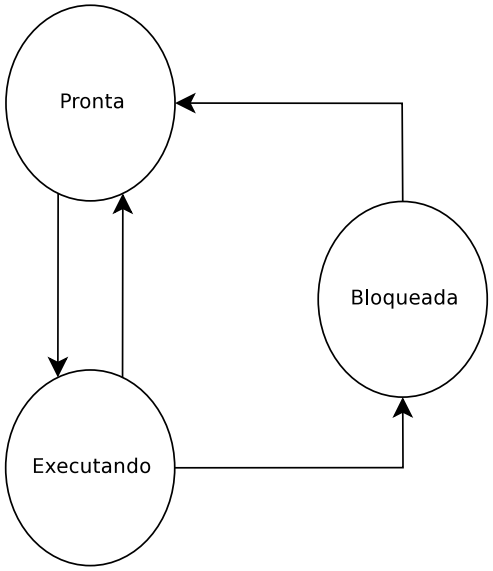


Figura 12: Diagrama simplificado de possíveis estados de uma tarefa

é eliminar o uso de variáveis globais. Contudo, as variáveis globais são a maneira mais rápida de trocar dados em um programa, não sendo possível eliminar todas as variáveis globais em sistemas de tempo real [Ganssle, 2004]. Dessa forma, a abordagem mais comum é desabilitar as interrupções em um código reentrante e reativar as interrupções ao terminar de executar o código reentrante. Isto certamente aumenta a latência do sistema e reduz sua habilidade de responder a eventos externos em tempo [Ganssle, 2004].

Um *kernel* que fica com as interrupções desabilitadas muito tempo (para tratar regiões críticas), acaba aumentando a latência das interrupções [Farines et al., 2000, Beal, 2005]. De acordo com Labrosse, a especificação mais importante de um sistema de tempo real é a quantia de tempo que as interrupções ficam desligadas [Labrosse, 2002].

O coração de um sistema de tempo real é o escalonador de tarefas, que decide qual tarefa será executada, quando, e por quanto tempo. A maior parte dos escalonadores utiliza um tempo limite para preemptar a tarefa em execução atualmente. Claramente, o mecanismo de temporização deve ser bastante preciso, caso contrário tarefas podem ser executadas antes ou depois do momento ideal [Kailas e Agrawala, 1997].

A figura 12 mostra os possíveis estados de uma tarefa e suas transições. Após ser criada, uma tarefa fica pronta. Isto quer dizer que ela está na fila aguardando para ser executada. O sistema operacional pode colocá-la em execução em qualquer momento, e também parar (preemptar) sua execução a qualquer momento, caso uma tarefa de maior prioridade precise ser executada. Caso uma tarefa em execução solicite o uso de algum recurso, como um arquivo em disco, rede, uma porta de entrada e saída, o *kernel* bloqueia esta tarefa até que o recurso esteja disponível para ser utilizado. Quando o recurso estiver disponível, a tarefa volta a ficar pronta, para só em seguida poder ser executada novamente.

Quando nenhuma tarefa está sendo executada, normalmente, o sistema operacional executa uma tarefa chamada de "Tarefa Ociosa do Sistema" *(Idle Task)* [Ganssle, 2004], que embora não faça nada, representa o tempo ocioso do sistema. Em alguns sistemas com necessidades de economia de energia, é neste momento que coloca-se o processador em modo de baixo consumo, pois qualquer interrupção irá trazer a execução de volta ao estado convencional.

A tarefa de tempo ocioso do sistema também é útil para calcular a porcentagem de uso que o sistema está demandando do processador. Se T é o tempo que a *Idle Task* consome do processador, a carga total de processamento do sistema é (1-T).

A freqüência com que a interrupção de *clock tick* interrompe os programas para executar o sistema operacional, define também o *time quantum* [Calandrino e Anderson, 2006]. O *time quantum*, é a menor fatia de tempo de processamento que o escalonador pode fornecer a uma tarefa. Dessa forma, se o *clock tick* for de 1000Hz, o sistema terá um *time quantum* de 1ms. Se uma tarefa durar 1,1ms, ela precisará de 2 *time quantums*. Existem relatos de sistemas de tempo real com *clock tick* de 3750Hz, levando a um *time quantum* de 250μs, sem aumentar consideravelmente o *overhead* do sistema operacional [Calandrino e Anderson, 2006].

O número de tarefas no sistema também contribui para o aumento da latência, pois o sistema operacional costuma desabilitar as interrupções enquanto percorre a lista de tarefas pendentes [Laplante, 2004]. Sabe-se que existe uma relação quadrática entre o *overhead* causado pelo sistema operacional e o número de tarefas [Kohout et al., 2003]. Na área de sistemas de tempo real, caso exista um sistema com muitas tarefas, é necessário executar medidas de latência para verificar se o escalonador não está desabilitando interrupções por tempos longos e inaceitáveis [Laplante, 2004].

Os *kernels* também podem variar com relação a sua arquitetura interna. A maioria dos *kernels* são classificados como monolíticos. Um *kernel* monolítico executa todas suas tarefas em *kernel space*, incluindo escalonamento, gerenciamento de memória, gerenciamento de sistema de arquivos em discos, controladores de dispositivos (*device drivers)* e serviços de sincronização e comunicação entre tarefas e processos (IPC - *Inter Process Communication*).

Já a filosofia que o *microkernel* implementa consiste em executar unicamente as partes essenciais do *kernel* em modo *kernel,* que normalmente consistem do escalonamento de processos, gerenciamento de memória e IPC. Todo resto do sistema operacional, incluindo até mesmo *device drivers* é implementado como tarefas em modo usuário. A grande vantagem deste modelo é que o sistema operacional torna-se muito robusto. Até mesmo um *hardware* defeituoso não seria capaz de causar danos ao sistema em execução, já que seu código está executando em modo protegido. A desvantagem é que como o sistema operacional está divido em espaço de *kernel* e epaço de usuário, existe a necessidade de constantes trocas de contexto, aumentando a sobrecarga do sistema operacional. O sistema operacional mais conhecido que implementa um *microkernel* é um sistema operacional de tempo real da QNX chamado de Neutrino.

4.4 Sistemas Operacionais de Tempo Real (RTOS)

Um *kernel* de tempo real, também chamado de sistema operacional de tempo real (RTOS) permite que aplicações de tempo real sejam projetadas, mantidas, expandidas e alteradas facilmente e com segurança. A adição de tarefas de baixa prioridade ao sistema, não causa influência nenhuma nas outras tarefas de prioridade mais alta já existentes [Labrosse, 2002].

Embora seja possível implementar um sistema de tempo real sem o uso de um sistema operacional, esta tarefa não é fácil e nem aconselhável [Barabanov, 1997]. Além disso, a execução de vários programas de forma simultânea pode causar tempos de resposta inesperados, pois os programas podem interagir de formas não esperadas [Wolf, 2007b].

A escolha de um RTOS (*Real Time Operating System)* é importante para dar suporte a *prioridades, interrupções, timers, comunicação entre tarefas, sincronização, gerenciamento de memória e multiprocessamento* [Baskiyar e Meghanathan, 2005].

De acordo com Labrosse, existem mais de 150 fornecedores de RTOS. Os pacotes de desenvolvimento podem variar de US$70,00 a US$30.000,00 ou mais, e em alguns casos, é preciso pagar *royalties* ao fabricante do RTOS para cada venda realizada do produto final [Labrosse, 2002].

Os sistemas operacionais de tempo real podem ser desenvolvidos do "zero", a partir de um projeto específico e formal para o seu desenvolvimento, ou através de uma camada de adaptação (normalmente de baixo nível) entre um sistema operacional existente e o *hardware*. Normalmente os sistemas desenvolvidos desde o princípio para ser um RTOS seguem um processo de desenvolvimento bastante formal para que o sistema possa ser facilmente validado e certificado por normas e padrões internacionais de segurança. A seguir pode ser vista uma lista de alguns sistemas operacionais de tempo real desenvolvidos desta forma:

- *Integrity*[6] da *Green Hills Software* que é utilizado em aviões como o A380, o F35 e o Eurofighter;
- DEOS (*Digital Engine Operating System*) da *Honeywell*, que é utilizado pelo Boeing 777 [Krodel e Romanski, 2007] e nos sistemas de controle de vôo de jatos regionais da Embraer [Adams, 2005];
- *LynxOS*[7] da *LynxWorks* que é tradicionalmente usado em aplicações militares e aeroespaciais. Também é usado para implementar parte do sistema de controle de tráfego aéreo dos EUA;
- *VxWorks*[8] da *Wind River Systems* que é tradicionalmente utilizado pela agência espacial dos EUA, a NASA, para implementar robôs de exploração espacial.

A outra maneira de implementar-se um RTOS é adaptando um sistema operacional atual para responder a requisitos de tempo real com confiabilidade. Uma proposta para realizar esta adaptação chama-se ADEOS. O ADEOS (*Adaptative Domain Environment for Operating Systems)* permite dar previsibilidade

[6] Maiores informações em: http://www.ghs.com/products/rtos/integrity.html
[7] Maiores informações em: http://www.lynuxworks.com/solutions/milaero/milaero.php3
[8] Maiores informações em: http://cdn.windriver.com/products/vxworks/

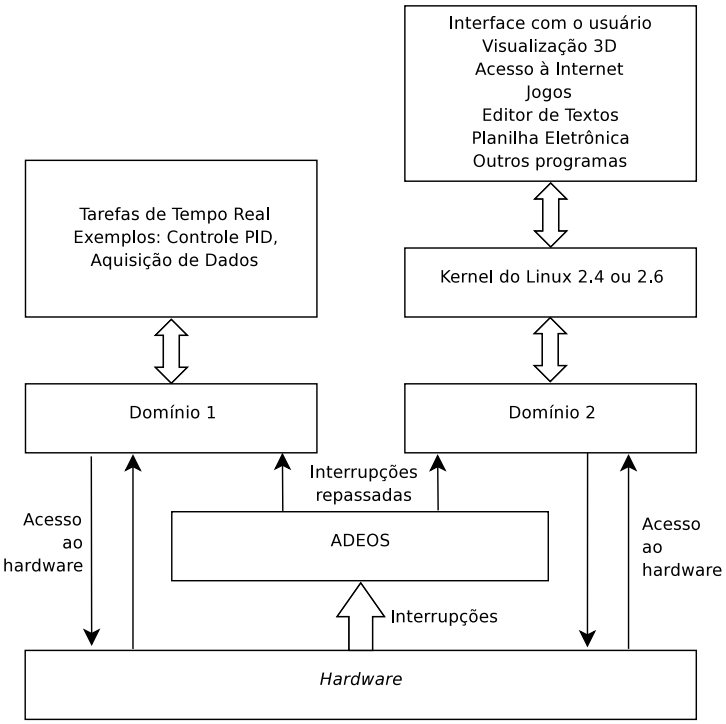


Figura 13: Arquitetura de um sistema com ADEOS

ao tratamento de interrupções no sistema [Gerum, 2005]. O ADEOS implementa um sistema chamado *Optimistic Interrupt Protection* [Gerum, 2005] que é descrito em [Stodolsky et al., 1993]. A arquitetura geral de um sistema baseado em ADEOS pode ser vista na figura 13. Como pode ser observado pela figura, uma das grandes vantagens do ADEOS, é proporcionar o melhor dos dois mundos. Um mesmo computador pode ser utilizado tanto para tarefas de tempo real quanto de interface com o usuário.

O ADEOS foi proposto inicialmente em 2002 [Yaghmour, 2002] e sua arquitetura, em teoria, permite que vários sistemas operacionais sejam executados simultaneamente em um mesmo computador. Com o ADEOS o sistema é dividido em domínios, criando uma fila de interrupções (*interrupt pipeline* - *ipipe*) que é distribuída para estes domínios. O ADEOS é responsável por interceptar todas as interrupções do *hardware*, e roteá-las para um domínio específico, como pode ser visto na figura 13. O RTAI, Xenomai e RTLinux, três versões de Linux de Tempo real, implementam uma arquitetura baseada em ADEOS, de forma que o ambiente de tempo real é o domínio de maior prioridade, dando uma resposta determinística para as tarefas de tempo real [Barbalace et al., 2008].

Em outras palavras, o que o ADEOS faz é isolar tarefas de tempo real e convencionais, permitindo que o Linux seja executado normalmente, permitindo o uso de vários aplicativos gráficos, rede, dentre outros utilitários. Como o Linux todo, incluindo seu *kernel,* consiste de um dos domínios do ADEOS de baixa prioridade (ver figura 13), quando uma tarefa de tempo real precisa ser executada, o Linux todo é "preemptado". Como esta operação é bastante rápida, o usuário não percebe a diferença. Entretanto,

caso a tarefa de tempo real esteja sendo executada com uma freqüência muito grande, o usuário pode ter a impressão que o computador todo travou, já que o *mouse*, teclado e tela podem parar de responder, mantendo as tarefas de tempo real em operação. Também é comum dizer que esta camada de adaptação é um *nanokernel*, já que ela é resposável por decidir quando um domínio recebe as interrupções ou não.

Outra abordagem que foi comum durante anos e ainda permanece nos dias atuais é utilizar o Windows NT para controlar sistemas de tempo real. O *time quantum* do Windows NT é de 20ms, que pode ser demasiadamente longo para algumas situações [Dedicated Systems, 2001h]. A análise realizada pela *Dedicated Systems* mostrou que o tratamento de interrupções no Windows NT é bastante rápido, tendo um valor médio de 3,7μs com máximo de cerca de 22μs. Com o sistema sobrecarregado, foram obtidos valores superiores a 600μs [Dedicated Systems, 2001h]. A conclusão é que o Windows NT é um excelente sistema operacional de propósito geral, porém inadequado para tarefas de tempo real [Martin Timmerman, 2000a].

Com isto, surge a necessidade de extensões capazes de dar uma capacidade realista de tempo real para o Windows NT [Martin Timmerman, 1998] e seus sucessores (Windows 2000, XP e Vista). O *Hyperkernel* implementa este recurso usando o mesmo conceito de *kernel* de separação [Martin Timmerman, 2000b], como utilizado no ADEOS pelo RTAI e Xenomai, entretanto possui problemas de determinismo. Mesmo assim uma latência máxima de 19μs foi observada em testes realizados com o *Hyperkernel* em um Pentium MMX 200 MHz [Dedicated Systems, 2001c].

Outra extensão para o Windows NT, o *INTime,* utiliza o mesmo princípio [Martin Timmerman, 2000c], adicionando proteção de memória entre os domínios, conferindo maior confiabilidade e estabilidade ao sistema. A latência máxima obtida no mesmo Pentium foi de cerca de 24μs [Dedicated Systems, 2001d].

Outro produto comercial, o *RTX*, utiliza a abordagem de implementar suas características de tempo real como um *driver* do Windows NT. Isto oferece um bom determinismo, mas se um *driver* de qualquer dispositivo do Windows apresentar problemas, isto irá interferir no sistema de tempo real [Martin Timmerman, 2000e]. O relatório de testes feito pela *Dedicated Systems* no mesmo Pentium 200 obteve um tempo máximo para tratamento de interrupções de cerca de 15μs [Dedicated Systems, 2001f].

Em uma comparação entre as três extensões, os especialistas da *Dedicated Systems* acabaram por concluir que o *INTime* é a melhor opção, pois apesar de ser mais lento que seus concorrentes, ele oferece maior segurança e proteção [Dedicated Systems, 2001a].

Como vem sendo discutido, as interrupções e conseqüentemente suas latências, desempenham um papel importante em vários aspectos dos sistemas operacionais e dos sistemas de tempo real. Em um RTOS, o tempo de resposta de interrupção a ser considerado deve ser sempre o de pior caso. Se um sistema responde uma a interrupção em 50μs 99% das vezes, e responde em 250μs em 1% das vezes, o tempo de resposta deve ser como considerado 250μs [Labrosse, 2002].

A latência para tratamento das interrupções pode ser bastante influenciada pelo sistema operacional. Em geral, pode-se calcular a latência de uma interrupção como:

`L = [tempo máximo que as interrupções ficam desligadas] + [tempo para iniciar a execução da primeira instrução de uma ISR]`

A resposta de interrupção é definida como o tempo entre a ocorrência de uma interrupção e o início da execução do código do usuário que trata a interrupção.

No caso de sistemas preemptivos, ela é definida como

`R = L + [Tempo para salvar o contexto da CPU] + [Tempo de execução da rotina de entrada em ISR do` *`Kernel`*`]`

Outra medida importante de um *kernel,* é o tempo de recuperação de interrupções, que consiste no tempo que o sistema leva para voltar a sua operação normal após uma interrupção ter sido tratada. No caso de sistemas preemptivos, pode-se calcular este tempo como:

TR = [Tempo para determinar se existe alguma tarefa de maior prioridade pronta] + [Tempo para restaurar o contexto da CPU] + [Tempo para executar a instrução de retorno de interrupção]

Em todas estas situações, o sistema operacional influencia pelo tempo que mantém as interrupções desligadas, e pelo tempo que leva para tomar suas decisões de escalonamento.

4.5 Serviços de um RTOS

Em teoria, tanto os sistemas operacionais de tempo real (RTOS) quanto os sistemas operacionais de propósito geral (GPOS), como Linux, MacOS e Windows possuem mecanismos internos e funcionamento bastante semelhantes.

A grande diferença presente nos RTOS são técnicas e otimização utilizadas para favorecer o determinismo e previsibilidade do sistema. Isto é feito utilizando-se algoritmos especiais de escalonamento. De fato, a maioria dos sistemas operacionais não implementam políticas de qualidade de serviço (QoS), dando maior prioridade para tarefas mais importantes e dividindo a CPU de forma mais justa. Os algoritmos de escalonamento usados normalmente em sistemas de tempo real como por exemplo, o *Rate Monotonic* (RM) e o *Earliest Deadline First* (EDF) implementam sistemas de escalonamento que podem garantir uma maior qualidade de serviço para os programas de tempo real. [Ingram, 1999]

A seguir serão discutidos alguns serviços que os sistemas operacional disponibilizam para os programas que estão sendo executados com foco em tempo real.

4.5.1 Mecanismos de Sincronização e Comunicação

Semáforos

O Semáforo é um mecanismo de sincronização disponível em praticamente todos os sistemas operacionais. Sua função é permitir que várias tarefas compartilhem o uso de recursos de forma concorrente com confiabilidade. Os recursos podem ser os mais diversos possíveis, como informações na memória, placas

Listagem 1 Exemplo de implementação falha de região crítica

```
1 enquanto (recurso_em_uso = sim) aguarde
2 recurso_em_uso =sim
3 //Executa código crítico
4 recurso_em_uso=nao
```

de aquisição, *displays*, teclados, dentre outros.

Os semáforos podem ser binários ou de contagem. Um semáforo binário é usado para controlar o acesso a um único recurso, como um *display*, possuindo apenas 2 estados (em uso ou livre). Um semáforo de contagem é usado quando deseja-se acessar um recurso de uma coleção de vários recursos iguais. Muitos autores também chamam os semáforos binários de *mutex* para enfatizar sua característica de exclusão mútua entre 2 tarefas (*MUTual EXclusion*) [Laplante, 2004, Tanenbaum, 2001].

Quando uma região de código utiliza um dispositivo, muitas vezes esta região de código precisa executar várias operações para terminar de usar o dispositivo. Se durante estas operações, o escalonador começar a executar outra tarefa, esta outra tarefa também poderá tentar usar o dispositivo, causando inconsistências e erros no uso deste dispositivo.

Um exemplo clássico seria duas tarefas imprimindo texto em uma impressora, ou na porta serial disponível em um sistema embarcado.

A tarefa A imprime:

```
O resultado de i é 10
```

E a tarefa B imprime:

```
O valor de j é 7
```

É comum (e esperado) que o escalonador comece a executar a tarefa B antes de terminar a tarefa A, de forma que o resultado na impressora seria algo como:

```
O resulO valor de jtado de ié 710
```

A solução para este problema, é definir uma região crítica. Quando uma tarefa entra em uma região crítica, outro programa que precisa da mesma região crítica é bloqueado pelo sistema operacional ao tentar entrar nela, podendo continuar sua execução quando o primeiro processo terminar de usar a região crítica. Dessa forma, a impressora pode ser controlada a partir de uma região crítica, de forma que o primeiro processo "A" adquire a região crítica através de um semáforo e executa seus procedimentos até o fim, liberando então a região crítica, e conseqüentemente o recurso (neste caso, impressora) para outros processos/tarefas.

Intuitivamente é bastante simples implementar um semáforo para proteger uma região crítica, como pode ser visto no pseudo-código da listagem 1. Contudo, uma interrupção pode ocorrer logo após a execução da linha 1, e antes de executar a linha 2, quando a variável "`recurso_em_uso`" ainda não recebeu o valor "`sim`". Caso a interrupção tente utilizar este mesmo recurso, incoerências e resultados inesperados poderão ocorrer.

Os semáforos nada mais são que variáveis especialmente controladas pelo sistema operacional, de forma a garantir sua integridade. Entretanto, para efetuar esta tarefa, o sistema operacional pode utilizar recursos de hardware, como a instrução TSL (*Test and Set)* na arquitetura x86 que permitem em uma única instrução, verificar o conteúdo de uma posição de memória e alterar este conteúdo, de forma que não existe o risco do escalonador trocar tarefas enquanto o semáforo está sendo atualizado, como poderia ocorrer na listagem do algoritmo 1.

Comunicação entre processos

A comunicação entre processos ou *Inter-Process Communication* (IPC) é um sistema que permite que tarefas sendo executadas troquem informações entre si. Um exemplo seria um planejador de trajetórias passando *set points* para um controlador PID, que repassa os dados de controle para um *device driver* responsável por acionar o motor sendo controlado.

Existem duas formas mais conhecidas em sistemas de tempo real para trocar informações entre tarefas. A primeira forma consiste em memória compartilhada, que basicamente é uma região de memória onde todos os programas podem ler e escrever. Qualquer programa que deseje mandar mensagem para outro pode disponibilizar esta informação na memória compartilhada e o outro programa irá obter os dados desta região de memória compartilhada. Como esta técnica pode causar corrompimento de dados e falha, já que não se tem muito controle sobre o que os programas estão escrevendo, é comum utilizar mecanismos de sincronização, como semáforos para controlar o acesso a estas regiões de memória. O motivo desta técnica ser muito usada, é porque ela é a forma mais rápida e eficiente de trocar informações entre tarefas, não importando a quantia ou volume de informações.

A outra maneira é chamada de passagem de mensagens. Todos os programas que desejarem enviar informações, devem fazer uma chamada para o *kernel* passando a mensagem desejada com um identificador. O programa que desejar ter acesso a estas informações deve fazer uma outra chamada para o *kernel* para recuperar estas informações. Embora muito mais seguro que o anterior, este método é mais lento e a quantia de dados também é limitada.

4.5.2 Escalonamento de Tempo Real

O escalonador é o *software* responsável por determinar quando e por quanto tempo uma tarefa terá o tempo do processador disponível para sua utilização.

Antes mesmo de realizar um escalonamento, é preciso analisar se um conjunto de tarefas podem ser escalonadas por um sistema. Esta análise pode ser feita da seguinte forma [Tanenbaum, 2001]:

$$\sum_{i=1}^{m} \frac{Ci}{Pi} \leq 1$$

Onde m é o número de processos. Ci/Pi é a fração de tempo que o processo i usa da CPU. Se a soma for 1, obtém-se 100% de uso da CPU.

Com 30 anos de pesquisa sobre escalonamento de tempo real, sabe-se que com restrições de exclusão mútua é impossível encontrar um escalonador ótimo em tempo de execução, bem como a operação de decidir se é possível escalonar um conjunto de tarefas periódicas que usam semáforos para obter exclusão mútua é um problema *NP-Difícil* [Laplante, 2004, Stankovic et al., 1995]. Na verdade os problemas mais importantes do escalonamento de tempo real possuem limitações práticas excessivas ou possuem complexidade *NP-Completa* ou *NP-Difícil* [Laplante, 2004].

Os termos *NP-Completa* ou *NP-Difícil* referem-se a formas que a ciência da computação usa para classificar a complexidade de problemas. Um problema é considerado polinomial (P) quando ele pode ser resolvido em um tempo máximo imposto por uma função polinomial, como por exemplo $100 * n^{100}$ [Guimarães, 2008]. Um problema é definido como da classe NP quando seu menor tempo limite de resolução é uma função polinomial, mas atualmente só existem soluções com tempo exponencial, como por exemplo, 100^n, que é bastante lento. NP é uma abreviação para polinomial não determinístico, enquanto um problema NP-Completo, também chamado de intratável, consiste de um problema NP que não possui nenhum outro problema em NP mais difícil de ser resolvido [Feofiloff, 2008]. Finalmente, um problema NP-Difícil (*NP-Hard*) é aquele em que o algoritmo que consegue resolvê-lo também é capaz de resolver qualquer problema da classe NP.

Mesmo com todas as suas limitações, os algoritmos descritos a seguir são bastante confiáveis e utilizados em diversas situações práticas. O *Rate Monotonic* por exemplo, foi escolhido para implementar um sistema avançado de automação da estação espacial *Freedom* [Stankovic et al., 1995], que depois acabou tornando-se a estação espacial internacional.

Contudo, vale ressaltar que estes escalonadores supõem que as tarefas não se comunicam entre si e não dependem umas das outras, além de supor que todas as tarefas são periódicas. Em 1995, Stankovic sugere que ainda existem muitas lacunas na pesquisa de escalonamento de tempo real, citando o exemplo do jato militar A-7E que possui 75 tarefas periódicas e 172 aperiócias com requisitos significativos de sincronização. A sugestão seria estender o *Rate Monotonic* para integrar tarefas periódicas e não periódicas [Stankovic et al., 1995].

4.5.3 Escalonamento Rate Monotonic

O escalonamento de taxa monotônica proposto por Liu e Layland [Liu e Layland, 1973] foi um marco na área de sistemas de tempo real. Neste sistema de escalonamento de tarefas deve-se determinar as prioridades das tarefas de acordo com seus períodos de execução, de forma que as tarefas com menores períodos têm maior prioridade. De acordo com os autores, esta implementação leva a um algoritmo de escalonamento ótimo.

Liu e Layland provaram matematicamente o funcionamento deste algoritmo e ainda forneceram informações importantes para sistemas de tempo real. Através de uma análise formal, é possível provar

que se a porcentagem de uso do processador estiver abaixo de 69%, o escalonamento sempre atenderá aos prazos, independente do número de tarefas [Liu e Layland, 1973]. Com isto, pode-se construir a tabela 3.

Uso da CPU (%)	Condição do escalonamento
0-25	Poder de processamento excessivo
26-50	Muito Seguro
51-68	Seguro
69	Limite Teórico
70-82	Questionável
83-99	Perigoso
> 100	Sobrecarregado

Tabela 3: Percentagem de uso da CPU versus confiabilidade do escalonamento. Adaptado de [Laplante, 2004]

O *Rate Monotonic*, embora interessante, baseia-se em várias premissas [Tanenbaum, 2001]:

1. Cada processo periódico deve terminar dentro de seu período;
2. Nenhum processo é dependente de outro processo;
3. Cada processo precisa da mesma quantia de CPU a cada uso;
4. Qualquer processo não periódico não tem *deadline*;
5. A preempção dos processos ocorre instantaneamente e não possui *overhead*.

Recentemente Wolf afirmou que infelizmente ainda não possuímos uma boa compreensão teórica de sistemas de tempo real, mesmo havendo alguma teoria útil, como o escalonamento *rate-monotonic* [Wolf, 2007b].

4.5.4 Escalonamento Earliest Dealine First

O escalonador "Primeiro o tempo limite mais próximo" ou *Earliest Deadline First* (EDF) é um algoritmo dinâmico, que preocupa-se com prazos limite (*deadlines*) e não com tempos de execução. O algoritmo sempre busca a tarefa com maior urgência e a escolhe para execução.

A tarefa pronta com o *deadline* mais próximo é sempre a de maior prioridade, em qualquer momento no tempo. Enquanto o *Rate Monotonic* garante o escalonamento apenas se a carga de utilização do processador for menor que 69%, o EDF atinge bons resultados para qualquer valor abaixo de 1 (100%). Embora o EDF seja mais flexível, o RM é mais previsível, já que o escalonamento é estático e não muda dinamicamente. Contudo, em condições de sobrecarga (acima de 100% de CPU), o EDF também pode perder *deadlines* [Laplante, 2004].

Muitos RTOS oferecem para o projetista e programador a opção de escolher qual tipo de escalonamento o sistema operacional irá utilizar para as tarefas.

4.5.5 Deadlocks

Um *deadlock* ocorre em uma situação onde várias tarefas concorrem pelo uso de um mesmo recurso. Ao tentar utilizar recursos de forma simultânea o sistema acaba entrando em uma situação bloqueante em que nenhuma das tarefas consegue continuar sua execução.

Um exemplo bastante simples consiste em um sistema embarcado que possui um *display* e um teclado numérico. Neste sistema hipotético, uma tarefa A é disparada pelo sistema operacional periodicamente a partir de um *timer*, sendo que esta tarefa mostra uma pergunta no *display* e aguarda uma resposta via teclado. Para fazer isto com segurança, tal tarefa utiliza um semáforo para garantir seu acesso exclusivo ao *display* e em seguida, outro semáforo para garantir seu acesso exclusivo ao *display* enquanto aguarda a resposta do usuário.

Uma tarefa B é disparada automaticamente via uma interrupção sempre que uma tecla é pressionada no teclado. Esta tarefa utiliza primeiramente o semáforo do teclado para garantir que tenha controle total sobre o teclado, e em seguida, utiliza o semáforo do *display* para escrever neste tudo que o usuário está digitando.

Ocasionalmente, o usuário pode apertar um botão do teclado e iniciar a tarefa B ao mesmo tempo que a tarefa A estava começando a ser executada. Como elas alocam os semáforos de forma inversa, a tarefa A irá alocar o semáforo do *display* primeiramente. Enquanto isto, a tarefa B aloca o semáforo do teclado primeiramente. Quando tanto a tarefa A quanto a B tentarem alocar o próximo semáforo, ambas irão ficar esperando a liberação dos recursos indefinidamente, pois uma depende da outra, e vice-versa.

Como os *deadlocks* ocorrem raramente, muitos projetistas adotam o algoritmo da Avestruz [Tanenbaum, 2001], que basicamente consiste de ignorar o erro e fingir que nada aconteceu. Sabe-se que muitos jogos eletrônicos adotam esta solução.

No caso de sistemas de tempo real, uma solução é necessária. Para tanto existem diversos algoritmos que implementam a detecção e recuperação de *deadlocks*. Normalmente estes algoritmos são baseadas em tabelas que possuem uma contagem dos recursos disponíveis no sistema e quais estão em uso por quais tarefas. Entretanto, pode-se previnir *deadlocks* requisitando todos recursos antes de utilizá-los [Labrosse, 2002], que é uma forma relativamente fácil de evitar que o problema ocorra.

4.5.6 Inversão de prioridade

O problema de inversão de prioridades ocorre principalmente em *kernels* de tempo real [Labrosse, 2002]. Supondo a existência de 3 tarefas T1, T2 e T3, com 3 prioridades diferentes sendo $P_{T1} > P_{T2} > P_{T3}$.

Inicialmente, as tarefas T1 e T2 estão suspensas aguardando algum evento ocorrer, enquanto a tarefa 3, de menor prioridade, está sendo executada. Em um determinado momento a tarefa 3 adquire um semáforo para acessar um recurso compartilhado, e começa a executar operações com este recurso compartilhado.

Neste momento o evento que a tarefa 1 estava aguardando ocorre, e o *kernel* imediatamente passa a executar a tarefa T1, já que ela é a tarefa de maior prioridade.

Durante sua execução, a tarefa T1 necessita do recurso compartilhado que já está em uso pela tarefa 3 e tenta adquirir o semáforo. Como o recurso já está em uso, a tarefa é suspensa pelo *kernel* para aguardar o recurso ser liberado pela tarefa T3.

Dessa forma, a tarefa T3 volta a ser executada, até que um evento ocorra e cause uma preempção nesta tarefa para que a tarefa T2 seja executada, pois $P_{T2} > P_{T3}$. Depois da execução de T2, o kernel volta a executar T3, que ao terminar sua execução libera o semáforo, permitindo a execução de T1.

Neste caso, a prioridade da tarefa T1, que é a maior do sistema, foi reduzida a prioridade da tarefa T3, já que está tarefa ficou suspensa aguardando um recurso de T3. A situação foi ainda pior, pois como T3 tem menor prioridade que T2, a tarefa T1 foi obrigada a aguardar o fim da execução tanto de T2 quanto de T3 [Ganssle, 2004]. O quadro descrito consiste de uma situação de inversão de prioridade.

Esta situação de bloqueio é inevitável e uma conseqüência direta do fato de se tentar garantir integridade no acesso dos recursos compartilhados [Burns, 1991]. Uma das soluções para a inversão de prioridade é o protocolo de herança de prioridade [Burns, 1991]. Dessa forma, quando uma tarefa de prioridade mais alta precisa ser executada e possui uma região crítica em uso por uma de menor prioridade, a tarefa de menor prioridade herda sua prioridade, podendo terminar de usar a região crítica e a liberando.

Um caso clássico ocorreu em 1997 na missão *Pathfinder* da NASA para o planeta Marte. Logo que o robô *Sojourner* começou sua operação, uma inversão de prioridade ocorreu entre tarefas que tentavam acessar o barramento do computador, levando um *watchdog timer* de segurança a reiniciar o sistema todo. Por sorte, os engenheiros haviam deixado vários recursos de depuração ativados no *software* final do sistema, e graças a este recurso foi possível resolver o problema remotamente, simplesmente mudando um valor de *FALSE* para *TRUE* na configuração de gerenciamento de prioridades do *vxWorks [Jones, 1997]*.

4.5.7 Gerenciamento de memória

Uma importante tarefa dos sistemas operacionais é o gerenciamento da memória. Isto é feito com o suporte da unidade de gerenciamento de memória (MMU), que tem como principal tarefa estabelecer limites que separam diferentes tarefas e processos [Sutter, 2002]. Como este assunto já foi explorado em vários aspectos ao longo deste capítulo, esta seção é apenas um complemento sobre alocação dinâmica de memória.

A alocação dinâmica de memória, é um recurso através do qual os sistemas operacionais permitem que tarefas aloquem e liberem memória durante sua execução conforme a necessidade. Entretanto, funções típicas utilizadas por programadores como *malloc()* e *free()* são perigosas por não serem determinísticas devido à fragmentação da memória [Ganssle, 2004], além do seu uso não ser recomendado em sistemas

críticos pela norma MISRA-C:2004 da indústria automobilística [MISRA-C-2004, 2004]. A fragmentação de memória ocorre freqüentemente em sistemas embarcados com pouca memória, já que o uso dinâmico das memórias feito pelos programas acaba alocando e liberando diferentes espaços de memória, pequenos blocos de memória livres, que por serem pequenos normalmente não são utilizados, levando a uma memória fragmentada. Ao tentar usar a função *malloc()* o sistema operacional precisa procurar em toda memória por alguma parte onde seja possível alocar o espaço solicitado pelo programador, consumindo muito tempo de processamento em modo *kernel*. Além disto, alguns sistemas operacionais precisam freqüentemente executar uma tarefa chamada de compactação, para diminuir a fragmentação da memória

A documentação de alguns sistemas operacionais de tempo real, como o *FreeRTOS,* sugerem que em algumas arquiteturas o *malloc()* nunca seja usado, dando preferência para variáveis alocadas estaticamente [FreeRTOS, 2003].

4.6 Padronização

A necessidade de sistemas confiáveis levou a indústria e a comunidade científica a estabelecer normas para padronizar os sistemas operacionais de tempo real. Os principais padrões são o POSIX de tempo real, o OSEK da indústria automobilística, o APEX para sistemas aviônicos, e o μITRON para eletrônica de consumo. Os padrões e normais apresentadas a seguir foram consultados em [Bouyssounouse, 2005].

4.6.1 POSIX

O POSIX (*Portable Operating Systems Interface*) é um padrão baseado no sistema operacional Unix, almejando portabilidade entre diversos sistemas operacionais. Um programa escrito usando padrões POSIX, poderia simplesmente ser recompilado para poder ser executado em qualquer sistema operacional que implemente este padrão. Muitos sistemas operacionais de tempo real, como QNX, VxWorks e LynxOS implementam o padrão POSIX.

Uma extensão do padrão, o POSIX 1003.1B espeficica várias regras relacionadas a sistemas de tempo real [Obenland, 2001] e também é conhecido como RT-POSIX [Obeland, 2001]. Este foi o primeiro padrão a trazer portabilidade para programas de tempo real, incluindo mecanismos de sincronização, herança de prioridades e garantia de tempos de resposta. Um programa de tempo real escrito nestes padrões poderia ser executado em qualquer sistema operacional compatível com POSIX sem a necessidade de alterações.

4.6.2 OSEK

O OSEK/VDX é uma abreviação em alemão (*Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug*) que significa Sistemas e interfaces abertas para eletrônica automotiva.

Este padrão aberto prevê o controle distribuído de veículos. O padrão também prevê portabilidade

entre diferentes processadores indo de 8 a 32 bits, configuração facilitada (na qual se adapta o sistema operacional facilmente para diversas situações), alocação estática das tarefas e memória (como visto na seção 4.5.7) e eventos que podem ser acionados com o tempo.

4.6.3 APEX

O padrão APEX, foi criado pela ARINC (*Aeronautical Radio, Incorporated*) e quer dizer Interface para *software* de aplicação aviônica (*Avionics Application Software Standard Interface*). O APEX permite analisar *softwares* "*safety critical*" de tempo real, certificá-los e executar estes *softwares*. Sua especificação é bastante completa e formal, abrangendo detalhes como a forma com que computadores de um avião devem trocar mensagens entre si, como o sistema operacional deve tratar e separar tarefas e como o escalonamento de tarefas deve ser realizado.

Um dos seus objetivos é dar segurança para executar várias tarefas em um único computador, tendo em vista que normas federais anteriores dos Estados Unidos obrigavam cada sub unidade a ter seu computador isolado, aumentando muito os custos, peso e uso de espaço físico. Já existem casos de sucesso onde sistemas de tempo real baseados em RTOS foram construídos e certificados de acordo com o padrão APEX, como por exemplo, os sistemas críticos do Boeing 777 [Binns, 2001].

4.6.4 μITRON

A especificação μITRON foi desenvolvida no Japão para padronizar sistemas operacionais de tempo real utilizados na construção de sistemas embarcados. A maior parte das empresas e desenvolvedores que utilizam esta padronização são japoneses. Em um estudo realizado no Japão em 2000, 36% dos produtos embarcados desenvolvidos no Japão pelos programadores que responderam à pesquisa utilizam algum sistema operacional baseado no padrão μITRON [Tron, 1998].

Neste capítulo foi analisado como um sistema operacional de tempo real funciona, bem como recursos que o *hardware* dos computadores disponibiliza para que o sistema operacional desempenhe suas tarefas com qualidade e determinismo. Como foi apresentado, as interrupções têm um papel fundamental para coordenar a execução das tarefas através do escalonador e manter o *clock tick* do sistema.

A seguir será feita uma análise experimental de vários aspectos de sistemas operacionais de tempo real relacionados às interrupções e seus comportamentos temporais.

5 Programação de sistemas de tempo real

5.1 Linguagens de programação

Embora existam linguagens de programação específicas para implementar sistemas de tempo real com bastante formalismo [Laplante, 2004], sabe-se que a maior parte dos sistemas de tempo real são feitos em linguagem C. Um estudo realizado em 2007, demonstrou que 63% dos sistemas embarcados são desenvolvidos em linguagem C, 22% em C++, e o restante em outras linguagens que incluem Java, Basic, Matlab e Labview [Nass, 2007].

Embora já tenha-se discutido se o C e o C++ são as linguagens ideais para sistemas de missão crítica, alguns dos *softwares* mais confiáveis do mundo estão escritos nestas linguagens [Hatton, 2007]. No caso da linguagem C, muitos erros podem ocorrer devido a grande flexibilidade da linguagem, de forma que é recomendável implementar sistemas de tempo real em C utilizando normas formais, como a MISRA-C, que é uma norma para programação de sistemas críticos de veículos elaborada pela associação para confiabilidade de *software* em carros (*Motor Industry Software Reliability Association*).

Uma outra linguagem, chamada de Java, foi criada no início da década de 90 para uso em aplicações embarcadas, como controles remotos e fornos de microondas. O uso de uma máquina virtual, oferece uma abstração entre o *hardware* e os programas, de forma que o mesmo programa pode ser executado nos mais diferentes processadores e *hardwares* graças a esta camada de abstração. Ironicamente, estes mecanismos de portabilidade aumentam o *overhead* e diminuem o determinismo do Java, de forma que os programadores de sistemas de tempo real têm receio no uso da linguagem Java. Além disto, sua licença [Sun MicroSystems, 2007] menciona por exemplo, que o Java não pode ser usado para projetar, construir, operar ou mesmo dar manutenção em qualquer facilidade nuclear. Entretanto, como a linguagem Java é fortemente "tipada", ela reduz o risco de falhas de programação.

De qualquer forma, a IBM forneceu recentemente um sistema para a marinha americana que utiliza Linux e Java de Tempo real, incluindo *garbage collection*[9] em tempo real. De acordo com Mckenney, esta decisão foi tomada devido a maior produtividade e disponibilidade de programadores Java do que programadores especializados em linguagem C e Ada, que normalmente são usadas em aplicações militares [McKenney, 2008].

O sistema da IBM de *hard real time* possuí 1ms como pior caso de latências para Java com *garbagge collection* e 70 microsegundos para Java sem *garbage collection.* Este segundo caso trata-se da implementação da especificação de Java de Tempo Real (RTSJ) [McKenney, 2008]. O NIST (Instituto Americano de Padrões e Tecnologias) é um dos institutos envolvidos em especificar e disseminar extensões de tempo real para a linguagem Java.

O sistema militar descrito acima é executado em um servidor IBM *Blade*, com 2000 threads (tarefas)

[9] *Garbage Collection:* Um mecanismo da linguagem responsável por eliminar automaticamente da memória variáveis e outros dados que não são mais usados, e não foram destruídos explicitamente pelo programador.

em java divididas em 4 processadores x86 [McKenney, 2008].

5.2 Falácias sobre programação de baixo nível

Alguns programadores acreditam que podem escrever código otimizado, proporcionando uma execução mais rápida e eficiente. Isto é um mito, pois na maioria dos casos o otimizador presente no compilador irá gerar um código binário executável o mais otimizado possível, independente das tentativas manuais de otimização. Além disto, na maioria das vezes, estes códigos acabam tornando-se maiores que os códigos escritos da forma convencional após passarem pelo compilador [Ganssle, 2004].

O interessante da norma MISRA-C 2004 [MISRA-C-2004, 2004], é que ela trata de como o código fonte deve ser escrito, enfatizando que o código deve ser bem organizado e bem documentado, além de ser escrito da forma mais clara possível, para facilitar a manutenção preventiva, corretiva e evolutiva.

5.3 Depuração e testes

Alguns anos atrás, era comum usar componentes com especificação militar ("*mil-spec*") em aplicações críticas, o que causava dependência de fornecedores e maiores custos na construção dos sistemas. Hoje em dia, 90% dos produtos militares de comunicação são baseados em componentes comerciais de prateleira [Sandborn, 2008], que nem sempre passaram por rigorosos testes, tornando importante o papel dos testes, depuração e certificação de um sistema.

A questão da confiabilidade do RTOS faz parte de uma gama muito maior de problemas. Pequenos detalhes podem corromper a memória e conseqüentemente um sistema todo de forma catastrófica. Dois exemplos interessantes são referentes a recentes normas internacionais de fabricação de equipamentos usando solda sem chumbo. Em alguns casos, os componentes *"lead-free"*, usam materiais que podem formar pequenos filamentos com o tempo, causando potenciais curto circuitos [Sandborn, 2008]. Uma usina nuclear em Connecticut foi desligada em 2005 por que este fenômeno ocorreu com um dos diodos de um sistema. Além disto, um satélite de 200 milhões de dólares foi considerado uma perda total em 2000 por causa de pequenos filamentos que surgiram em um processador importante do sistema [Sandborn, 2008]. Estas conseqüências inesperadas reafirmam a forte necessidade de constantes e rigorosos testes nos sistemas, e de fato, as fases de testes e depuração normalmente são as mais longas de um projeto.

Esta divisão não apenas ocorre na prática, mas também faz parte de normas de certificação de sistemas de tempo real. O desenvolvimento de *software* crítico para uso em aviação pelas regras da norma DO-178B deve ser balizado pelo uso de ferramentas que contemplam RTOS, BSP, *drivers,* aplicações e matemática. Esta norma sugere que na divisão do tempo do projeto, 10% seja alocado para levantamento de requisitos, 10% para projeto, 25% do tempo em implementação e 35% em testes, sendo o restante distribuído em tarefas mais simples [Amianti, 2008]. Com relação aos testes e depuração, alguns estudos recentes demonstram que esta fase acaba consumindo de 24% até 50% do tempo total de desenvolvimento [Parab

et al., 2007, Nass, 2007].

Outra dificuldade na área de tempo real, é que a depuração de sistemas de controle normalmente tem que ser feita "*on the fly*", enquanto os atuais métodos de *debug* consistem em "parar o computador" [Wolf, 2007b]. Em algum momento é necessário plugar o sistema a ser testado no sistema de controle, e ocasionalmente erros que não eram esperados em teoria podem vir a ocorrer.

Ainda com relação aos testes, pode-se citar o caso da aviação, onde todo *software* desenvolvido para um avião precisa ser certificado pelo padrão RTCA DO-178-B. Esta norma requer que seja demonstrado que toda condição que possa implicar em um salto (desvio condicional em um programa) seja testada para condições falsa e verdadeira em conjunto com todas outras condições que poderiam afetar este desvio [Knight, 2007]. Executar manualmente este tipo de teste é pouco viável, de forma que é necessário apoiar-se em ferramentas especializadas de teste automático.

Com relação aos sistemas operacionais de tempo real, também é desejável que os fabricantes executem testes sistemáticos em seus sistemas, o que de fato ocorre. O processo de desenvolvimento do RTLinux da FSMLabs, por exemplo, era bastante interessante antes da aquisição da propriedade intelectual do produto pela *Wind River*. Um sistema de testes automático compilava diariamente cada uma das 15 versões do sistema para diferentes arquiteturas, e executava diversos testes de confiabilidade e qualidade diretamente em placas de cada arquitetura [Dougan, 2008]. Destes testes, 350 eram relacionados a requisitos de tempo real, como latência de tratamento de interrupções, *jitter*, tempo de aquisição de semáforos, tempo de troca de contexto, dentre outros, oferecendo confiabilidade ao produto final, e um histórico de características de tempo real que ficava armazenado em um banco de dados para cada versão do software [Dougan e Mwaikambo, 2004]. Neste processo, todos equipamentos de teste eram ligados automaticamente a noite, os testes eram executados, e enviados para os responsáveis automaticamente.

6 Ferramentas Desenvolvidas

Além dos resultados experimentais obtidos que serão discutidos a seguir, outros resultados também foram obtidos com relação à infraestrutura do laboratório de mecatrônica e colaboração para a comunidade de *software* livre. Estes resultados adicionais serão brevemente discutidos neste capítulo.

6.1 Robô Scara

No decorrer desta pesquisa, um trabalho realizado em conjunto com os colegas de laboratório Dalton Matsuo Tavares e Leonardo Marquez Pedro possibilitou tornar funcional um antigo robô da IBM do tipo SCARA, que estava inoperante devido a problemas nas placas eletrônicas de controle. Mesmo que estas placas de controle estivessem funcionais, seria difícil realizar qualquer tipo de pesquisa de alto nível, já que o robô possui uma arquitetura fechada, sem acesso aos seus detalhes de implementação. O trabalho realizado possuiu uma natureza bastante multidisciplinar que envolveu sistemas de controle, mecânica, eletrônica, computação e sistemas de tempo real.

Parte deste trabalho já havia sido desenvolvido previamente por um membro do laboratório (Marquez) em seu trabalho de conclusão de curso. O documento criado por Marquez foi utilizado como ponto de partida, onde o mapeamento de sensores e atuadores já havia sido realizado, e o projeto de uma placa para controle dos motores também já havia sido inclusive testado.

Para realizar o controle do sistema, um PC industrial foi utilizado com componentes consolidados na área de robótica e automação industrial.

6.1.1 O robô SCARA

SCARA é um sigla para *Selective Compliant Assembly Robot Arm*, mas também pode-se encontrar a sigla com o significado de *Selective Compliant Articulated Robot Arm*. Em geral os robôs SCARA são braços robóticos de 4 eixos, podendo se mover para qualquer ponto X, Y, Z dentro de sua área de trabalho (*workspace*). O quarto eixo de movimento consiste na rotação do punho (ângulo θ-Z). Os movimentos em X, Y e θ-Z são obtidos com 3 juntas rotatórias paralelas, e o movimento vertical normalmente consiste de um eixo independente de deslocamento linear. O robô SCARA existente no laboratório é um IBM 7545, sendo que sua fabricação foi realizada por uma empresa chamada *Sankyo* sob encomenda da IBM .

Os robôs SCARA são normalmente utilizados em tarefas de montagem na indústria, e são reconhecidos por usualmente serem mais rápidos que sistemas cartesianos semelhantes. A montagem baseada em um único pedestal requer pouco espaço e facilita a instalação do robô. Por outro lado, o SCARA pode ser mais caro que outros sistemas cartesianos e o *software* de controle precisa calcular a cinemática inversa do robô para movimentos lineares.

O primeiro motor é acoplado a um redutor harmônico com alto grau de redução que está ligado ao primeiro braço do robô, ou ombro. O segundo motor está ligado diretamente ao eixo do segundo braço,

na configuração chamada de *Direct Drive*. O terceiro motor é responsável pelo controle de altura do eixo Z através de uma rosca sem fim que é movimentada pelo eixo do motor. E o quarto motor é responsável por girar o *end effector* do robô.

Os quatro motores são conectados a *drivers* de potência que são alimentados diretamente por 220V derivados da fonte principal do robô, passando por um fusível térmico que desarma automaticamente o sistema com excesso de temperatura, e envia um sinal para o computador de controle.

Através deste *drivers*, é possível realizar um controle de velocidade independente em cada motor, ou manter o motor parado. Para realizar este controle, é necessário aplicar um sinal PWM que deve ter a freqüência fixa em pelo menos 10KHz, e a amplitude entre -12Vcc e +12Vcc. Todos motores possuem *encoders* incrementais embutidos, que fornecem 500 pulsos por revolução do eixo, totalizando 0,72 pulsos por grau sem contar reduções.

Com relação aos sensores de fim de curso, o robô possui um total de 12, sendo 3 ópticos no controle de altura do eixo Z, 3 indutivos no controle de posição rotatório do segundo braço, 3 ópticos no controle do primeiro braço e 3 ópticos no controle da rotação do TCP. Todos estes sensores usam lógica de 24 volts, devendo ser alimentados com 24v. Os sensores fornecem o sinal de 24 volts na saída quando abertos, e aterram o sinal quando fechados. A única exceção está nos sensores indutivos onde a lógica de saída é inversa à apresentada.

6.1.2 Interfaceamento eletrônico

O interfacemento eletrônico entre subsistemas do robô e o PC industrial teve a principal preocupação de isolar totalmente o Robô do PC, protegendo ambos dispositivos de surtos de tensão e outros transientes que podem ocorrer em qualquer um dos lados do sistema. Para tanto todas linhas de comunicação entre robô e PC são isoladas opticamente nos dois sentidos.

A primeira interface óptica desenvolvida foi a interface para os *encoders,* que trabalham com níveis de tensão de 12V. Dessa forma, os 12V provenientes de cada canal dos encoders foram ligados a resistores na entrada de acopladores ópticos modelo CNY-17. Ao fazer os primeiros testes com este sistema de acoplamento, muitas leituras incorretas ocorreram, e depois de diversos testes, foi necessário adicionar uma porta inversora (74LS14) com *schimit trigger,* que solucionou por completo o problema de leituras incorretas. Esta porta tem característica de histerese, oferecendo uma onda perfeitamente quadrada em sua saída. Cada encoder precisa de pelo menos 2 destes circuitos, pois possui 2 canais (A e B). É através da diferença de fase entre os canais que é possível detectar a direção em que o sistema está girando.

A saída deste circuito foi ligada diretamente a uma placa especial para leitura de *encoders* no PC Industrial utilizado para controlar o robô, chamada de IP-QUADRATURE, de forma que a montagem descrita permite saber qual é posição em graus de cada eixo do robô.

Para realizar o controle dos motores, uma saída do PC industrial através do IP-OPTODAQ foi uti-

lizada para gerar um sinal de tensão analógico que é utilizado como referência para um circuito gerador de PWM. A saída deste é por sua vez conectada à entrada TG dos *drivers* de potência do robô. Vários circuitos que geram PWM foram analisados no trabalho de conclusão de curso de Marquez, e o circuito escolhido foi o baseado no circuito integrado discreto TL494, que é capaz de gerar um PWM de saída com base numa tensão variável em um pino de entrada, podendo variar a sua freqüência em grandes valores através de ajustes em um circuito RC.

Esta parte do circuito gera um sinal de PWM variando de 0V a 5V com freqüência de 10KHz. Este sinal é inserido em um acoplador óptico CNY-17, de forma a isolar os circuitos, e a saída do acoplador óptico é utilizada como referência para um amplificador operacional de alta freqüência, que foi usado com a função de gerar a tensão de -12V em sua saída quando o PWM está em 0 e a tensão de +12V na saída quando o PWM esta em 5V. Este sinal de -12V/+12V é ligado diretamente ao *driver* de potência, e com isto consegue-se controlar a velocidade do motor, variando a tensão por comandos no PC industrial.

Um terceiro circuito de acoplamento foi implementado para conectar os sensores de fim de curso a entradas digitais do PC industrial através da placa de entrada e saída IP-DIO. Esta placa de entrada e saída também foi utilizada para acionar reles e contatores que foram conectados aos *drivers* de potência, permitindo ligar e desligar via *software* o bloco de potência do sistema, permitindo a parada emergencial do sistema.

6.1.3 Computador Industrial

O computador industrial utilizado foi um Inova PC com processador AMD K6-II de 500MHz e 64MB de memória RAM. O computador não possui memória *flash* ou outros tipos de disco. Em geral os computadores industriais são baseados ou no barramento VME[10] ou no barramento *CompactPCI*[11], sendo o Inova composto por um *rack CompactPCI*. A placa de CPU é inserida no *rack*, e através do *backplane* troca informações com as outras placas e dispositivos do sistema.

Para conectar este computador industrial ao robô foram utilizados *carrier boards* da Acromag modelo APC8620 e *industry packs* da SBS, que hoje é uma empresa do grupo GE-FANUC. Cada *carrier* tem 2 *slots* para conexão de *Industry Packs,* que são pequenas placas com algum tipo de lógica implementada para serem usadas em determinada aplicação. A *carrier* faz com que o PC industrial possa conversar com os *Industry Packs* (IPs), e permite que os IPs conversem com o mundo externo através de conectores localizados no painel frontal da *carrier* que podem ser ligados a borneiras para acessar os pinos de I/O do IP. Todas estas conexões podem ser vistas na figura 14.

Na primeira *carrier* foram instalados dois *industry packs:*

[10] Maiores informações em: http://www.vita.com/learn.html
[11] Maiores detalhes em: http://www.picmg.org/test/compci.htm

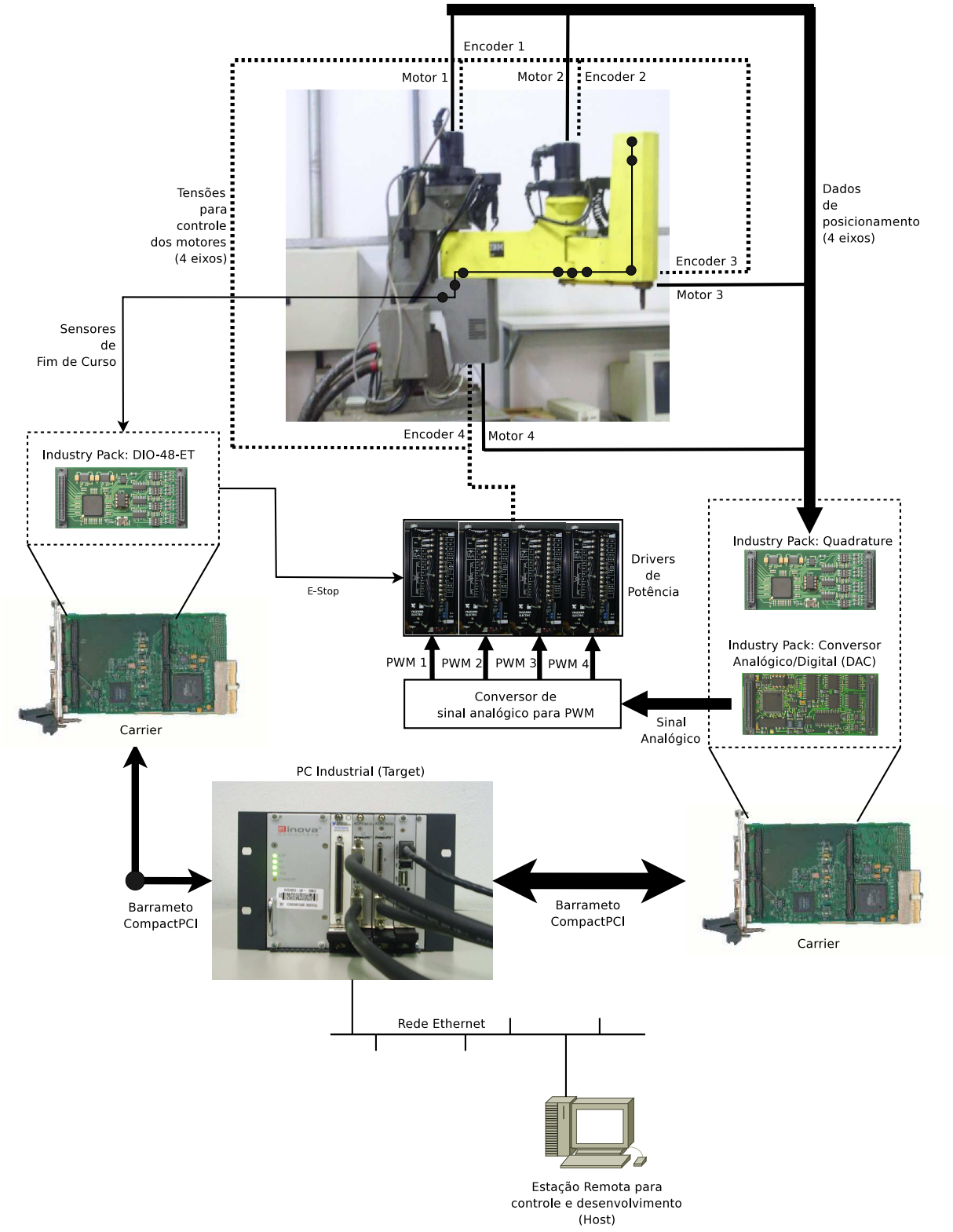


Figura 14: Arquitetura geral de *hardware* do robô SCARA

1. IP-QUADRATURE: Cada IP deste tipo pode fazer a leitura de até 4 *encoders* em diversos modos. O modo utilizado no robô SCARA, foi o *quadrature*, em que o IP multiplica por 4 os pulsos recebidos do *encoder* instalado no motor, aumentando a resolução de medida de posição para 0,18 pulsos para cada grau;

2. IP-OPTODAQ: Este IP possui entradas e saídas analógicas acopladas opticamente. No caso do robô Scara, foram utilizadas 4 saídas do IP-OPTODAQ que podem variar de -10vcc a +10vcc com resolução de 12bits para controlar a velocidade dos motores.

Na segunda *carrier*, apenas um *industry pack* foi instalado:

- IP-DIO48-ET: Este *industry pack* possui 48 linhas de entrada e saída digital que podem ser configuradas independentemente para gerar interrupções em eventos de mudança de estados, o que é bastante interessante para implementar sensores de fim de curso. Este IP está sendo utilizado tanto para monitorar sensores de fim de curso do robô, que também são utilizados para auxiliar na calibração do robô quando ele é ligado. Outro uso deste IP é para acionar o rele e contator que acionam o modo de emergência do sistema, desligado o bloco de potência do robô.

6.1.4 Sistema de Tempo Real

Para implementar o sistema de controle, optou-se pelo uso do Linux de tempo real RTAI, por tratar-se de um sistema aberto e com bom determinismo, já que buscava-se construir uma arquitetura flexível para realizar pesquisas tanto na área de robótica quanto de tempo real.

Para implementar o sistema, foi necessário desenvolver um sistema de *boot* remoto, onde toda vez que o robô é ligado, o computador industrial descarrega via rede pelo protocolo TFTP o *kernel* do sistema operacional e a imagem do sistema de arquivos em utilização. Este cenário também permite facilmente executar outros sistemas no ambiente, bastando trocar os arquivos que estão no servidor. Para tanto uma ferramenta chamada *etherboot* foi utilizada.

Uma tarefa que consumiu certo tempo, foi a implementação dos *device drivers* para os *industry packs*, pois nenhum deles possuía *drivers* para Linux. Uma vez concluída a implementação dos *device drivers*, foi possível realizar os primeiros testes em malha aberta, enviando tensão para os motores, e verificando sua movimentação e a leitura desta movimentação pelos dados informados pelos *encoders*.

A próxima etapa consistiu em compilar um *kernel* do Linux com o *patch* do RTAI, apenas com os recursos essenciais ao sistema. Já com o sistema operacional de tempo real baseado em RTAI em operação, Marquez levantou as plantas dos motores, e foi possível implementar controladores PID em malha fechada para controlar o robô de forma bastante confiável. As tarefas de PID foram implementadas com intervalo de execução de 1ms em *kernel space*, com requisitos de *hard real time*. Posteriormente foram implementados sistemas de calibração e de leitura dos sensores de fim de curso.

Como a malha de controle é executada em espaço de *kernel,* foi necessário implementar um FIFO de tempo real para que um aplicativo de interface com o usuário pudesse ser implementado. A primeira versão deste aplicativo podia ser acessada via *telnet*, e recebia como parâmetro os ângulos desejados das juntas do robô, movimentando o robô para o ponto solicitado, e mantendo-o naquela posição.

Numa versão posterior, o programa evoluiu para um servidor TCP/IP, de forma que foi possível controlar o robô via rede por meio de outros programas. Marquez implementou um planejador de trajetórias via Matlab, onde foi possível controlar o robô via rede a partir do Matlab.

Em uma outra etapa, Tavares implementou um sistema de controle baseado em um *framework* para robótica chamado *Open Robot Control System* (OROCOS), que consiste de uma plataforma para controle em tempo real de sistemas mecatrônicos [Bruyninckx, 2002], incluindo filtros de Kalman, cálculo de cinemática direta e inversa e execução de outras tarefas comuns para controle de robôs.

O sistema operacional implementado para o robô SCARA trata-se de um sistema embarcado, de forma que ao invés de utilizar-se ferramentas convencionais do Linux, optou-se por um conjunto de ferramentas simplificadas típicas de sistemas embarcados. Estas ferramentas estão disponíveis em um pacote chamado *busybox*.

A imagem do sistema operacional completo ocupou apenas 6MB e vem sendo usado com confiabilidade e determinismo.

6.1.5 Interface com o usuário

Para operar o robô, diversas interfaces com o usuário foram implementadas. Uma dela foi baseada em tecnologia *Asynchronous Javascript And XML (AJAX)* permitindo qualquer computador ou dispositivo controlar o robô remotamente [Aroca et al., 2007]. De fato, após a implementação do sistema, foi possível controlar o robô a partir de PDAs conectados via rede sem fio *wifi,* ou qualquer computador via *internet.* Para operação via *internet*, uma *webcam* foi disponibilizada no laboratório e um botão de parada de emergência instalado tanto fisicamente no laboratório, quanto na página de controle do robô.

Outro projeto interessante foi implementado por Martins [Junior et al., 2008], onde é possível visualizar o robô SCARA em um ambiente 3D, e os movimentos realizados no ambiente 3D são repassados via TCP/IP para o robô, que reproduz os movimentos.

Planos futuros incluem controlar o robô por comandos de voz, além de um sistema de acesso telefônico, onde será possível telefonar para o robô SCARA, e pedir para ele executar alguma tarefa, recebendo uma resposta via voz sintetizada pelo telefone.

6.2 *Mechatronics Real Time Linux*

Um outro fruto desta pesquisa foi um *Live* CD que foi preparado inicialmente com a intenção de dar aulas didáticas para alunos de graduação sobre a construção de sistemas mecatrônicos confiáveis com a

utilização de sistemas operacionais de tempo real.

Este conceito de ensinar alunos de graduação em engenharia mecatrônica a desenvolver produtos já pensando em ferramentas e RTOS, já está sendo utilizado em cursos de graduação em outros países como na Bélgica [Bruyninckx et al., 2001] e nos Estados Unidos [Jacker, 2002]. Os primeiros cursos acadêmicos que ensinam conceitos de sistemas de tempo real usando experimentos práticos com RTOS foram realizados por Sorton [Sorton e Kornecki, 1998] e Kornecki [Kornecki et al., 2000].

O curso de graduação de "Desenvolvimento de Produtos Mecatrônicos", ofereceu aos alunos programas prontos que demonstravam conceitos de tempo real, de forma que os alunos tem o primeiro contato com um programa já funcional, e suas tarefas são feitas, alterando-se estes programas, como proposto em [Sorton e Kornecki, 1998].

Alias, é interessante notar que neste tipo de aula, os alunos ficam bastante interessados, pois normalmente consistem de atividades práticas envolvendo componentes mecânicos, de *hardware* e *software* simultaneamente. Outro assunto estudado e que pode ser ensinado com o uso deste CD, é o desenvolvimento de *device drivers*, que é um tópico que normalmente é ignorado na maioria das universidades [Kornecki et al., 1998], resultando em falta de mão de obra qualificada para fazer camadas de acoplamento entre *hardware* e *software*. O CD também inclui exemplos de um excelente tutorial prático de programação em tempo real do NIST, disponível em [Proctor, 2006].

Até o final de Julho de 2008 o site do projeto tinha recebido cerca de 2100 acessos a partir de 495 cidades. A distribuição dos acessos pode ser visto na figura 15. Destes 2100 acessos, 300 *downloads* do CD foram completados com sucesso.

Figura 15: Mapa de cidades que acessaram o *website* do *Live* CD. Fonte: *Google Analytics*

Na disciplina que já foi oferecida duas vezes pelo orientador deste trabalho, os alunos já implementaram robôs autônomos, malhas PID e sistemas de controle via rede, utilizando componentes de baixo custo conectados à porta paralela do PC, de forma que é possível ensinar conceitos e técnicas importantes de *hardware e software* a um baixo custo.

Outro aspecto interessante, é o surgimento cada vez maior de *Live* CDs de diversos fabricantes. Quando este CD foi implementado, apenas o QNX era conhecido por ter um *Live CD* em que foi possível fazer testes de tempo real. Ainda neste ano a Wind River lançou um *Live CD* baseado em Linux com ferramentas para desenvolvimento e testes do VxWorks, para divulgar e tornar mais fácil que o público conheça e teste o ambiente de tempo real da *Wind River*.

O CD do *Mechatronics Real Time Linux* está disponível gratuitamente no endereço http://www.mecatronica.eesc.usp.br/~aroca/slax-rt/.

7 Análise Experimental

7.1 Comparação de sistemas operacionais

Como os sistemas operacionais podem variar muito em termos de sua arquitetura interna e requisitos de *hardware*, comparar *kernels* não é uma tarefa fácil. Com centenas de *kernels* para se escolher, Ganssle [Ganssle, 2004] propõe que deve-se utilizar uma matriz de comparação para decidir qual a melhor opção. Ganssle recomenda que os seguintes itens sejam analisados: popularidade, reputação, disponibilidade de código fonte, portabilidade, escalabilidade, se o sistema é preemptivo, número máximo de tarefas suportadas, tamanho do *stack* (pilha de memória), serviços disponíveis, performance, certificações recebidas e ferramentas disponíveis.

Laplante [Laplante, 2005] apud [Laplante, 2004], sugere treze métricas de comparação para se selecionar um RTOS. Estes critérios detalham cinco características desejáveis para sistemas de tempo real: temporização adequada, projeto para suportar cargas de pico, previsibilidade, tolerância a falhas e manutenabilidade. Dentre os 13 critérios, o primeiro a ser considerado é a máxima latência das interrupções, estando listado entre os critérios de maior peso para decisão se um RTOS é melhor do que outro.

Já Taurion [Taurion, 2005], separa os critérios para escolher um sistema operacional de tempo real em gerais e técnicos. Os aspectos gerais incluem avaliar se o sistema já é usado no tipo de aplicação que será desenvolvida e se seu fornecedor tem maturidade e suporte técnico eficiente. Também deve ser avaliada a disponibilidade de profissionais que saibam trabalhar com tal sistema operacional, a portabilidade do sistema e a sua política de licenciamento. Com relação aos aspectos técnicos, deve-se questionar sobre a qualidade do ambiente de desenvolvimento disponível, bem como a forma que o sistema operacional gerencia a memória e quais são as formas disponíveis de *boot* deste sistema. Taurion ainda recomenda verificar os recursos de conectividade disponíveis (*bluetooth, IP*, outros) e se o sistema pode ser facilmente reconfigurável. A lista ainda inclui a análise de eficiência dos comandos de entrada/saída e questões sobre o gerenciamento de processos, como o fato do sistema operacional utilizar algoritmos determinísticos.

Já com relação a testes experimentais, a *Dedicated Systems* tem um histórico de boa reputação. Martin Timmerman, um professor da academia militar *Royal* e da universidade de *Vrije*, desenvolve pesquisas com foco em sistemas operacionais de tempo real. Em 1983 fundou um centro de desenvolvimento de sistemas para as forças armadas da Bélgica, e em 1998 fundou a *Dedicated Systems*, que iniciou um projeto de testar sistematicamente sistemas operacionais de tempo real. Alguns dos testes são vendidos na forma de relatório, e outros são disponibilizados gratuitamente. Os relatórios desenvolvidos pela equipe de Timmerman são de alta qualidade e vasto conteúdo técnico. Infelizmente, não se viram mais resultados e relatórios sobre RTOS emitidos pela *Dedicated Systems* nos últimos 5 anos. Um dos documentos que ajudaram a balizar os experimentos realizados neste trabalho de pesquisa é o *Evaluation Report Definition* [Dedicated Systems, 2001b], que descreve em detalhes a metodologia utilizada pela *Dedicated Systems*

para testar sistemas operacionais.

Outro ponto a ser considerado é que uma comparação, quando realizada de forma experimental, deve ser realizada na mesma plataforma de *hardware* para todos sistemas analisados. Segundo Dougan [Dougan e Mwaikambo, 2004], muitos vendedores de RTOS publicam valores de performance que nem sempre mostram a realidade do sistema. O tempo de troca de contexto, segundo ele, é uma soma de tempos de *hardware* e *software*, entretanto alguns fabricantes de RTOS publicam este número com base apenas no tempo de troca de contexto do *hardware*, que é independente do sistema operacional.

7.2 Materiais e Métodos

Em um artigo publicado em 2000 foi constatado que a análise de performance em sistemas operacionais de tempo real não vem recebendo muita atenção. Além disso, as análises realizadas até então foram consideradas extremamente inapropriadas e duvidosas já que medem o tempo médio de resposta, sendo insuficientes para determinar a qualidade de um RTOS [Halang et al., 2000].

A necessidade de avaliar o pior caso de tempo de execução leva a um problema com relação à análise de sistemas operacionais de tempo real, pois somente considera-se o cenário com pior tempo de execução da aplicação - *Worst Case Execution* Time (WCET), ignorando o tempo de execução das rotinas do sistema operacional. Estas rotinas muitas vezes também impactam no tempo de execução do programa de tempo real. Por este motivo não se pode analisar um sistema operacional sem considerar o programa, e não se pode analisar um programa de tempo real sem considerar o sistema operacional. A análise deve ser conduzida em conjunto [Schneider, 2002].

Tal análise comparativa não pode ser feita sem um procedimento de testes adequado [Sacha, 1995, Tsoukarellas et al., 1995], de forma que uma metodologia sistemática será estabelecida e seguida para realizar os testes.

Os dois fatores mais importantes de performance de tempo real são o pior caso no tempo de resposta de uma tarefa e o pior caso no tempo resposta de uma interrupção [Sohal, 2001]. Entretanto, não faz sentido analisar métricas de sistemas operacionais como latências de interrupções e tempo para troca de tarefa, sem considerar diferentes cenários de porcentagem de uso do processador [Timmerman et al., 1998], já que é mais fácil para um sistema ser previsível quando não existe uma situação de sobrecarga.

Existem várias maneiras de analisar o comportamento temporal de um sistema. Uma destas maneiras é chamada de *sonda de software* [Shaw, 2003, Laplante, 2004]. Para fazer este tipo de medida, idealmente as interrupções deveriam ser desabilitadas, para garantir maior precisão na medida do tempo, contudo isto é raramente feito, pois na maioria dos computadores, as interrupções são necessárias para manter o próprio relógio do sistema funcionando, que fornecerá dados para a medida.

A sonda de *software* consiste em medir a diferença de tempo entre o início e o fim da execução de um trecho de código, como pode ser visto na listagem 2. Entretanto este algoritmo possui duas falhas.

A primeira é referente ao *overhead* da execução da função `agora()` (para obter o momento atual), que pode não levar um tempo fixo, e influenciar a medida. A outra falha, é o fato de chamadas de sistema executadas no código de teste poderem não ser executadas sempre com o mesmo intervalo de tempo.

Listagem 2 Proposta de sonda para medida de tempo de execução de um trecho de código

```
início = agora()
Execução do código em teste
fim = agora()
tempo de execução = fim - início
```

Dessa forma, a sonda de software deve ser realizada utilizando-se uma média de diversas medidas, e subtraindo-se o tempo da execução de um trecho de código nulo, como pode ser visto na listagem 3. A subtração do tempo de execução do código nulo, elimina o *overhead* da medida de tempo.

Listagem 3 Medida de tempo de execução de um trecho de código

```
T_execução=0
De 1 até n faça
  início_nulo = agora()
  fim_nulo = agora()
  T_medida = fim_nulo - início_nulo
  início = agora()
  //Execução do código em teste
  fim = agora()
  T_execução = T_execução+(fim - início) - T_medida
Fim
Resultado=T_execução/n
```

Mesmo assim, estabelecer medidas confiáveis pode ser um problema, já que *timers* de *software* não possuem grande precisão [Timmerman et al., 1998]. Além de imprecisos, o simples fato de utilizá-los adiciona imprevisibilidade e *overhead* às medidas [Timmerman et al., 1998].

Uma das explicações desta imprecisão é que na maioria dos PCs, um único oscilador fornece a freqüência de operação do processador e dos *clocks* de entrada dos *timers*, de forma que se este oscilador variar, todos os temporizadores e relógios do PC irão variar juntos, não sendo possível verificar discrepância entre eles [Proctor, 2001]. De fato isto ocorre em praticamente todos os PCs, onde um único cristal de aproximadamente 14MHz alimenta um gerador de *clock* (normalmente o 952018AF ou W83194R) que gera todos os *clocks* do PC, que vão sincronizar as portas USB, o barramento PCI, as pontes e o processador, como pode ser visto na figura 16. A freqüência que este gerador envia para o processador pode chegar a cerca de 500MHz em PCs de última geração. Os processadores só conseguem atingir freqüências de operação de vários gigahertz graças a multiplicadores internos presentes na própria pastilha das CPUs. As freqüências geradas para as pontes e barramentos não foram colocadas no diagrama, pois podem variar de acordo com vários fatores.

Embora seja possível realizar melhorias na sonda de *software* [Hajdukovic et al., 2003], o ideal seria testar um sistema de tempo real como uma caixa preta. No teste de caixa preta, o sistema é analisado sem ser considerado nenhum aspecto interno do sistema, mas apenas seus estímulos de entrada e seus

sinais de saída [Levi e Agrawala, 1990].

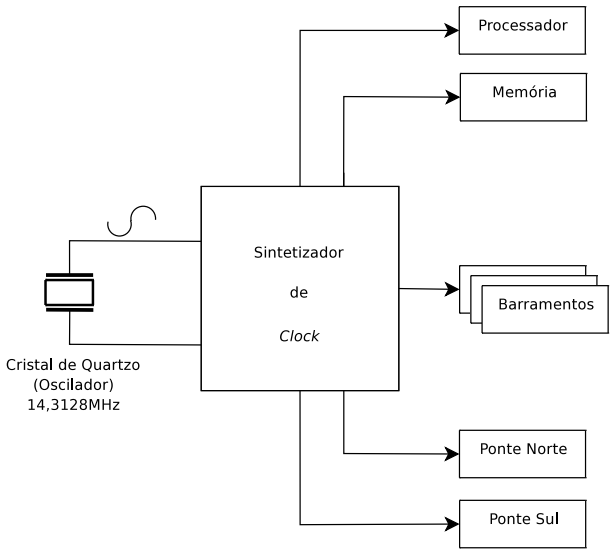


Figura 16: Sintetizador de *clock*. Todas freqüências e tempos são derivadas de um único cristal

Normalmente os pesquisadores utilizam a latência (medida via *software)* e a capacidade de transferência de dados como as métricas primárias para avaliação de desempenho de computadores, através do uso de *softwares* de análise de desempenho como o SPEC[12]. Entretanto, na maioria das aplicações reais os sistemas possuem grandes interações de I/O, que normalmente não são consideradas nestes testes, levando a conclusões incompletas ou incorretas [Pieper et al., 2007]. Dessa forma, a sonda de software será utilizada apenas para realizar uma análise interna do tempo que leva para instruções de entrada/saída ocorrerem.

Pelo fato de eventos importantes e urgentes serem informados ao processador através de interrupções, a latência para o início do tratamento de interrupções é uma métrica freqüentemente utilizada por fornecedores de sistemas operacionais como indicador de qualidade do RTOS [Farines et al., 2000, Köker, 2007]. De acordo com Labrosse [Labrosse, 2002], a especificação mais importante de um sistema de tempo real é a quantia de tempo que as interrupções ficam desligadas, já que a latência das interrupções é um componente do tempo de resposta do sistema [Laplante, 2004]. De acordo com Franke, a medida do tempo de resposta das interrupções externas proporciona uma boa idéia das capacidades de tempo real de um sistema [Franke, 2007].

Uma abordagem proposta em diversos trabalhos [Franke, 2007, Barabanov, 1997, Ganssle, 2004, Köker, 2007, Barbalace et al., 2008] consiste em utilizar a porta paralela do PC para receber uma interrupção e responder a esta interrupção, permitindo analisar o sistema como uma caixa preta. Utilizando um gerador de sinal externo, contadores e um osciloscópio, é possível utilizar este método para obter-se a latência

[12] Maiores informações em http://www.spec.org

de tratamento das interrupções, além do *jitter,* já que uma das formas mais precisas e efetivas de medir o tempo de execução de software é através de portas de saída [Stewart, 2001]. Proctor ainda reforça a afirmação de que testes de latência só podem ser feitos por meios externos [Proctor, 2001]. Taurion [Taurion, 2005], também afirma que as métricas mais comuns para medir a qualidade de sistemas operacionais são o tempo de chaveamento entre dois processos e a latência até o início do tratamento de uma interrupção.

Embora a medida da latência das interrupções através de mecanismos externos seja bastante utilizada, a Sociedade Internacional para Medidas e Controle (*International Society for Measurement and Control - ISA*) através de seu comitê para automação e controle (*Automation & Control Systems Commitee - A&CS*) não recomenda usar a latência como uma medida da resposta de tempo real, pois isto seria observar erros ocasionais no controlador [Dupré e Baracos, 2001].

O teste proposto pelo A&CS consiste em ligar um gerador de sinal de onda quadrada a uma entrada digital, e a um contador. O sistema de tempo real deve ser programado para copiar o sinal de entrada, para o sinal de saída, sendo este sinal conectado a outro contador. Esta configuração é a mesma utilizada para realizar o teste de latência de interrupções.

Um dos contadores vai contar quantos pulsos entraram no sistema sendo analisado, e o outro contador vai contar quantos foram copiados pelo sistema em testes para sua saída. Teoricamente, enquanto o sistema estiver estável, ambos os contadores estarão mostrando o mesmo valor.

Feito isto, deve-se aumentar a freqüência do gerador de pulsos até que o valor dos contadores comece a divergir. Neste momento, deve-se reduzir freqüência paulatinamente até obter-se a máxima freqüência de operação do sistema, que é observada enquanto os contadores estão mostrando o mesmo valor. De acordo com Dupré e Baracos [Dupré e Baracos, 2001], a freqüência obtida é o inverso do pior caso de tempo de resposta.

Tendo em vista estes métodos, os parâmetros quantitativos analisados de cada sistema operacional serão:

1. **Latência:** A latência é analisada externamente tomando o RTOS em testes juntamente com o *hardware* como uma caixa preta. A latência irá consistir da diferença de tempo entre o momento que uma interrupção é gerada, e o momento em que a tarefa associada a esta interrupção gera uma resposta. A latência foi medida em cenários com baixa carga de processamento, e em cenários com o sistema sobrecarregado;

2. **Jitter:** O *jitter* é uma medida indireta obtida a partir de diversas medidas da latência, consistindo de uma variação aleatória entre cada medida de latência. Em redes de comunicação, o *jitter* é definido pela RFC4689 como sendo a variação da latência de um pacote de dados para outro [Communications, 2007]. Por exemplo, um primeiro pacote de dados tem latência de 18ms e um segundo pacote tem latência de 15ms. Neste caso, o *jitter* é de 3ms. Em sistemas operacionais de tempo real, o *jitter* pode ter impacto notável, como é analisado por Proctor, ao tentar controlar um

motor de passo, por exemplo. A duração dos pulsos determina a rotação do motor, e o *jitter* faz com que o torque do motor varie, ocorrendo uma perda de "passos" [Proctor e Shackleford, 2001]. Para medir o *jitter*, calcula-se a diferença de tempo entre duas leituras de latência de interrupções, para cada medida realizada. Em seguida, seleciona-se a maior diferença calculada, que consiste do maior *jitter* do sistema. Posteriormente será analisado se os valores obtidos são compatíveis com aqueles sugeridos pela OMAC para considerar um sistema *hard real time*, que consiste de sua variação de latência (*jitter*) não ultrapassar $100\mu s$, em tarefas de ciclos de até 10ms [Hatch, 2006];

3. **Pior caso de tempo de resposta:** O pior caso de tempo de resposta será analisado pelo método proposto por Dupré e Baracos [Dupré e Baracos, 2001], através da análise do máximo valor de freqüência de interrupções que é atendida pelo sistema operacional. O teste será realizado em situações de baixa carga de processamento e de sobrecarga de processamento.

4. **Tempo para executar funções de entrada e saída:** Como a maior parte dos sistemas de tempo real possuem alguma forma de interação com o ambiente externo, é necessário que o computador realize operações de entrada e saída. Dentre as diversas opções para realizar este tipo de operação, uma delas é o acesso direto às portas do *hardware* através das funções *IN* e *OUT*. A análise do tempo que o sistema operacional leva para realizar estas operações também será realizada.

7.2.1 Procedimento de testes

Para realizar os testes foram utilizados os seguintes equipamentos:

1. Osciloscópio Minipa MO-310 de 100MHz com comunicação RS-232;
2. Gerador de sinais Minipa MFG-4202;
3. PC com processador Pentium II 400MHz, 256MB de memória;
4. PC com processador Celeron 700MHz, 128MB de memória;
5. PC com processador Pentium MMX 150MHz, 48MB de memória;
6. PC com processador Athlon 1.2GHz, 512MB de memória.

A interconexão dos equipamentos para realizar os experimento pode ser vista na figura 17.

O procedimento de testes para obter a latência e o *jitter* consiste em executar as seguintes operações:

1. Implementar uma ISR no sistema operacional a ser analisado, que ao receber uma interrupção, inverte o estado de uma saída digital (de 0 para 1 e de 1 para 0);

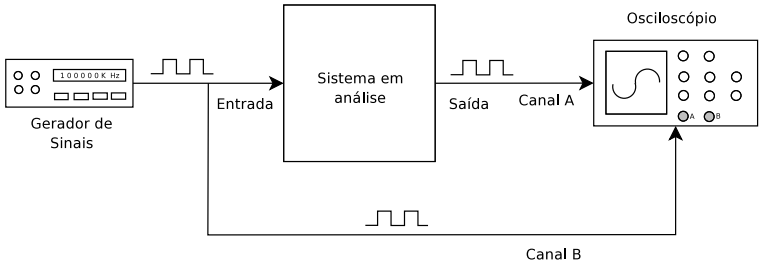


Figura 17: Conexão dos equipamentos para realizar os experimentos

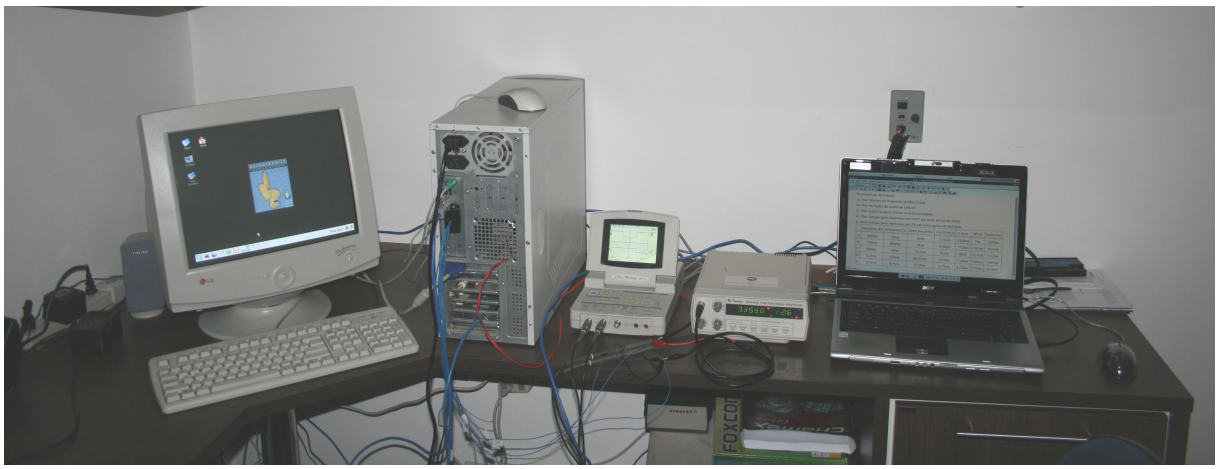


Figura 18: Bancada de testes onde os experimentos foram realizados. Da esquerda para a direita: computador em testes, osciloscópio, gerador de sinais e estação de trabalho

2. Configurar o gerador de sinais para gerar uma onda quadrada com amplitude de 3V e conectar sua saída simultaneamente ao canal B do osciloscópio e à entrada de interrupções da porta paralela (pino 10 do conector DB-25);

3. Conectar uma saída digital da porta paralela do PC (pinos de 2 a 9) ao canal A do osciloscópio;

4. Obter 30 dados de latência para diversas freqüências. Repetir o procedimento com o sistema operacional em condições de sobrecarga.

Com esta configuração, utilizando-se o osciloscópio, é possível verificar qual foi o intervalo de tempo entre o momento em que um pulso do gerador de sinais foi enviado para o PC e o momento em que o PC respondeu invertendo o estado de sua saída digital, como pode ser visto no diagrama da figura 19, que demonstra como observar as latências na tela de um osciloscópio. A figura 20 mostra uma foto dos dois sinais na tela do osciloscópio. Tendo em vista que a interrupção na porta paralela é gerada em uma transição de nível baixo para alto [Rubini e Corbet, 2001], pode-se observar na figura também que é apenas nestes momentos que pode-se medir a latência. Uma foto da bancada de testes descrita pode ser vista na figura 18.

Adicionalmente, foi utilizado o *software LG-View* que recebe os dados do osciloscópio em um computador através de uma porta serial RS-232, permitindo um tratamento posterior e mais preciso destes dados. O recurso de cursores foi utilizado, permitindo medir com precisão a latência das interrupções. O uso do cursor para realizar esta medida através do *LG-View* pode ser visto na figura 21. O resultado da latência pode ser visto na medida ΔT. No caso da figura, a latência é de $3\mu s$.

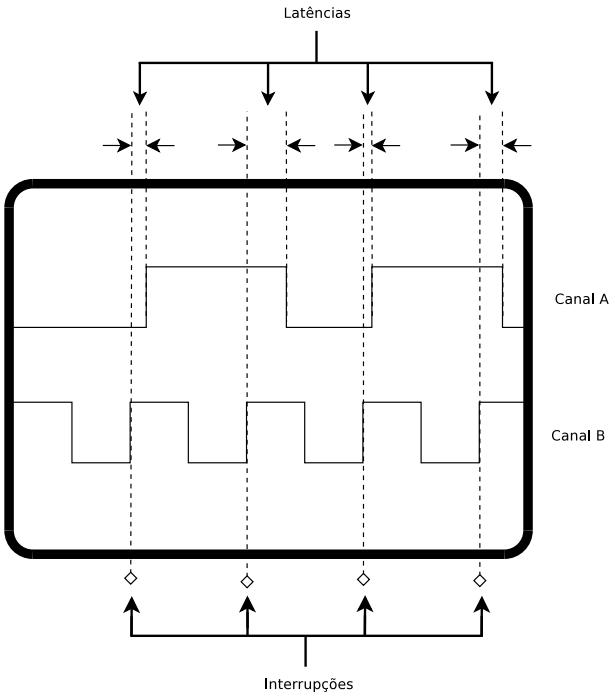


Figura 19: Latências para responder aos estímulos externos observadas na tela de um osciloscópio

Esta analise irá fornecer a latência de resposta das interrupções. É interessante notar que este termo pode ser confuso, sendo que alguns autores consideram latência de interrupção como o tempo entre a ocorrência da interrupção e o início do ISR, enquanto outros tratam como latência o tempo até o fim do ISR. Neste trabalho, como o sistema é analisado como uma caixa preta, a latência medida consiste do tempo total entre a ocorrência da interrupção e seu tratamento.

Com base na latência, pode-se obter a variação da latência, ou *jitter*, que é o quanto o valor da latência varia. Em um sistema de tempo real é esperando que o *jitter* seja sempre nulo (zero). Ou seja, a latência é sempre exatamente a mesma, demonstrando determinismo e previsibilidade no sistema. O *jitter* foi medido calculando a diferença entre cada par de medidas de latência, e obtendo-se destes valores o máximo entre eles.

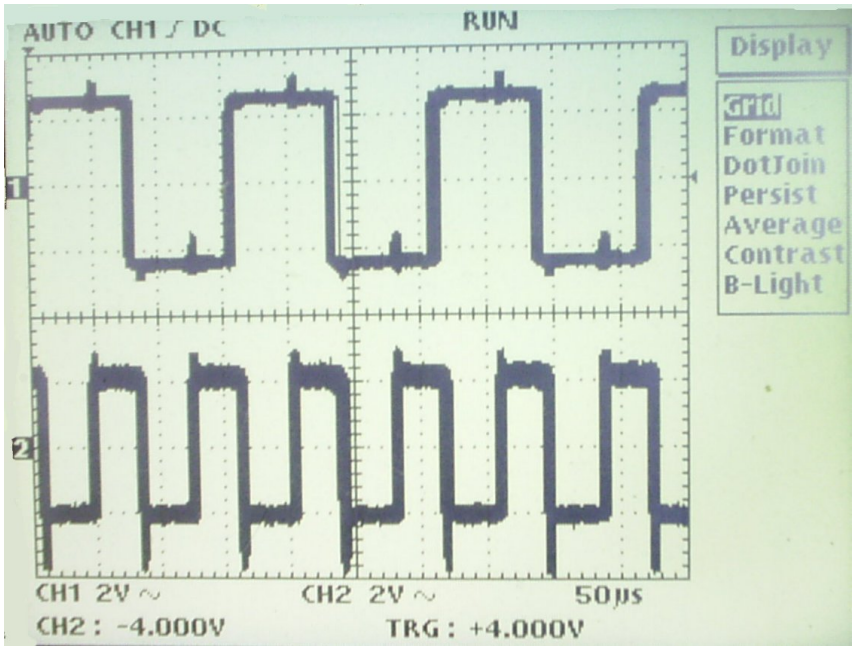


Figura 20: Sinais de entrada [1] e saída [2] do sistema analisado visualizados na tela do osciloscópio

Para auxiliar na medida do *jitter* e dos valores da latência, foi utilizada a função *persist* do osciloscópio, cujo efeito pode ser observado na figura 22. Com este recurso, o osciloscópio sobrepõe em sua tela todas as medidas realizadas enquanto a opção estiver ligada, de forma que uma área "borrada" forma-se representando as latências que ocorreram no sistema. Posteriormente, basta posicionar o cursor do osciloscópio no final da área "borrada" para obter-se a maior latência. Para cada teste realizado, o modo *persist* foi ligado durante 5 minutos. Após este intervalo de tempo, o menor valor, o maior valor e mais 28 valores aleatórios foram selecionados utilizando o cursor do osciloscópio para formar a amostra analisada.

Nesta mesma configuração de teste, também pode-se analisar qual a máxima freqüência de entrada que o sistema consegue responder de forma confiável, como proposto pela ISA. Como não dispunha-se de contadores confiáveis de alta freqüência, uma abordagem diferente foi adotada, mas que levará a medida das mesmas quantidades. O teste consiste em implementar uma ISR em conjunto com um *timer* de um segundo. Este novo ISR será responsável simplesmente por incrementar um contador de *software* que vai contar os pulsos. A cada um segundo, o *timer* vai copiar o valor deste contador, e exibir este valor. Como a freqüência consiste no número de pulsos que ocorrem por segundo, o valor exibido será a freqüência medida pelo sistema operacional através de interrupções externas.

Incrementado-se o valor da freqüência de saída da onda quadrada do gerador de sinais, acompanha-se a medida de freqüência exibida no osciloscópio e no PC. Enquanto estes valores forem iguais o sistema estará respondendo de forma confiável às interrupções. Quando os valores começarem a divergir, o sistema não estará mais respondendo de forma confiável. Neste momento, obtém-se o limiar da freqüência máxima

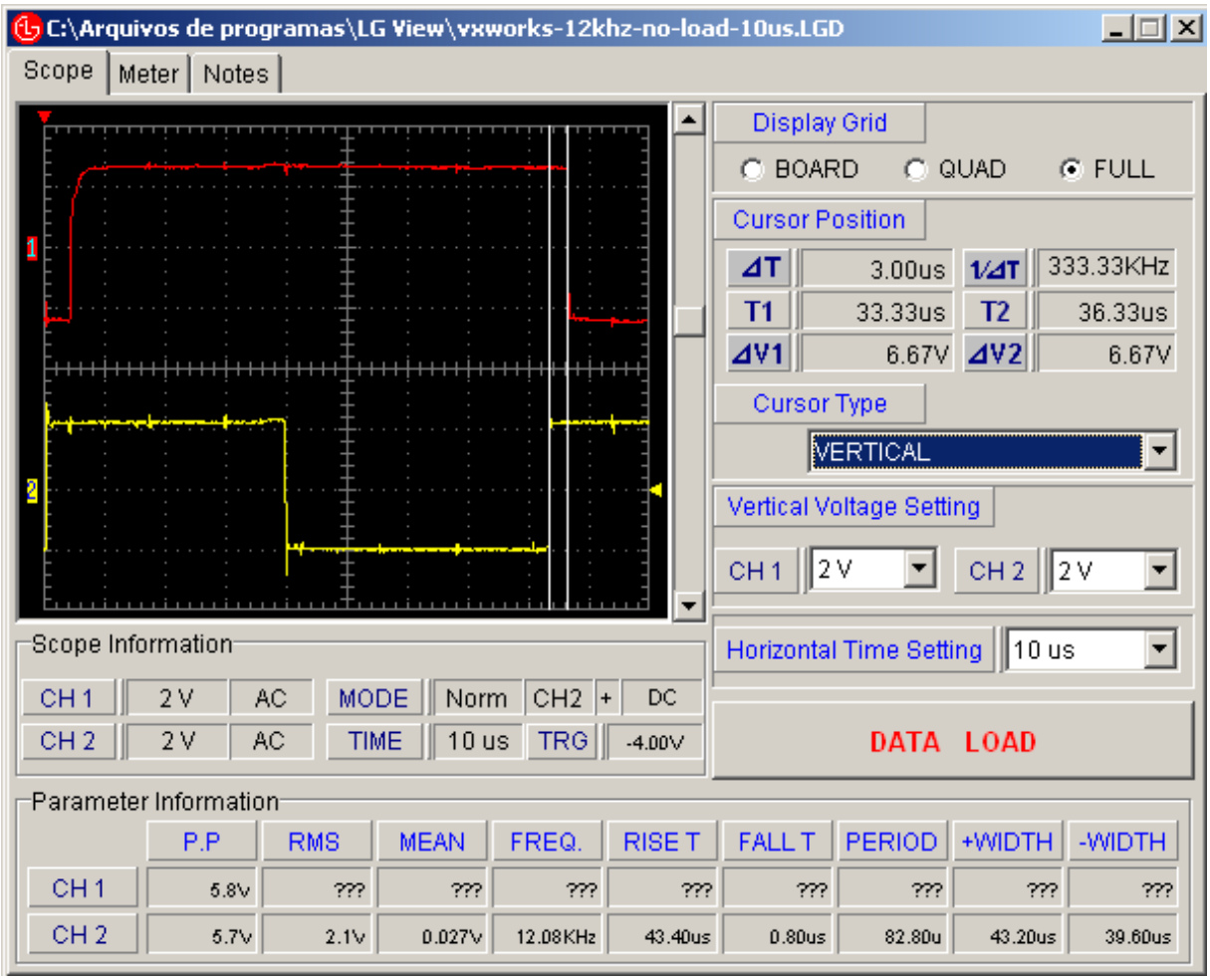


Figura 21: Utilização dos cursores no *software LG-View* para obter a latência

suportada pelo sistema operacional com confiabilidade. Este teste também deverá ser realizado em condições de sobrecarga e com o sistema com o menor número de tarefas possível.

Uma última questão sobre o procedimento dos testes é como sobrecarregar o sistema. Köker [Köker, 2007] sugere sobrecarregar o sistema apenas executando um *ping flood* contra a máquina em testes. Já Rosenquist utiliza operações de entrada e saída em disco, copiando grandes quantias de dados, além do *ping flood* [Rosenquist, 2003]. O *ping flood* consiste em enviar milhares de pacotes por segundo via rede para o endereço IP da máquina em testes, como pode ser visto pelo comando executado na listagem 4.

Este teste será realizado, entretanto ele pode influenciar negativamente nos experimentos dependendo da configuração do *hardware*. Caso a placa de rede esteja na IRQ 5, por exemplo (o que é relativamente comum), as interrupções de *flood* serão tratadas antes das interrupções de teste da latência que vão ocorrer na IRQ 7 da porta paralela, dando resultados bastante negativos. Neste caso o sistema operacional não poderia fazer nada para priorizar a IRQ 7, já que ela é priorizada via *hardware*.

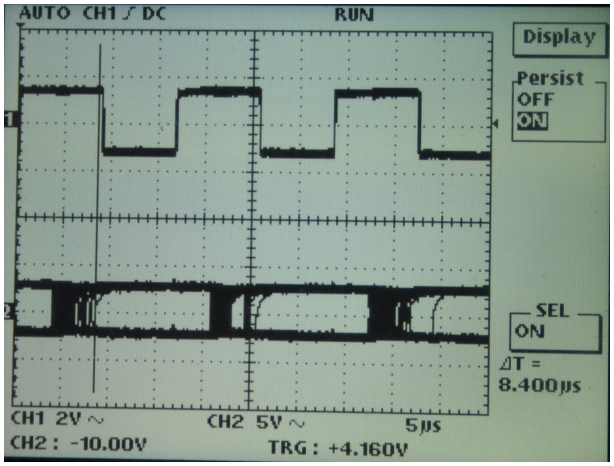


Figura 22: Utilização da função *persist* do osciloscópio para medir latências máximas

Listagem 4 Resultado de um *ping flood* executado contra um computador em testes

```
bash-3.1# ping -f 192.168.254.40
PING 192.168.254.40 (192.168.254.40) 56(84) bytes of data.
..
— 192.168.254.40 ping statistics —
4551 packets transmitted, 4549 received, 0% packet loss, time 6900ms
rtt min/avg/max/mdev = 1.157/1.462/4.018/0.294 ms, ipg/ewma 1.516/1.487 ms
```

7.3 Análise genérica

Antes de testar os sistemas operacionais, foram realizados testes sem o uso de um sistema operacional, para analisar e conhecer o comportamento do *hardware* em testes independente do sistema operacional. Os quatro computadores descritos na seção 7.2.1 foram analisados utilizando a arquitetura de testes já descrita e demonstrada na figura 17. Mais especificamente, foram realizados os testes que medem a máxima freqüência de entrada que o sistema é capaz de atender.

Para realizar o teste, um programa foi implementado em linguagem C e compilado utilizando o compilador *Borland C* para DOS. Este programa implementa um laço principal *main()* e uma ISR que incrementa um contador a cada interrupção recebida. O laço principal consiste de um *loop* infinito que a cada um segundo mostra o valor do contador e em seguida zera seu valor, de forma a mostrar a medida da freqüência de entrada em Hertz.

Embora o DOS seja um sistema operacional, ele não pode ser considerado um *kernel*. Sua única função é carregar um único programa na memória para execução, e atualizar o relógio do sistema. O programa compilado foi executado em cada uma das máquinas, a partir de um disquete que continha o *boot* do DOS e o programa de testes, e os dados foram coletados para um arquivo para posterior análise.

A medida foi realizada para diversas freqüências de entrada. A figura 23 mostra a curva de resposta de um Pentium MMX 150MHz, demonstrando a freqüência de entrada ajustada no gerador de sinais (Freqüência de Entrada) versus a freqüência obtida na leitura via (Freqüência medida) *software*.

Como pode-se ver pela figura, o comportamento do PC foi bastante estável e linear até cerca de 400KHz

de entrada. Para valores superiores a esta freqüência, o programa simplesmente parou de responder, mostrando sempre a mesma leitura independente da freqüência de entrada. Embora o computador não tenha travado, pois o teclado continuou funcionando (interrupções de teclado), o que pode ter ocorrido é a alta freqüência de entrada ter causado a execução constante do ISR impedindo o processador de executar o *loop* da função *main* que atualizava o valor da medida. Cada ponto do gráfico representa a média de 30 leituras para uma determinada freqüência de entrada, além de ser exibido também os valores máximo e mínimo obtidos nas medidas. Utilizando a regra de inverter a maior freqüência suportada, (1/400KHz), pode-se obter um pior caso de resposta de 2,5μs.

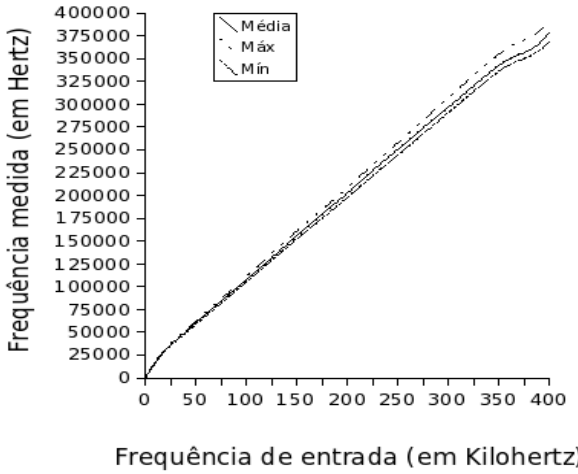


Figura 23: Freqüências inseridas em um Pentium 150MHz através de interrupções externas, e seus valores medidos por um osciloscópio e pela rotina de tratamento de interrupções

A figura 24 mostra as freqüências obtidas ao realizar o mesmo teste, nas mesmas condições, entretanto em um computador com processador Athlon de 1,2GHz. Com um maior poder de processamento, pode-se observar que o laço de repetição principal foi capaz de atualizar constantemente e exibir a medida realizada, entretanto, o *hardware* não foi capaz de receber mais que cerca de duzentas mil interrupções por segundo (200KHz) que é onde a curva tornou-se praticamente constante no eixo da freqüência medida. Para 200KHz, o pior caso de resposta é de 5μs.

Ao realizar o teste com um processador Celeron de 700MHz, o sistema também demonstrou capacidade de responder às interrupções externas corretamente até cerca de 200KHz, como no processador testado anteriormente. O resultado pode ser visto na figura 25.

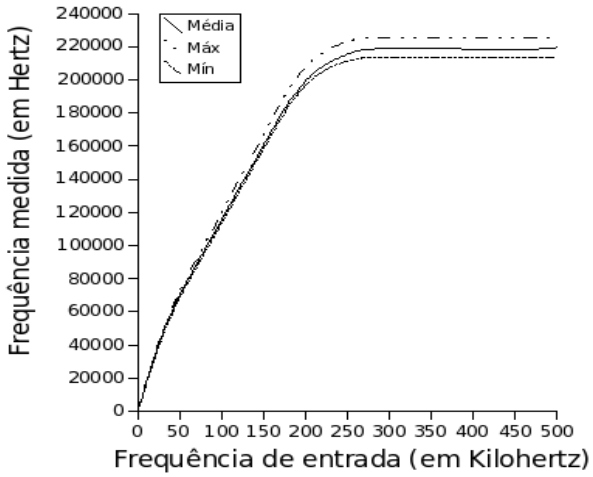


Figura 24: Freqüências inseridas na entrada de um Athlon versus a freqüência medida

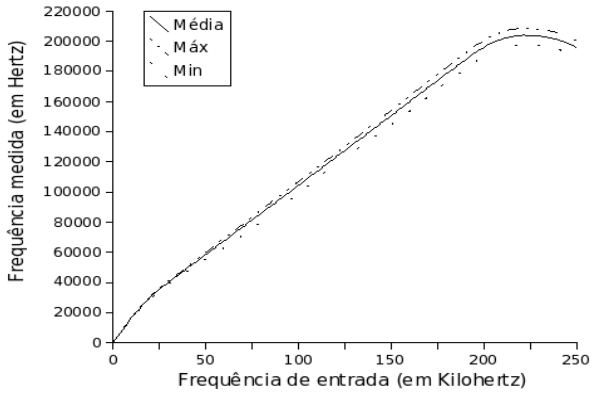


Figura 25: Freqüências inseridas na entrada de um Celeron 700MHz versus a freqüência medida

O teste realizado no processador Pentium II de 400MHz demonstrou muitas variações, e muitas regiões onde medidas incorretas ocorreram, como pode ser visto na figura 26. O sistema tratou as interrupções de forma estável até cerca de 100KHz, levando a um pior caso de resposta de $10\mu s$.

Uma análise mais profunda foi realizada, e foi constatado que a imprecisão das medidas ocorreu principalmente devido a grandes desvios na rotina de temporização disponível na biblioteca dos.h do compilador Borland C. Na verdade, o programa para realizar este experimento simplesmente consiste de uma ISR que incrementa em um (1) o valor de uma variável toda vez que uma interrupção ocorre. Em conjunto com a ISR, um *loop* infinito imprime o valor desta variável, aguarda um segundo através da função *sleep* e reinicia o ciclo. Um teste adicional foi realizado trocando a função *sleep()* que recebe como parâmetro o número de segundos a esperar, pela função *delay()* que recebe como parâmetro o número de milisegundos a esperar, contudo os resultados foram ainda piores que os obtidos na figura 26, onde foram medidas frequência com 1/3 do valor da freqüência de entrada. Um fato curioso é o fato de que o mesmo *software* sendo executado em outros computadores não teve a mesma característica irregular apresentada neste gráfico. Como a análise desta questão específica não é o foco deste trabalho, pretende-se realizar

um estudo mais aprofundado desta característica em um trabalho futuro.

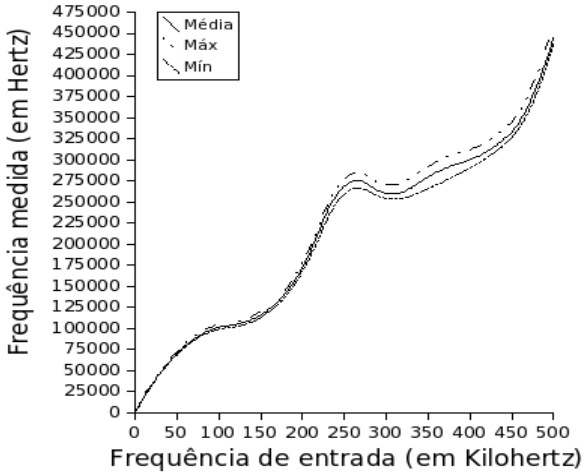


Figura 26: Freqüências inseridas na entrada de um Pentium-II 400MHz versus a freqüência medida

A partir desta análise também foi escolhido qual computador seria utilizado para realizar o teste dos sistemas operacionais. Na verdade, esta escolha levou vários meses devido a diversas dificuldades encontradas para conseguir que todos sistemas operacionais fossem executados no mesmo *hardware*.

Como os RTOS são usados nos mais diversos tipos de processadores e *hardware*s, com placas com diferentes recursos e características de entrada e saída, os fabricantes fornecem pacotes de *software* chamados de BSP (*Board Support Package)*. Os pacotes de suporte à placa permitem executar um sistema operacional em uma placa específica com determinado processador, aproveitando os recursos desta placa. A execução dos sistemas operacionais QNX, Windows XP, Linux, μC/OS-II foi possível em todos computadores deste estudo com razoável facilidade. Entretanto, o VxWorks e o Windows CE, devido a diversos motivos que serão discutidos posteriormente, só foram executados com sucesso no Pentium II. Dessa forma, o Pentium II foi escolhido como plataforma de *hardware* para realização dos testes.

Dada a escolha deste computador específico, alguns experimentos adicionais sem o uso de um *kernel* foram realizados neste PC.

De acordo com a metodologia discutida, alguns testes vão consistir de alterar o estado de uma saída digital da porta paralela do PC. Como foi discutido previamente, operações de entrada e saída consomem bastante tempo de processamento, de forma que outro ponto importante desta análise é medir o tempo que estas operações levam para ser executadas. Para obter esta informação foram realizadas 2000 operações de saída com a instrução *outb()*, alternando seus valores de saída. As medidas foram feitas utilizando-se uma sonda de *software* com o TSC como medida de tempo. Os dados obtidos estão sumarizados na tabela 4.

Os resultados apresentados na tabela 4 são coerentes com dados de outros pesquisadores, que obti-

Informação	Duração do comando
Máximo	2.12μs
Mínimo	2.09μs
Média	2.1μs
Desvio Padrão	130ns

Tabela 4: Informações sobre o tempo de execução da instrução outb()

veram valores que variam entre $1\mu s$ [Haridasan, 2003, RadiSys, 1998] e $3{,}2\mu s$ [Yodaiken e Barabanov, 1997]. Com uma velocidade de 400MHz, o processador em testes é capaz de executar, em geral, uma instrução a cada 2,5ns, no caso de instruções que levam um ciclo de *clock*, já que algumas podem levar vários ciclos para serem executadas. Isto significa, que o tempo que o processador leva para executar a função *outb()*, permitiria a ele executar cerca de mil outras instruções. Isto também causa impacto na latência de interrupções do sistema, já que as interrupções só podem ocorrer depois do término da execução de uma instrução.

Outra conseqüência das afirmações acima, é que existe um limite superior de freqüência ao gerar uma onda quadrada na porta paralela de um PC que é muito inferior à velocidade do processador. Para gerar uma onda na saída da porta paralela são necessárias duas instruções *outb* consecutivas para gerar um nível alto e em seguida um nível baixo. Dessa forma, se o computador executasse um laço infinito sem nenhuma espera (*delay* ou *sleep)* seria obtida uma onda com período de cerca de $4{,}2\mu s$ que consiste de uma freqüência máxima de 238KHz. Este dado teórico também é coerente com o gráfico da figura 27, onde um *software* foi utilizado para gerar diversas freqüências de saída até saturar o *hardware* em seu limite superior.

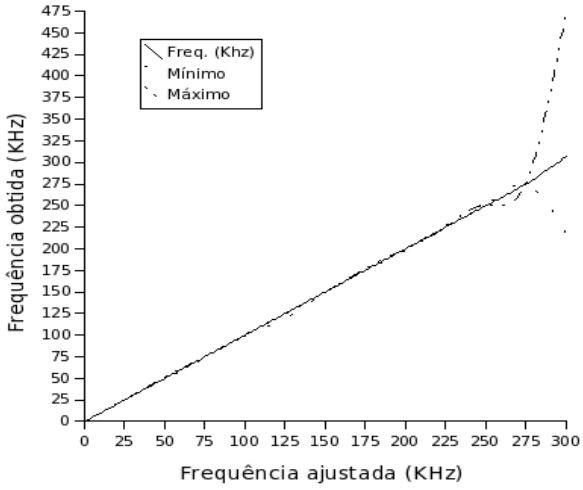


Figura 27: Medidas realizadas em um osciloscópio a partir de freqüências geradas em um Pentium II com a instrução *outb()*

Finalmente, foram obtidos os dados de latência para o Pentium II sem o uso de sistema operacional. A

figura 28 mostra o tempo que a rotina de tratamento de interrupção levou para responder às interrupções em diversas freqüências. Deve-se saber que o tempo que a instrução *outb()* leva não foi considerado neste gráfico. O teste foi realizado conforme o diagrama da figura 17. O eixo Y do gráfico indica a latência em microsegundos.

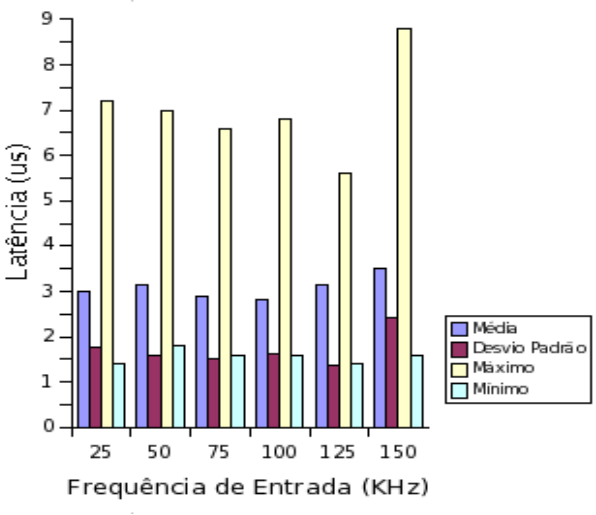


Figura 28: Tempo de resposta da rotina de tratamento de interrupções em várias freqüências

7.4 Windows XP

Embora o Windows XP não seja um sistema operacional de tempo real, é comum encontrar diversas situações onde utiliza-se o Windows XP para controlar alguma aplicação crítica, incluindo equipamentos médicos e sistemas para controle de motores em locomotivas de trens [Stiennon, 2008]. Por este motivo o Windows XP foi incluído neste estudo.

O Windows XP possui um *clock* com resolução de 10ms a 15ms [Sato e Asari, 2006] e um gerenciador de tarefas que permite ao usuário determinar a prioridade de um processo em execução, como pode ser visto na figura 29. Nesta figura nota-se a existência de uma prioridade com título "Tempo Real" e é importante considerar que esta opção não oferece capacidade de tempo real a uma tarefa. De fato, a escolha deste nível somente determina que o processo escolhido possuirá a maior prioridade do sistema.

Entretanto, a Microsoft vem realizando diversos esforços para inserir características de tempo real em seus sistemas operacionais. O Windows NT chegou a ser bastante usado na indústria para controle de processos de tempo real, devido a sua confiabilidade, de forma que o sucessor do Windows NT, o Windows 2000, possui um escalonador de tarefas com duas classes. Uma variável, e uma de tempo real: as tarefas da classe variável somente são escalonadas quando as tarefas de tempo real terminaram de realizar suas atividades [de Oliveira et al., 2001].

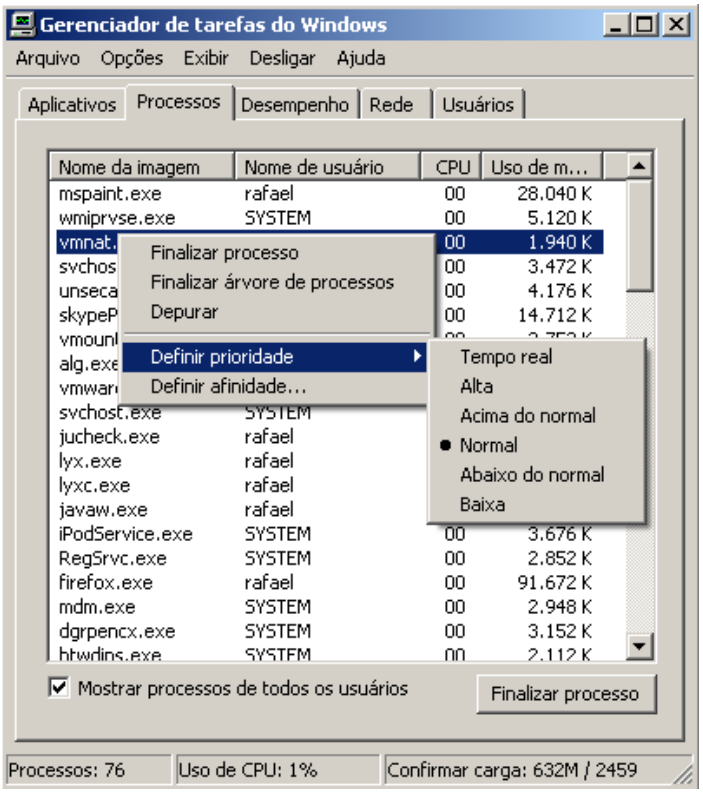


Figura 29: Gerenciador de Tarefas do Windows XP

Como a implementação de programas que controlam portas de entrada e saída no Windows XP é relativamente trabalhosa, foi utilizado um utilitário chamado *RapidDriver*[13] que automatiza o desenvolvimento de *device drivers* para o Windows. O *RapidDriver* instala um *driver* de baixo nível junto ao *kernel* do Windows, permitindo ao usuário desenvolver sua aplicação em espaço de usuário. O programa de testes foi desenvolvido em linguagem Delphi 6.0, e foi capaz de tratar as interrupções com auxilio do *RapidDriver*. Outra precaução tomada, foi instalar uma versão do Windows a partir de um disco formatado, para evitar interferências de *softwares* e *drivers* que poderiam estar em uma instalação anterior do sistema. Por ser um sistema operacional de uso genérico, ele necessita ser instalado em um disco rígido, e a plataforma computacional consiste de PCs.

O primeiro experimento realizado para determinar o pior caso de tempo de resposta foi realizado em 4 diferentes situações: 1 - programa em teste com prioridade normal e o sistema sem sobrecarga, 2 - programa em teste com prioridade normal e o sistema sobrecarregado, 3 - programa em teste com prioridade tempo real e o sistema sem sobrecarga, 4 - programa em teste com prioridade tempo real e o sistema com sobrecarga. Para sobrecarregar o Windows foram criados 250 processos levando a um consumo de cerca de 700MB de memória, sendo grande parte desta memória *swap* em disco.

Durante estes testes, e em medidas posteriores foi identificado que freqüências de entrada superiores

[13] Maiores informações em: http://www.entechtaiwan.com/dev/rapid/index.shtm

a 25KHz tornam o sistema instável. Isto levaria a um pior caso de resposta de $40\mu s$. Contudo, ao sobrecarregar o sistema, o Windows já deixou de responder corretamente às interrupções com freqüências acima de 5KHz levando a um pior caso de resposta de $200\mu s$. A situação foi ainda pior ao executar o *ping flood* em conjunto, fazendo com que o sistema todo trave e exiba uma tela azul de erro com a seguinte mensagem: "Foi detectado um problema, e o Windows foi desligado para evitar danos ao computador". É importante ressaltar que a ocorrência da tela azul foi grande e o procedimento para tal erro ocorrer é bem determinado, bastando inserir interrupções a 25KHz na porta paralela enquanto executa-se um *ping flood* contra o Windows XP.

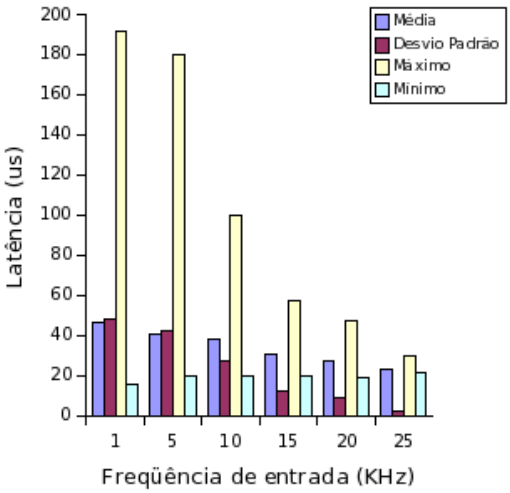
A figura 30 mostra os resultados da análise de latências para o tratamento das interrupções desde o momento da ocorrência destas até um sinal de resposta ser gerado em uma saída digital da porta paralela do PC. A análise também foi realizada para os quatro cenários já descritos. Em (a) e (b) as interrupções foram tratadas em até aproximadamente $200\mu s$, entretanto em (c) e (d) o tempo para tratar as interrupções chegou a quase 1ms. Para freqüências acima da 5KHz o Windows também tornou-se instável e travou apresentando tela azul.

Uma observação importante com relação ao gráfico da figura 30, é que quanto maior a freqüência, menor o intervalo de latência que pode ser medido, já que uma freqüência mais alta possui um menor período, afetando as medidas, como se elas fossem cada vez menores. Isto pode ser verificado pela tendência de queda dos gráficos a medida que a freqüência aumenta, entretanto isto não representa um problema nas medidas, já que estamos buscando o pior caso que pode ser observado em freqüências menores. Este comportamento também ocorrerá nos gráficos de latência dos outros sistemas operacionais.

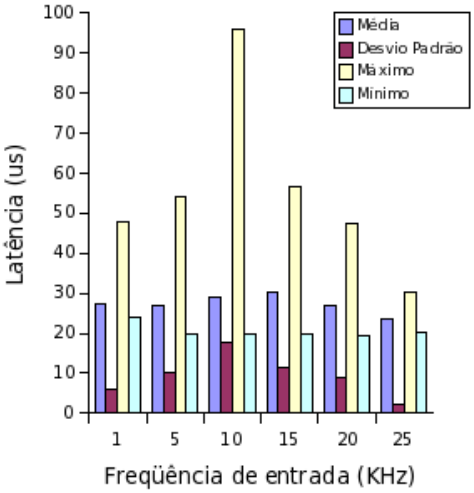
A variação das latências *(jitter)* mostradas na figura 30 também foi observada para cada um dos cenários, sendo $168\mu s$ no caso (a), $32\mu s$ no caso (b), $700\mu s$ no caso (c) e $348\mu s$ no caso (a).

Um teste adicional foi realizado no Windows para conhecer o seu *overhead* em operações simples. Como foi apresentado na tabela 4, o PC em testes leva cerca de $2\mu s$ para realizar uma operação de escrita em porta de saída digital. O experimento coletou sessenta amostras de tempo da execução das instruções *OUT* para escrever em uma porta de saída e *IN* para ler de uma porta de entrada. Os resultados destas medidas nos quatro cenários de carga e prioridades podem ser vistos na tabela 5.

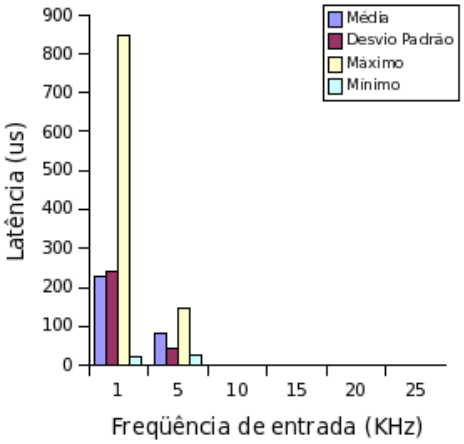
Nos experimentos realizados o Windows XP foi estressado ao máximo com o uso de *ping floods* que geram milhares de interrupções por segundo via placa de rede somado a uma freqüência fixa de interrupções sendo gerada através das interrupções da porta paralela. O Windows XP demonstrou bastante estabilidade e boa resposta de tempo real até um certo limite. O uso da prioridade tempo real também demonstrou-se eficiente, pois enquanto a tarefa de tempo real estava consumindo tempo de CPU, todas outras, inclusive o ponteiro do *mouse* e o teclado, paravam de responder.



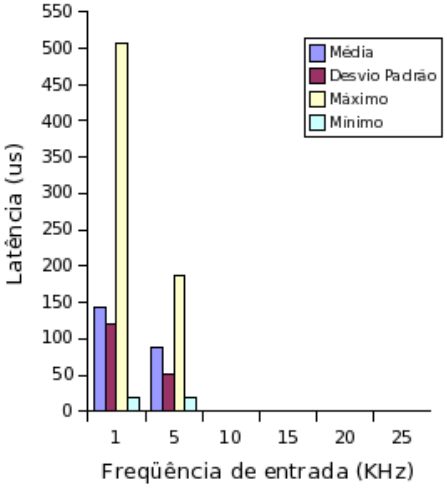
(a) Prioridade normal, sem sobrecarga



(b) Prioridade Tempo Real, sem sobrecarga



(c) Prioridade normal, com sobrecarga



(d) Prioridade Tempo Real, com sobrecarga

Figura 30: Latências para tratar interrupções no Windows XP

Operação/Condição de Execução	Dado	IN (μs)	OUT (μs)
Prioridade normal	Máximo	78,3	123,93
	Mínimo	25,44	26,52
	Média	30,27	30,75
	Desvio Padrão	3,83	5,07
Prioridade normal com o sistema sobrecarregado	Máximo	152117,82	131972,63
	Mínimo	25,97	28,81
	Média	171,61	257,96
	Desvio Padrão	3414,23	3576
Prioridade tempo real	Máximo	69,05	2115,95
	Mínimo	26,2	28,6
	Média	30,58	41,74
	Desvio Padrão	3,15	69,93
Prioridade tempo real com o sistema sobrecarregado	Máximo	2184,76	1290,14
	Mínimo	26,29	29,66
	Média	72,75	79,43
	Desvio Padrão	77,67	59,33

Tabela 5: Tempo para execução de funções de acesso a portas de entrada e saída no Windows XP

A conclusão é que verificando suas restrições e aplicações, o Windows pode ser utilizado com certo determinismo em um sistema de tempo real. Cinkelj ainda afirma que é possível realizar a aquisição de dados com garantias de tempo real *soft* no Windows XP quando o computador não está sobrecarregado [Cinkelj et al., 2005]. Entretanto, ele não poderia ser utilizado em sistemas de tempo real *hard* seguindo os limites propostos pela OMAC.

7.5 Windows CE

Embora a Microsoft não tenha um histórico de desenvolvimento de sistemas operacionais de tempo real, ela vem investindo de forma tímida, mas crescente em tecnologias de tempo real e robótica. Por exemplo, dos 76.000 empregados da Microsoft, apenas 11 estão trabalhando no Microsoft *Robotics Studio*, que atualmente é gratuito e promete mudar o paradigma de como programa-se robôs [Cherry, 2007]. Além disto, o segmento de *software* embarcado na Microsoft gerou "apenas" 154 milhões de dólares no segundo semestre de 2004, o que representa apenas uma pequena porcentagem da receita total da Microsoft neste mesmo semestre, que foi de 11 bilhões de dólares [Taurion, 2005].

A Microsoft possui diversos sistemas operacionais para sistemas embarcados. O Windows XP *Embedded*, e o Windows CE *Embedded*. Mais recentemente também surgiu o Windows *Mobile*, que é uma versão de Windows CE otimizada para telefones celulares. Destes sistemas, apenas o Windows CE foi desenvolvido para atender demandas de *hard real-time*, sendo que a própria Microsoft recomenda utilizar utilitários externos para dar capacidade de tempo real às outras versões de Windows, como o RTX e o INTime [Microsoft, 2007b]. Além disso, a Microsoft ainda dispõe do Windows CE automotivo, especialmente customizado para aplicações da indústria automobilística.

Como em outros sistemas operacionais, a interrupção de *clock* do Windows CE é a "batida de coração" do sistema operacional [Viswanathan, 2006], entretanto uma inovação bastante interessante está presente no Windows CE com relação ao *clock*: é o *clock tick* variável. Como sabe-se, o *clock tick* pode ser uma fonte de *overhead* nos sistemas operacionais. Com o *clock tick* variável do Windows CE, o *kernel* verifica que em determinado momento não é preciso gerar *clock ticks* a cada 1ms, mas somente em 100ms, alterando a freqüência *clock tick*. Dessa forma, o *kernel* ajusta o clock conforme as necessidades atuais [Viswanathan, 2006] do sistema. Isto também implica em economia de energia e melhoria na capacidade de processamento.

Outra característica muito interessante do Windows CE *Embedded* 6.0, é que a Microsoft disponibilizou o código fonte deste sistema operacional. As partes disponíveis dependem de cada pacote adquirido, entretanto, existem pacotes que dão acesso a 100% do código fonte aos desenvolvedores [Windows For Devices, 2006]. Além disto, a Microsoft disponibiliza todo pacote de desenvolvimento do Windows CE sem custo nenhum por quatro meses para os desenvolvedores conhecerem e testarem o sistema antes de realizar a compra.

O Windows CE possui um *kernel* preemptivo com 256 níveis de prioridades [Cedeno e Laplante, 2007]. Estas prioridades são organizadas em várias classes, que determinam se a aplicação em execução é ou não de tempo real.

A versão de Windows CE analisada é a mais recente disponível: o Windows CE 6.0 *Embedded*. Para desenvolver um sistema com o Windows CE, utiliza-se a ferramenta de programação da Microsoft Visual Studio com a adição da ferramenta *Platform Builder*. Esta ferramenta permite configurar e personalizar cada item que o sistema operacional vai possuir, e gerar uma imagem completa do sistema operacional, incluindo o *kernel, device drivers*, aplicativos, interface gráfica e até navegador *web*. Como o *Platform Builder* compila o sistema operacional por completo, este processo é bastante lento. A compilação de uma imagem do sistema em um computador *Core 2 Duo* de 1,6GHz e 1GB de memória RAM levou cerca de 3 horas.

Antes de compilar o sistema deve-se escolher a plataforma de destino do sistema (*target)* para que o binário gerado pela compilação seja compatível com o processador desejado. É possível escolher as arquiteturas ARM, MIPS, SH4 ou x86 (PCs), sendo a última utilizada.

O pacote de desenvolvimento do Windows CE, disponibiliza muitas ferramentas práticas e úteis para verificar se o sistema está atendendo aos requisitos de tempo real. Algumas ferramentas interessantes para análise de sistemas de tempo real são:

- O *Kernel Tracker* permite visualizar graficamente a linha do tempo de execução do sistema, com eventos e interrupções mostrados no momento em que ocorrem;
- O *ILTiming* que mede latências para o tratamento de interrupções em diversas situações.
- O *OSBench* permite medir o tempo que o Windows CE leva para criar semáforos, entrar e sair de regiões críticas, criar *threads*/tarefas, e dezenas de outros parâmetros importantes.

Os testes da *Dedicated Systems* indicaram que o Windows CE 3.0 tem um comportamento estável e robusto de tempo real, entretanto sua configuração é bastante complicada devido ao fato do sistema ser altamente configurável [Dedicated Systems, 2002g]. Ao avaliar o Windows CE .NET 4.0, as mesmas opiniões foram mantidas, incluindo o fato da documentação ser vasta, embora confusa e não estruturada, dificultando o aprendizado para iniciantes na plataforma [Dedicated Systems, 2002h]. De fato, uma das dificuldades encontradas ao implementar os testes foi encontrar a informação necessária na vasta documentação, já que as pesquisas na documentação da Microsoft nem sempre sanavam as dúvidas.

Em outra análise, testes realizados no Windows CE usando as técnicas semelhantes às utilizadas nesta pesquisa, observou-se bastante estabilidade e determinismo do Windows CE para tarefas de 1ms, com jitter não superior a $10\mu s$ e latência de tratamento de interrupção não superior a $13\mu s$ [Tacke e Ricci, 2002].

Para iniciar o Windows CE, é preciso gravar uma imagem do sistema operacional em uma memória *flash*, ou fazer o *boot* através da rede. O Visual Studio já inclui todas ferramentas para executar o processo via rede, bastando iniciar o computador com um disquete especial, que tem a função de descarregar a imagem do sistema via rede, e iniciar sua execução.

Um dos problemas encontrados durante o desenvolvimento foi incompatibilidade com *hardware.* O Windows CE não iniciou por completo em um PC com processador Athlon, e o disco para inicialização do sistema via rede suporta poucos modelos de placas de rede. Para obter sucesso nesta etapa foi necessário utilizar uma placa já obsoleta e difícil de encontrar, compatível com NE2000. Entretanto, uma vez iniciado o sistema, até mesmo a depuração e monitoramento com as ferramentas apresentadas podem ser feitas via rede. Além disto, o Windows CE utiliza a porta serial RS-232 do PC para enviar mensagens de depuração para o computador de desenvolvimento (*host).* As mensagens são tão detalhadas que apenas o processo de inicialização do sistema gera 150Kbytes de texto.

Para executar a análise proposta, foi necessário implementar um *device driver* junto ao *kernel* do Windows CE, que implementa os procedimentos para realizar os testes. Dentro deste *device driver* foram criadas várias tarefas na forma de *threads*, sendo uma tarefa executada a cada um segundo para medir a freqüência de entrada, e uma tarefa associada às interrupções.

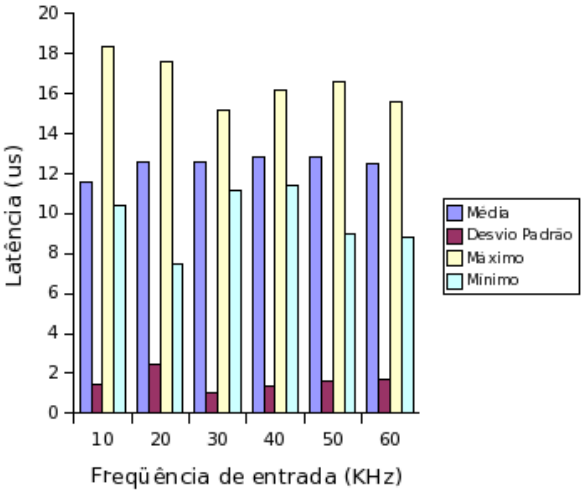
Inicialmente, a tarefa que foi executada a cada um segundo possuía prioridade 251, uma das mais baixas do sistema. Após analisar a documentação e verificar que prioridades acima de 248 devem ser usadas para tarefas que não tenham requisitos de tempo real, a prioridade desta tarefa foi alterada para 91. As prioridades de 0 a 96 são as mais altas do sistema e reservadas para *device drivers* com requisitos de tempo real.

Ao executar o procedimento de testes com a baixa prioridade (251), o Windows chegou a responder freqüências de entrada de até 50KHz, mas com muitos erros de medida. Para uma entrada de 60KHz, a medida obtida variou entre 48KHz e 95KHz, com um valor médio de 64KHz. Foi notável a mudança de qualidade das medidas ao reduzir a prioridade para 91. A tabela 6 sumariza as freqüências geradas no gerador de sinais e obtidas na medida do Windows CE com um sinal de entrada de 50KHz.

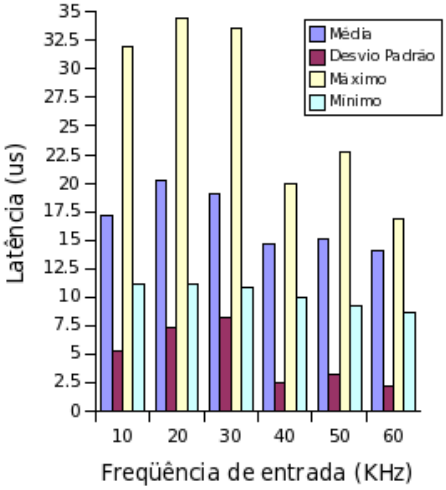
Estado do sistema	Máximo	Mínimo	Média	Desvio Padrão
Operação Normal	52155Hz	50368Hz	51120Hz	417Hz
Sistema Sobrecarregado	51618Hz	50434Hz	50952Hz	354Hz
Ping Flood	51353Hz	47250Hz	49709Hz	1171Hz

Tabela 6: Freqüências medidas pelo Windows CE a partir de um sinal de entrada de 50KHz

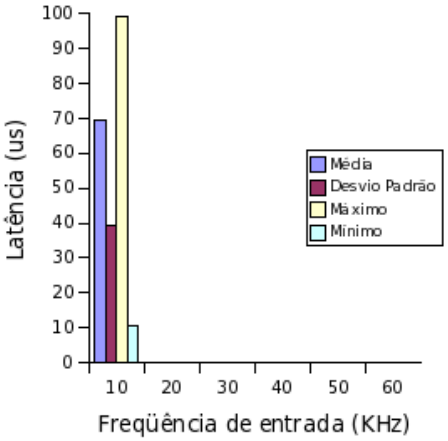
Como pode ser visto na tabela 6, o Windows mostrou bastante estabilidade na medida das freqüências de entrada, mesmo nas piores condições. Além disto, a freqüência do gerador de sinais foi ajustada várias vezes até seu valor máximo de 1MHz, sem causar nenhum dano ao sistema em execução. Entretanto, a



(a) Operação normal



(b) Sistema sobrecarregado



(c) Com *ping flood*

Figura 31: Latências para tratar interrupções no Windows CE

freqüência máxima de entrada para qual o sistema respondeu de forma adequada foi de 50KHz, e com base neste valor pode-se definir o pior caso de tempo de resposta do Windows CE como de 20μs (1/50KHz).

Os resultados do teste externo de latências podem ser vistos na figura 31. Tanto na situação (a) quanto (b) da figura, o sistema mostrou-se bastante estável e previsível. No cenário (c), só foi possível medir as latências para freqüências de entrada de 10KHz, já que a sobrecarga do sistema com o *ping flood* foi tão grande, que provavelmente o sistema não conseguiu responder às interrupções.

Para os cenários da figura 31, também foram obtidos valores máximos de variação da latência (*jitter)*, sendo o valor em (a) de 5,8μs , o valor em (b) de 22,2μs e em (c) de 88,8μs.

Por fim também foi analisado o tempo que o sistema leva para executar instruções de entrada e de saída. Estes valores podem ser vistos na tabela 7.

Operação/Condição de Execução	Dado	*IN* (μs)	*OUT* (μs)
Sistema em operação normal	Máximo	309,24	2,49
	Mínimo	1,54	1,56
	Média	5,35	1,61
	Desvio Padrão	33.62	0,17
Sistema sobrecarregado	Máximo	350,77	2,37
	Mínimo	1,55	1,56
	Média	6,23	1,61
	Desvio Padrão	38,86	0,12
Sistema sobrecarregado com *ping flood*	Máximo	524,85	8139,67
	Mínimo	1,56	1,54
	Média	6,63	45,61
	Desvio Padrão	50,71	489,8

Tabela 7: Tempo para execução de funções de acesso a portas de entrada e saída no Windows CE *Embedded* 6.0

Uma consideração final sobre o Windows CE é relacionada à sobrecarga do sistema: nas tentativas de sobrecarregar o sistema, ao abrir dezenas de programas simultaneamente, houve várias ocasiões em que o Windows exibiu uma mensagem avisando que o sistema não tinha mais recursos para executar o novo programa, e permitia escolher algum programa em execução para finalizar e liberar recursos para executar o programa desejado. Esta característica é muito interessante, pois consiste de um mecanismo do *kernel* proteger o sistema de estados inesperados quando o uso de recursos ultrapassa um limite aceitável.

Além disto, não houve nenhuma situação em que o sistema tenha travado, levando à necessidade de um *reboot*. No pior caso de sobrecarga, o sistema chegou a ficar sem dar resposta por algum tempo, mas em seguida voltou a responder.

A conclusão é que o Windows CE *Embedded* 6.0 é um sistema operacional bastante robusto e confiável para execução de tarefas de tempo real, inclusive em conformidade com os requisitos da OMAC, além dele oferecer ferramentas e opções de desenvolvimento práticas e eficientes.

7.6 QNX Neutrino

O QNX Neutrino é um dos sistemas operacionais de tempo real mais tradicionais do mercado. Baseado na arquitetura de *microkernel*, ele também é completamente compatível com os padrões POSIX e certificado pelas normas DO-278 e MIL-STD-1553. Nesta arquitetura, o *kernel* implementa apenas quatro serviços essenciais: escalonamento de tarefas, comunicação entre tarefas, comunicação de rede de baixo nível e detecção de interrupções. Todo restante do sistema é implementado como tarefas de usuário, tornando o *kernel* rápido, confiável e pequeno (com apenas cerca de 7KBytes de código) [Barabanov, 1997].

Uma das vantagens do *microkernel* sobre outros tipos de *kernel* é que mesmo que um erro grave ocorra,

como por exemplo no sistema de gerenciamento de arquivos, todo resto do sistema não é influenciado pelo problema. Dessa forma, a arquitetura de *microkernel* oferece um ambiente mais robusto que a usada em outros SOs, entretanto seu maior problema é o *overhead* causado pela proteção de memória [Timmerman, 2001] que deve ser utilizada com grande freqüência, pois todas partes do sistema são fortemente isoladas. De acordo com Chatterjee, um dos motivos pelo qual a performance de sistemas de *microkernel* é inferior a de SOs de *kernel* monolítico, é que o *hardware* é feito pensando em sistemas com *kernel* monolítico [Chatterjee e Strosnider, 1996].

Em um estudo realizado em 2008, Amianti usou os critérios e relatórios da *Dedicated Systems*, para analisar sistemas operacionais de tempo real com relação a instalação, arquitetura, bibliotecas, rede, ferramentas, documentação e performance [Amianti, 2008]. Neste estudo, concluiu-se que o QNX Neutrino seria o melhor sistema operacional de tempo real para ser usado em um veículo aéreo não tripulado (VANT).

O modelo comercial do QNX é considerado híbrido [Rosen, 2007], pois a empresa fornece o código fonte do seu sistema operacional (salvo pequenas partes que dependem de terceiros), bem como distribui o *software* gratuitamente para pesquisa e desenvolvimento. Todo pacote de desenvolvimento é disponível gratuitamente sem limitações de funcionalidade ou prazo, de forma que a compra de licenças de uso só precisa ser feita quando um produto comercial baseado neste sistema for lançado comercialmente.

De acordo com os testes da *Dedicated Systems,* realizados em um Pentium 200MHz, o QNX 4.25 tem um pior caso de latência de 4,25μs [Martin Timmerman, 2000d], e leva um tempo máximo de 4μs para tratar uma interrupção de *clock* [Dedicated Systems, 2001e]. O mesmo teste foi realizado no QNX 6.1, e resultados de latência muito próximos foram obtidos. Além disto, constatou-se que o QNX pode responder a interrupções que ocorrem com intervalo de 10μs, sendo esta a sua máxima freqüência de atendimento a interrupções [Dedicated Systems, 2002d]. O tempo de 4μs para tratamento de interrupções de *clock tick* também foi mantido de uma versão para outra.

A grande diferença do QNX 4 para o QNX 6, é que na versão 4 apenas a arquitetura PC era suportada, e na versão 6, o sistema passou a suportar as arquiteturas StrongARM, MIPS, PowerPC e SH4, além do PC. Nesta versão, o RTOS da QNX passou a ser chamado de Neutrino 6.2. Ele manteve as mesmas características do 6.1, com uma pequena redução na velocidade de tratamento de interrupções, e melhoras em outras aspectos, como na documentação [Dedicated Systems, 2002c]. Um outro recurso da nova versão chama-se particionamento adaptativo (*adaptative partitioning)* que permite criar restrições do uso do processador por tarefas, como por exemplo: a tarefa A não pode usar mais que 30% da CPU, e a tarefa B não pode usar mais que 10% da CPU.

Ao comparar o QNX 6.1, VxWorks AE 1.1 e Windows CE .NET (sucessor do Windows CE 3.0) a *Dedicated Systems* concluiu que todos os três sistemas operacionais são adequados com relação aos seus comportamentos de tempo real, entretanto o QNX 6.1 é bastante superior aos outros dois concorrentes

[Dedicated Systems, 2002a]. Em um relatório comparativo seguinte, o Linux da Red Hat foi incluído na comparação, não sendo indicado para sistemas de tempo real [Dedicated Systems, 2002b].

O sistema analisado nesta pesquisa foi o QNX Neutrino RTOS versão 6.3.2. Seu processo de instalação e desenvolvimento é extremamente simples e rápido, bastando descarregar uma imagem de um CD do *site* da QNX, e gravar esta imagem em CD. Após gravar o CD, basta dar o *boot* em um computador através deste CD, e escolher se a opção de instalar o sistema em um disco rígido ou de rodar a partir do CD, sem a necessidade de discos adicionais. Esta abordagem de *Live CD* é muito prática, pois permite rodar o QNX em qualquer computador sem ser necessário instalar o sistema, mesmo que o computador já tenha outro sistema operacional instalado.

Todos os testes foram realizados diretamente a partir do CD, que funcionou em diversos modelos de computadores, incluindo Athlon, Pentium e Core 2 Duo. Ao iniciar o sistema, o QNX também detectou todas placas de rede instaladas no sistema, e iniciou sua interface gráfica (o *Photon)* onde até mesmo um navegador de internet está disponível, que na primeira tentativa de uso conseguiu navegar na *internet.*

O desenvolvimento de aplicações pode ser feito em linguagem C ou C++, e para tal utiliza-se um ambiente integrado de desenvolvimento *Integrated Development Environment (IDE)* chamado QNX Momentics IDE que pode ser utilizado no Windows, QNX ou Linux. Esta ferramenta é baseada em um IDE gratuito chamado *Eclipse.* Através deste IDE é possível criar, compilar e depurar programas via rede através de um modelo de desenvolvimento *host/target.* Neste modelo, um computador utilizando o *Eclipse* atua como *host,* sendo utilizado para compilar e verificar saída dos programas. Um outro computador, *o target* recebe via rede o programa compilado no *host* e o executa. Esta abordagem é interessante em sistemas de tempo real, pois muitos computadores utilizados para controlar este tipo de sistema, não possuem nenhum tipo de teclado ou *display.*

A execução do procedimento foi bastante simples, bastando iniciar o computador *target* com o *Live CD* do QNX e iniciar o programa *qconn*, que permite realizar o desenvolvimento remoto. Após criar um projeto no *Eclipse,* basta mandar executar este programa em um *target*, especificando seu endereço de rede, e o programa é executado diretamente no *target,* enquanto suas saídas são exibidas na tela do *Eclipse.*

Para sobrecarregar o sistema, os programas disponíveis em sua interface gráfica foram abertos várias vezes até totalizar 146 processos em execução simultânea, e 100% de uso da CPU. Durante a execução dos testes, o programa para analisar a performance do sistema teve sua prioridade configurada para a máxima disponível no sistema.

O primeiro teste realizado observou as freqüências máximas de resposta do sistema, que foram estáveis até 50KHz, com alguns desvios durante a execução de um *ping flood.* Os resultados deste experimento podem ser vistos na tabela 8. Como o sistema foi estável até cerca de 50KHz, seu pior caso de resposta é de $20\mu s$, de forma análoga ao Windows CE.

Estado do sistema	Máximo	Mínimo	Média	Desvio Padrão
Operação Normal	48236Hz	47720Hz	48126Hz	101Hz
Sistema Sobrecarregado	53781Hz	35538Hz	48183Hz	4815Hz
Ping Flood	60242Hz	49430Hz	51506Hz	2167Hz

Tabela 8: Freqüências medidas pelo QNX Neutrino a partir de um sinal de entrada de 50KHz

Operação/Condição de Execução	Dado	*IN* (μs)	*OUT* (μs)
Sistema em operação normal	Máximo	3,22	3,63
	Mínimo	1,41	1,43
	Média	1,55	1,58
	Desvio Padrão	0,32	0,35
Sistema sobrecarregado	Máximo	3,19	62,95
	Mínimo	1,42	1,43
	Média	1,59	2,69
	Desvio Padrão	0,36	7,92
Sistema sobrecarregado com *ping flood*	Máximo	3,77	3,03
	Mínimo	1,44	1,44
	Média	1,58	1,58
	Desvio Padrão	0,39	0,16

Tabela 9: Tempo para execução de funções de acesso a portas de entrada e saída no QNX Neutrino 6.3.2

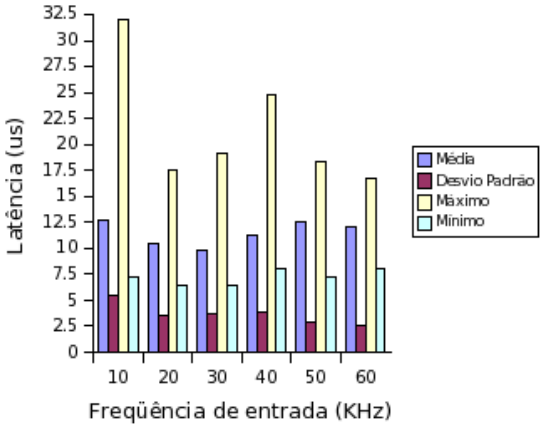
O teste das latências para tratamento de interrupção também foi executado e seus resultados podem ser vistos na figura 32. A partir das medidas de latência, também obteve-se os valores máximos de *jitter* que na situação (a) da figura foi de $32\mu s$, na situação (b) foi de $19{,}6\mu s$ e na situação (c) foi de $23{,}99\mu s$.

Com relação aos testes para executar operações de entrada e saída no *hardware*, os resultados podem ser vistos na tabela 9.

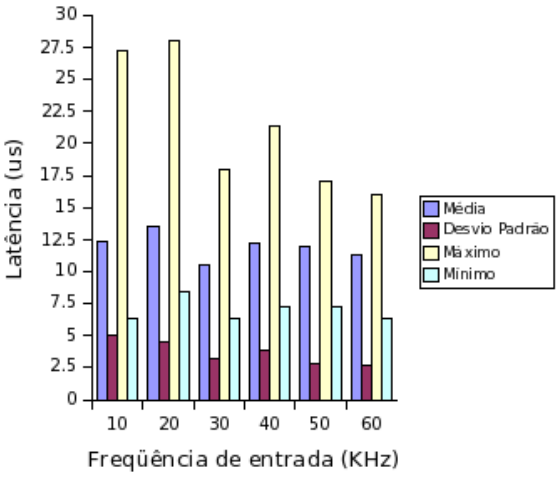
Os testes realizados demonstraram previsibilidade e robustez do sistema. Todos testes foram realizados em seqüência, sendo que nenhuma situação de sobrecarga levou a uma necessidade de reiniciar o sistema. As freqüências de teste produzidas pelo gerador de sinal chegaram a 1MHz, e em nenhuma situação o sistema demonstrou desvios em seu determinismo. Além disto, valores obtidos foram compatíveis com àqueles propostos pela OMAC.

7.7 μC/OS-II

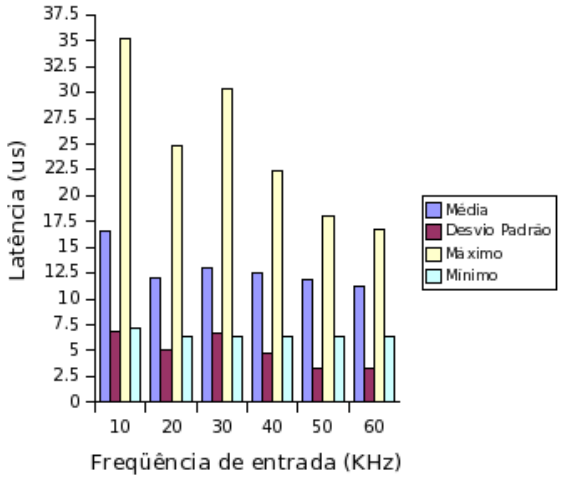
O μC/OS-II, abreviação de *Microcontroller Operating System Version II,* consiste de um *kernel* de tempo real bastante simples desenvolvido por Jean Labrosse, um projetista de sistemas embarcados com mais de 20 anos de experiência nesta área. Além disto, Labrosse é uma das autoridades mais conhecidas na área de RTOS [Ganssle, 2004]. De acordo com o livro em que Labrosse descreve em detalhes o funcionamento de seu *kernel*, argumenta-se que a decisão de implementar seu próprio sistema operacional



(a) Operação normal



(b) Sistema sobrecarregado



(c) Com *ping flood*

Figura 32: Latências para tratar interrupções no QNX Neutrino RTOS

foi realizada pois outras soluções comerciais não estavam atendendo com qualidade aos seus requisitos [Labrosse, 2002].

Portado para mais de cem arquiteturas [Ganssle, 2004], o μC/OS-II é principalmente utilizado em microcontroladores com poucos recursos. Este sistema também foi certificado pelo departamento de transportes americano, mais especificamente a divisão de aviação comercial - *Federal Aviation Administration* (FAA). Esta certificação corresponde a homologação para sua utilização em sistemas de missão crítica, como controle de vôo, de acordo com a norma RTCA DO-178B [Ganssle, 2004, Labrosse, 2002].

O sistema operacional é distribuído na forma de código fonte, sendo necessário compilar o μC/OS-II juntamente com a aplicação, gerando um único arquivo binário executável. Para a arquitetura PC, o sistema pode ser compilado no Microsoft Visual Studio ou Borland C, sendo a segunda opção utilizada para seguir exemplos do livro de Labrosse, que utiliza este compilador. O livro de Labrosse, adicionalmente, é uma ótima referência para aprendizado de conceitos de sistemas operacionais e tempo real, já que ele possui vários exemplos práticos que podem ser testados em qualquer computador com um compilador C e Windows, além de explicar os conceitos de forma simples e objetiva.

No PC, o μC/OS-II basicamente substitui o vetor de interrupções do DOS, de forma que que é possível executá-lo em uma janela de *prompt* de comando do Windows XP ou Windows 2000, já que o Windows emula o MS-DOS nestas janelas. Entretanto, para atender às garantias temporais, seria necessário iniciar o computador somente com o MS-DOS e posteriormente, rodar o programa compilado que carrega a aplicação juntamente com o sistema operacional.

Em sua configuração padrão, o μC/OS-II ajusta o *clock tick* do sistema para uma freqüência de 200Hz que aciona as rotinas internas do *kernel* preemptivo do μC/OS-II, que encarrega-se de sempre executar a tarefa de maior prioridade que esteja pronta para execução. O *kernel* suporta 64 tarefas, sendo 2 reservadas para o sistema. A tarefa de menor prioridade no sistema (prioridade 63) é a tarefa ociosa (*idle task)* que possui contadores para coletar estatísticas do sistema, que em conjunto com outras tarefas fornece a carga do processador. Dessa forma, cada tarefa do sistema possui uma prioridade fixa.

Adicionalmente, o *kernel* do μC/OS-II também fornece o protocolo de herança de prioridades para evitar que ocorra inversão de prioridade, e mecanismo de controle de reentrância.

Para executar a análise do sistema, um disco de *boot* do MS-DOS foi criado, e nele copiado os programas de teste compilados previamente. Após iniciar o computador com este disco, foi necessário apenas executar cada um dos programas.

Ao executar o teste de máxima freqüência suportada pelo sistema, diversas dificuldades surgiram. Uma rotina de tratamento de interrupção foi escrita de acordo com a documentação e exemplos do sistema operacional. Como este sistema fornece poucos serviços, a implementação da ISR e sua configuração foram feitas através de funções da biblioteca do Borland C para DOS. A documentação não impõe restrições com relação a isto, solicitando apenas que dentro desta rotina, seja feita uma chamada para avisar o

kernel que uma interrupção começou a ser tratada e que seu tratamento terminou. A ISR pode ser vista na listagem 5.

Listagem 5 Rotina de tratamento de interrupções para medir a freqüência de interrupções na porta paralela com o μC/OS-II

```
//Rotina de tratamento de interrupção da IRQ 7 - INT 0x0F
void interrupt far InterruptHandler(void) {
  OSIntEnter(); //Avisa o kernel que uma interrupção está sendo tratada
  disable(); //Desabilita interrupções até tratar a atual
 isrCounter++; //Incrementa o contador de interrupções que ocorreram
  enable(); //Volta a habilitar as interrupções
  outportb(0x20, 0x20); //Envia o comando End of Interrupt (EOI) para o controlador de interrupções
  OSIntExit(); //Avisa o sistema operacional que a interrupção já foi tratada
}
```

Ao executar os testes o sistema demonstrou-se muito instável, travando freqüentemente. Foi observado que quanto maior a freqüência de entrada, mais rápido o sistema travava, exibindo mensagens de erro incompletas e diversos caracteres espalhados pela tela. Algumas mensagens de erro que puderam ser lidas diziam: "Erro de estouro de divisão por zero. Se o problema persistir, entre em contato com o fornecedor do programa" e *"Floating point error. Abnormal program termination"*. Uma pesquisa sobre estes erros na *internet* indicou que eles são freqüentes ao executar programas antigos, como jogos, em computadores fabricados recentemente, e sugerem algumas correções que não solucionaram o problema. Outro motivo deste erro ter ocorrido pode ter relação com o fato deste *kernel* não oferecer isolamento e proteção de memória entre as tarefas.

Ao realizar mais testes, foi notado que a remoção da chamada às funções *OSIntEnter()* e *OSIntExit()* eliminou o problema. Ao analisar o código fonte destas rotinas, a única operação que elas realizam é incrementar ou decrementar uma variável chamada *OSIntNesting* para que o escalonador tome decisões de escalonamento. Certamente este problema está relacionado a algum estouro de variável ou vazamento de memória. Dessa forma, os testes foram realizados sem notificar o *kernel* através destas chamadas.

A freqüência de entrada foi incrementada aos poucos até atingir 520KHz, tanto de entrada, quanto medida pela tarefa criada para fazer a medida via *software*. Até este momento, a resposta foi linear sendo que nesta medida observou-se que o uso da CPU estava em 99%. Ou seja, todo processamento estava sendo gasto tratando interrupções e na tarefa de maior prioridade no sistema, responsável por exibir a freqüência medida na tela. Esta tarefa foi criada com prioridade 5. Ao aumentar mais a freqüência de entrada, e observar o uso da CPU atingir 100%, o sistema travou. Este procedimento foi repetido inúmeras vezes, notando a reincidência da falha.

Os resultados da máxima freqüência aplicada ao sistema podem ser vistos na tabela 10. Não foi possível executar o teste de *ping flood* já que o sistema operacional não oferece suporte a rede (isto seria possível comprando um pacote adicional do fornecedor do sistema). O teste de sobrecarga foi executado criando 9 tarefas e pressionando várias vezes o teclas do teclado do computador, para gerar interrupções

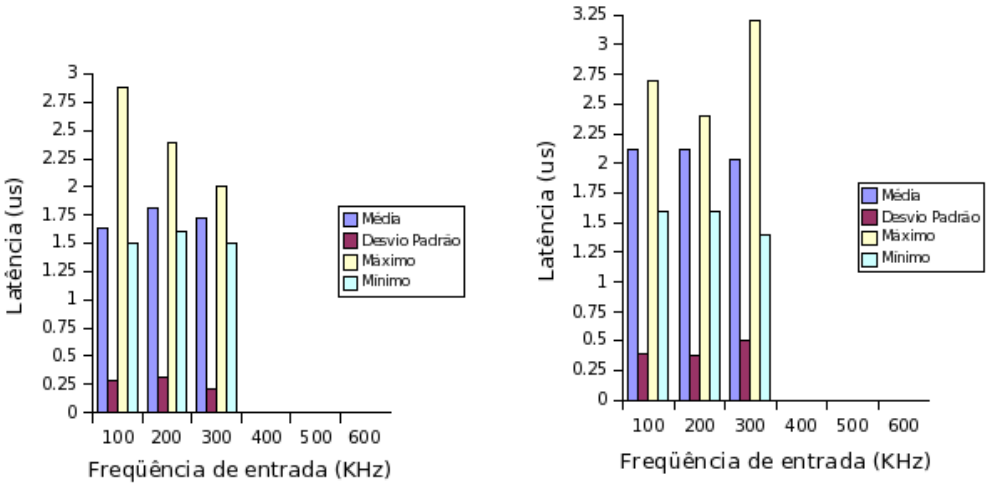
concorrentes às que estavam sendo medidas.

Um resultado muito interessante obtido nesta medida, é que foi possível medir com precisão freqüências de até 520KHz, o que não ocorreu ao testar o *hardware* sem um *kernel* (capítulo 7.3), como pode ser observado na figura 26. Certamente a maior qualidade nas medidas foi obtida graças a mecanismos de temporização mais precisos oferecidos pelo *kernel* que aqueles oferecidos pelo DOS e pela biblioteca do Borland C. Neste caso, a rotina utilizada para causar uma espera de um segundo foi a *OSTimeDlyHMSM(0, 0, 1, 0)*. Com base nestes dados, um pior caso de resposta de 1,92μs foi obtido.

Estado do sistema	Máximo	Mínimo	Média	Desvio Padrão
Operação Normal	521079Hz	520705Hz	520938Hz	85Hz
Sistema Sobrecarregado	520484Hz	419991Hz	501842Hz	24170Hz

Tabela 10: Freqüências medidas pelo μC/OS-II a partir de um sinal de entrada de 520KHz

A análise das latências para tratamento de interrupção podem ser vistas na figura 33, sendo que para valores acima de 400KHz, o sistema parou de responder e não voltou mais a responder mesmo desativando a freqüência de entrada. Na situação (a) da figura a maior latência obtida foi de 2,88μs e o maior *jitter* foi de 1,6μs, enquanto no cenário (b) a pior latência foi de 3,2μs e o maior *jitter* medido foi de 2,32μs.



(a) Operação normal (b) Sistema sobrecarregado

Figura 33: Latências para tratar interrupções no μC/OS-II

O tempo para executar as instruções de entrada e saída não foram analisados, já que o μC/OS-II não fornece rotinas para esta funcionalidade, sendo que foi necessário utilizar as funções *inportb()* e *outportb()* da biblioteca de entrada e saída do Borland C, que foram as mesmas utilizadas no teste apresentado na tabela 4.

Durante os testes pode-se observar que o μC/OS-II possui um excelente escalonador, já que os testes para medir a freqüência de entrada demonstraram ótimos resultados quando a tarefa de medida possuía a maior prioridade do sistema (5), enquanto seus resultados tornaram-se péssimos quando foi dada a menor prioridade disponível no sistema para esta tarefa (62).

O sistema também demonstrou pequena variação nas medidas ao comparar-se uma situação de sobrecarga e uma situação normal, com tempos bastante baixos e compatíveis com os requisitos da OMAC. Duas ressalvas com relação ao μC/OS-II, é que este sistema oferece poucos serviços para o programador, sendo necessário apoiar-se em outras bibliotecas e funções externas, e a implementação da rotina de tratamento de interrupções apresentou alguns problemas ao notificar o *kernel* da ocorrência de uma interrupção.

7.8 Linux

O Linux é um sistema operacional totalmente gratuito que foi desenvolvido nos moldes do consolidado Unix. Criado em 1991 por Linus Torvalds em um esforço de criar uma versão de Unix para o processador 80386 da Intel [Grier, 2007b], o Linux evoluiu muito ao longo destes anos e tornou-se popular no meio acadêmico, corporativo e de usuários finais. O Linux é um software gratuito sob os termos da licença GPL [Foundation, 2007] da *Free Software Foundation*. Esta licença garante que o Linux seja distribuído livremente, e suas alterações continuem a ser livres.

O Linux possui um modelo de desenvolvimento em que milhares de desenvolvedores ao redor do mundo colaboram escrevendo partes do sistema operacional, entretanto a decisão de quais contribuições são incluídas na versão oficial do *kernel* são centralizadas por Linus Torvalds. Este modelo de desenvolvimento causa receio e desconfiança em muitos desenvolvedores, por acreditarem que um sistema deste tipo não pode ser confiável [Taurion, 2005].

Para se ter uma idéia da qualidade e confiabilidade de *softwares* livre gratuitos, um estudo realizado em 2002 identificou 115 *softwares* livre gratuitos utilizados com sucesso no departamento de defesa americano [Bollinger, 2003]. O mesmo estudo ainda concluiu e recomendou a continuação do uso desta modalidade de *software*. Os *softwares* livres ainda são ótima alternativas para uso em setores públicos, onde busca-se baixo custo e padrões abertos [Statskontoret, 2004]. Adicionalmente, vários autores acreditam que com novas ferramentas surgindo cada vez mais para o Linux, ele tende a ser um sistema dominante no mercado de embarcados [Ip, 2001, Taurion, 2005].

O Linux possui um *kernel* monolítico, onde todos seus componentes importantes, como gerenciador de memória, escalonador de processos, sistema de arquivos, *device drivers*, fazem parte de um único grande programa. Além disso, é possível adicionar ou remover dinamicamente funções ao *kernel* através de módulos de kernel chamados de *Linux Kernel Modules (LKM)*. O *kernel* implementa proteção de memória com auxilio da MMU, e as tarefas são separadas em espaço de *kernel* e espaço de usuário.

Até a versão 2.4 do seu *kernel*, o Linux era um sistema operacional com grandes latências, pois muitas partes de seu *kernel* não eram preemptíveis. Em alguns casos, a latência dentro do *kernel* 2.4 do linux pode ser de dezenas ou até centenas de milisegundos [Heursch et al., 2001, Dedicated Systems, 2002e]. A partir do *kernel* 2.6, o próprio *kernel* passou a poder ser "preemptado", reduzindo latências. Este recurso é muito bom para sistemas embarcados e de tempo real [Santhanam, 2003]. Contudo, sabe-se que partes desta última versão do *kernel* também não são preemptíveis devido ao uso do *Big Kernel Lock* (BKL), um recurso que foi mantido de versões anteriores para manter compatibilidade com código legado [Franke, 2007].

Com relação a sistemas de tempo real, o Linux não é um sistema operacional de tempo real, entretanto, existe um *patch* de baixa latência chamado *low-preempt patch* que ao ser aplicado no Linux convencional, oferece a ele características de tempo real *soft*. Um *patch* é uma alteração que é aplicada a um programa para reparar um problema ou adicionar uma funcionalidade. O próprio Linus Torvalds, afirmou em 2006 que usaria o Linux para controlar uma máquina de corte a laser usando este *patch* de preempção com baixa latência.

Entretanto, incluir características de tempo real em um *kernel* com milhões de linhas de código, como o do Linux, é bastante complicado, sendo que normalmente outras abordagens que serão discutidas na próxima seção podem ser utilizadas. Dessa forma, o uso do *patch* de baixa latência pode não ser adequado para transformar o *kernel* do Linux em um *kernel* de tempo real. Além disto, um estudo de Ambike mediu a resolução do relógio de sistemas populares, como o Windows 2000 e Red Hat Linux 7.3, e obteve dados conclusivos para afirmar que estes sistemas não são adequados para aplicações de tempo real [Ambike et al., 2005].

Com relação aos mecanismos temporais do Linux, é possível configurar a freqüência com a que a interrupção de *clock tick* é gerada, através da opção HZ na configuração do *kernel*. Esta escolha tem dois lados: quanto maior o valor de HZ, maior a granularidade e precisão do relógio, entretanto, maior o *overhead* tratando a interrupção de *clock tick*. Até a versão 2.4 do *kernel* do Linux, o valor padrão era de 100Hz, mas em versões recentes do *kernel* 2.6, o valor passou a ser de 1000Hz [Venkateswaran, 2005].

Dentro do *kernel* do Linux, a variável *jiffies* armazena a quantia de vezes que a interrupção de *clock tick* ocorreu desde o *boot* do sistema. Para um *kernel* com HZ=1000, um *jiffie* corresponde a 1ms, e para um *kernel* com HZ=100, um *jiffie* corresponde a 10ms. O *kernel* também possui uma variável chamada CPU_KHz, que é determinada na hora do *boot* da máquina. Durante a inicialização, o *kernel* configura um *timer* de hardware e usa o TSC para contar quantas instruções foram realizadas em um determinado tempo, para calcular a velocidade do processador. Usando o CPU_KHz em conjunto com o TSC, é possível realizar medidas com certa resolução e qualidade [Venkateswaran, 2005].

No *kernel* 2.6 do Linux, o escalonador do *kernel* foi reimplementado para ter um tempo constante de decisão/escolha do próximo processo a ser executado, aumentando seu determinismo [Taurion, 2005].

Como já foi discutido, o *swap* de memória deve ser evitado ao máximo em sistemas de tempo real, pois isto introduz *delays* longos e aleatórios nas tarefas de tempo real. A especificação POSIX para tempo real, prevê as chamadas *mlock()* e *mlockall()* para impedir que uma tarefa seja removida da memória principal, sendo que o Linux implementa estas extensões [Beal, 2005].

A *Dedicated Systems* decidiu testar o Linux de propósito geral em seu programa de avaliações de sistemas operacionais de tempo real, devido a muitas solicitações de usuários, mesmo não se tratando de um RTOS. Os resultados mostraram um tempo máximo para tratamento de interrupções de $13\mu s$ em um Pentium 200. Eles também analisaram a freqüência máxima de resposta de interrupções, onde obteve-se interrupções espaçadas de $60\mu s$ como o limite onde o sistema não perdeu nenhuma interrupção [Dedicated Systems, 2002e]. Entretanto o tempo para tratar interrupções de *clock* no Red Hat Enterprise Linux 7.2 chegou a levar $141\mu s$ no pior caso.

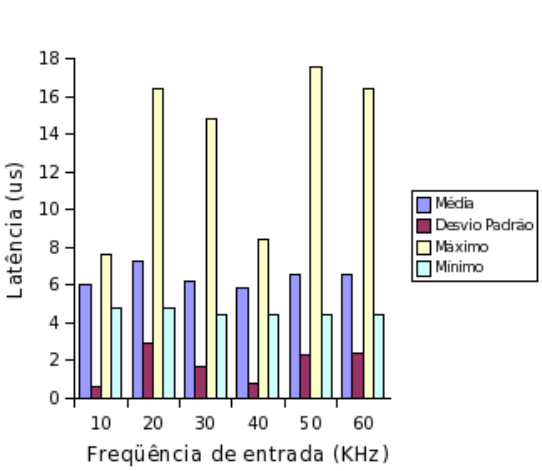
Na análise realizada neste trabalho, o *kernel* versão 2.6.17 foi obtido do *site* oficial (*http://www.kernel.org)*, compilado e analisado. A primeira análise realizada obteve a máxima freqüência de operação em que o *kernel* respondeu sem a perda de pulsos provenientes do gerador de sinais. Por motivos de segurança e proteção do sistema, interrupções de *hardware* só podem ser tratadas pelo *kernel*, de forma que um *device driver* foi implementado na forma de LKM para realizar os testes. O *kernel respondeu* de forma confiável até cerca de 72KHz, o que proporciona um pior caso de resposta de $13{,}89\mu s$. Os resultados do teste podem ser vistos na tabela 11.

Estado do sistema	Máximo	Mínimo	Média	Desvio Padrão
Operação Normal	73950Hz	71589Hz	72243Hz	441Hz
Sistema Sobrecarregado	74558Hz	66958Hz	70209Hz	2000Hz
Ping Flood	60422Hz	50834Hz	52675Hz	1835Hz

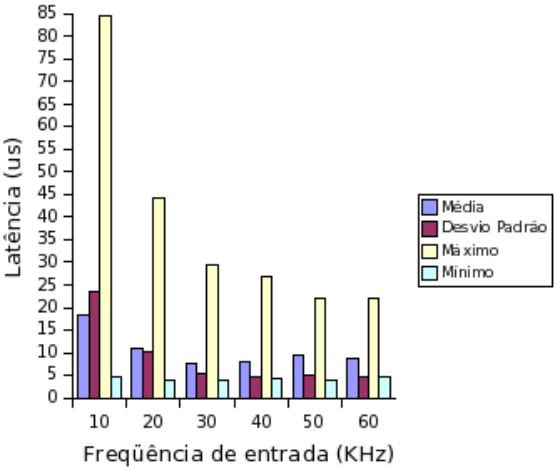
Tabela 11: Freqüências medidas pelo Linux a partir de um sinal de entrada de 72KHz

Para realizar o teste com o sistema sobrecarregado, 300 tarefas foram criadas e a utilização do sistema foi monitorada através do utilitário *top* que exibe quantas tarefas estão em execução, qual a porcentagem de uso da CPU pelo *kernel,* e por tarefas em espaço de usuário. Outro dado adicional exibido pelo *top* é uma informação chamada de *load average* do sistema. Esta medida é normalmente preferível por entusiastas de sistemas baseados em Unix, pois ao invés de mostrar o uso instantâneo da CPU, exibe quantos processos estão na fila esperando para ser executados no último minuto, nos últimos 5 minutos e nos últimos 15 minutos. Em geral um sistema é considerado sobrecarregado com um *load average* acima de 3, sendo que nos testes foi obtido um valor de 22.

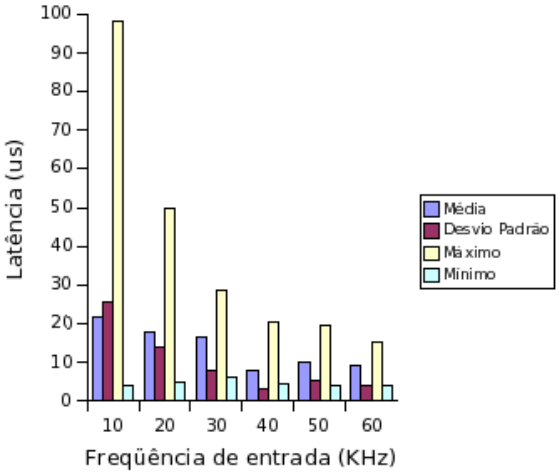
Como era esperado, o *ping flood* teve efeito bastante negativo, reduzindo o valor da medida em até 20KHz. Além disto, ao aumentar a freqüência do gerador de sinais para 1MHz durante a execução do *ping flood*, o Linux travou por completo, não voltando mais a responder.



(a) Operação normal



(b) Sistema sobrecarregado



(c) Com *ping flood*

Figura 34: Latências para tratar interrupções no Linux

As latências obtidas podem ser vistas na figura 34, e em geral apresentaram bons resultados médios, com um valor máximo de $98\mu s$ mesmo para o sistema sobrecarregado. O *jitter* obtido no cenário (a) foi de $11.9\mu s$, no cenário (b) de $77.6\mu s$ e no cenário (c) de $58\mu s$.

Os tempos para execução de funções de entrada e saída também foram bastante constantes e baixos mesmo em situações de sobrecarga, como pode ser visto na tabela 12.

Embora o Linux não seja um RTOS, ele mostrou bom comportamento temporal, porém na situação de uma alta freqüência em sua entrada somada a um *ping flood*, o sistema tornou-se instável e travou. Além disto, o *kernel* demonstrou um *jitter* bastante alto, o que pode causar variações inesperadas em sistemas de tempo real que necessitam de precisão.

Operação/Condição de Execução	Dado	*IN* (μs)	*OUT* (μs)
Sistema em operação normal	Máximo	2,12	2,12
	Mínimo	1,37	1,29
	Média	1,44	1,39
	Desvio Padrão	0,18	0,14
Sistema sobrecarregado	Máximo	1,24	1,93
	Mínimo	1,09	1,1
	Média	1,09	1,14
	Desvio Padrão	0,03	0,11
Sistema sobrecarregado com *ping flood*	Máximo	5,72	5,26
	Mínimo	1,06	1,07
	Média	1,23	1,21
	Desvio Padrão	0,85	0,6

Tabela 12: Tempo para execução de funções de acesso a portas de entrada e saída no Linux

7.9 Real Time Application Interface (RTAI)

Na área de tempo real, o Linux vem demonstrando um bom custo/benefício, sendo utilizado em casos que vão desde pequenas aplicações e testes até equipamentos médicos e científicos no estado-da-arte [Irwin et al., 2002, Katuin, 2003]. De acordo com Weinberg, o Linux de tempo real é capaz de atender plenamente 95% das aplicações de tempo real [Weinberg, 2001].

O *Real Time Linux* tem tido uma aceitação tão boa, que reconhecidas instituições com demanda para utilização de RTOS vêm optando pelo seu uso, como por exemplo, o Instituto Americano de Padrões e Tecnologias (NIST) [NIST, 2002] e a Administração Nacional de Espaço e Aeronáutica NASA [Kalynnda, 2002]. A NASA manteve por vários anos um projeto chamado *FligthLinux*, para utilizar-se Linux de tempo real em satélites e outros equipamentos aero-espaciais [Hardison, 2006]. Embora o projeto não esteja mais ativo na NASA, ele inspirou o projeto aberto *Open Fligth Linux*[14]. Na própria NASA, o Linux de tempo real já foi usado em ambiente de produção para construir um radar que coleta dados de tornados [Yodaiken, 1999].

Outro fator que indica o grande crescimento de versões de Linux para tempo real, é que empresas consolidadas no mercado de sistemas operacionais de tempo real, como *Wind River e Lynx Works* já disponibilizaram produtos e continuam trabalhando com suas versões comerciais de Linux de tempo real.

A Wind River tem casos de sucessos de vários clientes usando suas versões de Linux de tempo real, como a *Palm* (fabricante de PDAs) e a Honeywell, que utilizou o Linux de tempo real da Wind River ao desenvolver um projeto para a NASA [Wind River, 2007].

Por outro lado, existem empresas que são extremamente inflexíveis e resistentes ao uso do Linux. A *Green Hills*, fornecedora de um dos mais reconhecidos sistemas operacionais na área militar, o Integrity, causou bastante polêmica com um artigo de seu presidente criticando duramente o Linux e empresas concorrentes (especificamente Wind River e Montavista) por comercializarem soluções baseadas em Linux. A crítica é que estas empresas possuem suas versões de Linux de tempo real, porém recomendam

[14] Maiores informações em: http://www.openflightlinux.org/.

aos clientes comprar seus outros sistemas operacionais de tempo real [O'Dowd, 2008]. O mais interessante é que a própria *Green Hills Software* também possui ferramentas que podem trabalhar com Linux [Williston, 2008], mesmo possuindo um site com artigos sobre problemas do Linux em aplicações críticas (http://www.ghs.com/linux.html).

Em geral, a maioria das versões de Linux de tempo real são baseadas em conceitos já antigos e consolidados, que também são usados para virtualização de computadores e servidores [Zhang et al., 2006]. Trata-se do ADEOS, que já foi apresentado anteriormente nesta pesquisa.

Existem muitas versões de Real Time Linux, como o *Kansas University Real Time Linux* (KURT) que não usa o conceito de ADEOS [Dinkel et al., 2002], mas tenta alterar todo *kernel* do Linux para suportar tempo real. O Montavista [Montavista, 2008], que foi estudado pela *Dedicated Systems* [Dedicated Systems, 2003b], e foi criticado por necessitar de *patches* para ser realmente de tempo real, e apresentar várias falhas. A empresa que desenvolve o Montavista Linux respondeu ao relatório com uma carta dizendo que discorda do relatório, e que o Montavista Linux é um sistema operacional para tempo real *soft*.

Outras versões incluem o RTE Linux [Wang e Lin, 1998], o BlueCat Linux, do mesmo fabricante do LynxOS [LynxWorks, 2006] e o WindLinux da Wind River. Uma outra versão muito utilizada na indústria é o RTLinux (o nome RTLinux causa bastante confusão entre o produto chamado RTLinux e outras versões de Real Time Linux) da FSMLabs. Entretanto, em 2007 a Wind River adquiriu a propriedade intelectual do RTLinux [Systems, 2007] e passou a ser proprietária deste produto, que antes possuía uma versão gratuita e uma comercial, e passou a possuir apenas uma versão comercial.

Ao longo dos testes, a versão de Linux de tempo real analisada será o RTAI. De acordo com Kim e Ambike, a performance do RTAI é comparável a dos melhores RTOS, como VxWorks e QNX [jong Kim et al., 2006], possuindo determinismo suficiente para substituí-los [Barbalace et al., 2008]. Além disto, o RTAI já foi utilizado para implementar ferramentas de *Hardware in the Loop* (HIL) de tempo real [Lu et al., 2003], que permite realizar simulações mais reais e poderosas, aliando ferramentas de *hardware* e *software* em simulação.

Outro caso de sucesso do RTAI está no telescópio SOAR, que possui o posicionamento de seu espelho de 4 metros de diâmetro controlado pelo RTAI, enquanto a interface de usuário é implementada em LabView. A comunicação entre Labview e real time linux é feita via TCP/IP [Schumacher et al., 2004]. Outros projetos como o *Real Time eXperiment Interface* (RTXI) [Christini, 2008b] e o RTLAB [Christini, 2008a] apóiam-se no RTAI, e já fizeram sucesso com vários experimentos biológicos, incluindo até mesmo experimentos com tecidos cardíacos [Culianu e Christini, 2003].

O principal marco na área de tempo real para Linux foi introduzido por Yodaiken e Barabanov. Como o *kernel* do Linux não é todo preemptivel, a idéia deles baseou-se no conceito de máquinas virtuais e ADEOS, preemptando todo o *kernel* do Linux em favor de tarefas de tempo real. Esta abordagem

faz com que sobrecargas no Linux, independente da sua natureza não afetem as tarefas de tempo real [Yodaiken e Barabanov, 1997]. A primeira implementação de RTLinux tinha o pior caso de latência de tratamento de interrupção de $15\mu s$ [Yodaiken, 1999]. Dessa forma, em 1997 Barabanov propôs em sua dissertação de mestrado, implementar um sistema híbrido, onde o Linux continuaria funcionando normalmente, com extensões de tempo real [Barabanov, 1997]. Na verdade, mesmo antes do trabalho de Barabanov, Yodaiken já realizava pesquisas nesta área, inclusive possuindo uma patente [Yodaiken, 1997] do funcionamento do RTLinux depositada em 1997.

Em seu trabalho, Barabanov realizou um experimento muito interessante, em que ele implementou um programa bastante simples que gerava uma freqüência de valor fixo no auto-falante de um PC. Quando o computador não estava realizando muitas operações, este som era homogêneo, entretanto ao digitar no teclado, ou clicar no mouse, interrupções são geradas, e este programa é suspenso pelo sistema operacional, fazendo o som parar e voltar, de forma a não ser homogêneo a ponto de ser perceptível pelo ouvido humano. Com base neste cenário, foi implementado um *kernel* de tempo real para o Linux, com o intuito resolver este problema.

Através de uma proposta feita pelo seu orientador (Victor Yodaiken), Baranov implementou um sistema que emula interrupções, de forma que para se ter o real time Linux, é preciso aplicar um *patch* ao Linux. Este *patch* procura por instruções como CLI, STI e IRET, que manipulam interrupções, e substitui estas instruções por macros, de forma que o *kernel* do Linux nunca desabilita, habilita ou retorna de uma interrupção real de *hardware* [Barabanov, 1997], mas apenas informa ao *nanokernel* de tempo real estes estados.

Outro problema que foi enfrentado em seu trabalho, estava no fato das tarefas de tempo real estarem inicialmente em modo de usuário, de forma que o *overhead* de uma chamada de sistema, levava cerca de 71 ciclos de *clock*, comprometendo a performance de tempo real [Barabanov, 1997]. Com isto, foi decidido pela implementação de tarefas de tempo real como módulos do *kernel*, que podem ser carregados e descarregado do *kernel* em execução a qualquer momento. Esta abordagem incide nas mesmas características do VxWorks: como o programa de tempo real está em modo de *kernel*, as trocas de contexto são menos custosas e o compartilhamento de informações e variáveis são transparentes e simples, entretanto o sistema fica mais inseguro.

O escalonamento das tarefas no Real Time Linux proposto por Barabanov e Yodaiken utiliza o algoritmo *Rate Monotonic*, onde cada tarefa possui uma prioridade. Sempre que uma tarefa de prioridade mais alta do que a que está em execução fica pronta, ela imediatamente recebe o controle do processador.

Um aspecto importante desta implementação, é que todo resto do sistema operacional Linux, incluindo a interface gráfica, e interface com o usuário é inserido no escalonador de tempo real como sendo a tarefa de tempo real de menor prioridade do sistema. Dessa forma, apenas quando todas tarefas de tempo real tiverem cumprido suas tarefas e estiverem com suas atividades em dia, o restante do Linux é executado

[Barabanov, 1997]. Esta abordagem é excelente, já que a maior parte dos sistemas de tempo real, consistem em uma combinação de tarefas com requisitos *soft* e *hard real time* [Labrosse, 2002]. Com isto seria possível usar um único computador tanto para implementar funções de tempo real críticas quanto interface com o usuário.

Além disto, o escalonador EDF também foi implementado para o RTAI, e pode ser usado bastando realizar uma troca de módulos. Adicionalmente, podem ser usadas diferentes implementações do EDF [Garcia e Silly, 2003].

Logo depois de sua publicação e disponibilização gratuita, os primeiros usuários começaram a reportar casos de sucesso de uso do *Real Time* Linux, na aquisição de dados a 3KHz em experimentos com ratos de laboratórios, para monitorar pacientes em um hospital, e para controlar um carro com 70 sensores sendo lidos a 80Hz.

A validação deste Linux de tempo real foi realizada utilizando dois computadores: um deles gerando estímulos na porta paralela e aguardando retorno, para medir quanto tempo o outro computador levou para responder em um ambiente de testes semelhante ao utilizado nesta pesquisa, entretanto trocando o gerador de sinais e osciloscópio por outro computador. Os resultados obtidos em testes realizados em um PC 80486 de 120MHz reduziram o pior caso de latência para tratar interrupções no Linux convencional de $2494\mu s$ para $34\mu s$ no *kernel* do Linux utilizado na época.

O *Real Time* Linux possui dois principais mecanismos de comunicação entre processos (*Inter Process Communication*, ou IPC). O primeiro consiste de filas de tempo real (*Real Time* FIFOS) e o segundo consiste da memória compartilhada (*shared memory*). Os RT-Fifos são interessantes pois fornecem sincronização e enfileiramento, entretanto oferecem apenas um canal unidirecional de dados. Para comunicação bidirecional é preciso usar pelo menos 2 FIFOs. Já a memória compartilhada não oferece sincronização, mas é um mecanismo bastante rápido, que permite mais de duas aplicações trocarem dados [Cvetkovic e Jevtic, 2003].

Um mecanismo interessante disponível no RTAI chama-se disparo antecipado (*antecipated firing*). Conhecendo valores de latência do *hardware,* este sistema dispara eventos um pouco antes do seu prazo, na tentativa de que o prazo seja cumprido com segurança devido a latências e *overheads* do *hardware*. Entretanto, este valor em alguns casos precisa ser configurado manualmente pelo usuário de acordo com cada *hardware*, pois latências negativas podem acabar ocorrendo, caso o disparo do evento seja muito antecipado.

Por tratar-se de um sistema gratuito, muitos testes e informações sobre o RTAI estão disponíveis. Seu *jitter* é considerado baixo no pior caso (inferior a $10\mu s$) [Aarno, 2004], e seu tempo de chavear tarefas é de $4\mu s$. Suas latências variam entre $4\mu s$ e $20\mu s$ [Köker, 2007]. A resolução do relógio no RTAI é inferior a $5\mu s$, oferecendo um relógio confiável com resolução de $5\mu s$ [jong Kim et al., 2006].

Por padrão, as aplicações de *hard real time* no RTAI têm que ser desenvolvidas em *kernel space*. Isto

faz com que uma falha no *software* de tempo real faça o sistema todo travar [Sarolahti, 2001]. Este fato é ainda mais grave sabendo-se que em *kernel space*, boa parte dos serviços do *kernel*, bem como a proteção da MMU não estão disponíveis [Leroux e Schaffer, 2006]. Uma alternativa é utilizar o LXRT, um sub-módulo do RTAI que permite rodar aplicações de tempo real em *user space* com requisitos de *hard real time*. Entretanto, as latências do LXRT são tipicamente maiores do que se o *software* tivesse sido feito em *kernel space* [Sarolahti, 2001]. De qualquer forma, em um estudo para verificar a viabilidade de utilizar-se o Linux para uma tarefa de controle *hard real time* a 100Hz, Laurich concluiu que o *kernel* 2.4 do Linux não pode ser usado para tarefas de tempo real, entretanto tanto o RTAI quando o RTAI com LXRT podem ser considerados determinísticos para sistemas de tempo real em 100% das ocasiões [Laurich, 2004].

Como poderá ser observado nos resultados demonstrados a seguir, o RTAI é bastante estável e confiável para implementar tarefas de tempo real. Dessa forma, um dos produtos resultantes deste trabalho consiste de um *Live CD* que permite implementar e testar facilmente sistemas com RTAI. Maiores detalhes sobre este CD estão disponíveis no capítulo 6.2, sendo que a os testes do RTAI foram realizados utilizando este CD, bastando iniciar o computador com o CD, e executar os programas de teste.

Para sobrecarregar o sistema, foram criados 263 tarefas até obter-se um *load average* de 16. Ao executar o *ping flood*, o Linux respondeu com sucesso a praticamente 100% das requisições, perdendo apenas 1 pacote.

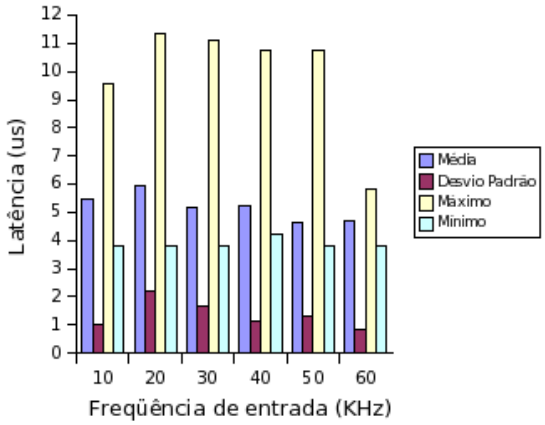
A tabela 13 mostra o resultado da máxima freqüência que foi possível medir com o RTAI, levando a um tempo máximo de resposta de $5\mu s$. Freqüências de até 1MHz foram geradas, sem afetar a estabilidade do sistema, entretanto enquanto o sistema era capaz de medir com sucesso o sinal de 200KHz, a performance de tarefas que não eram de tempo real foi severamente prejudicada. Ao tentar abrir um navegador *web (web browser)*, foram necessários 13 minutos desde de o clique em seu ícone, até o programa estar aberto.

Estado do sistema	Máximo	Mínimo	Média	Desvio Padrão
Operação Normal	200016Hz	199959Hz	199991Hz	14Hz
Sistema Sobrecarregado	199803Hz	199703Hz	199760Hz	21Hz
Ping Flood	198451Hz	192524Hz	52675Hz	2520Hz

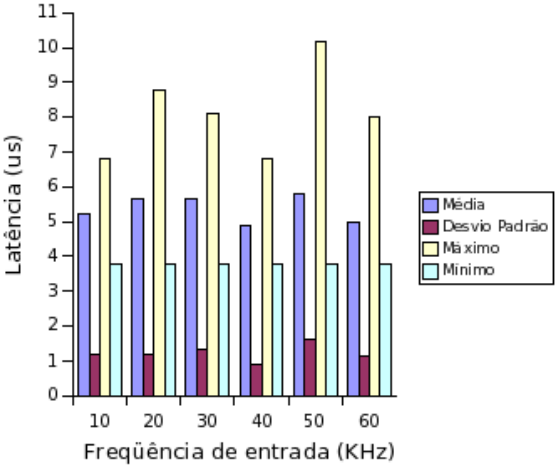
Tabela 13: Freqüências medidas pelo RTAI-Linux a partir de um sinal de entrada de 200KHz

As medidas de tempos máximos para executar operações de entrada e saída não foram realizadas pelo fato do RTAI utilizar as mesmas funções de entrada e saída *outb()* e *inb()* que o Linux convencional. Como estes testes já foram realizados, optou-se por não repeti-los, sendo que seus resultados podem ser vistos na tabela 12.

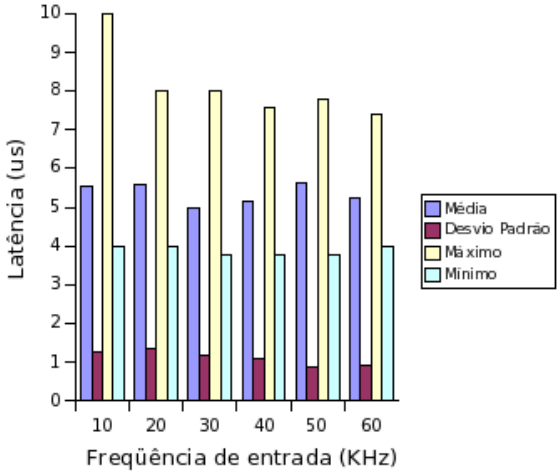
Com relação ao teste de latências para tratar interrupções, seus resultados podem ser vistos na figura 35. A pior latência obtida foi de $11{,}4\mu s$, sendo o *jitter* máximo no cenário (a) de $7{,}01\mu s$, no cenário (b)



(a) Operação normal



(b) Sistema sobrecarregado



(c) Com *ping flood*

Figura 35: Latências para tratar interrupções no RTAI-Linux

de 6,8μs e no cenário (c) de 4,01μs.

Pelos testes realizados, pode-se observar que a repartição do sistema em domínios deu bastante estabilidade e determinismo ao ambiente, de forma que as tarefas do domínio de tempo real tiveram alta prioridade e estabilidade. Além disto, o sistema demonstrou valores de latência e *jitter* bastante baixos e compatíveis com recomendações da OMAC para sistemas de tempo real.

Finalmente, uma discussão curiosa ocorreu em outubro de 2007 com relação ao RTAI. Quando questionado se o RTAI seria capaz de implementar um sistema *hard real time*, o autor do RTAI, Paulo Mantegazza respondeu com a seguinte afirmação:

> *"There is no math proof that RTAI is hard real time"*

Esta discussão está disponível em http://mail.rtai.org/pipermail/rtai/2007-October/018134.html, mas a afirmação é muito interessante, e deveria ser estendida para a maior parte dos sistemas operacionais, pois

como foi discutido amplamente neste trabalho, é muito difícil provar matematicamente que um sistema operacional seja *hard real time*.

7.10 VxWorks

O VxWorks é o RTOS comercial mais usado na indústria de sistemas embarcados [Ip, 2001, Cedeno e Laplante, 2007, Timmerman, 2000b, Timmerman, 2000a]. O VxWorks é usado na estação espacial internacional [Cedeno e Laplante, 2007], e nos famosos robôs *Spirit* e *Opportunity* que foram para Marte. Na verdade, o uso do VxWorks parece ser constante em aplicações de exploração espacial feitas pela NASA [Stumpf, 2003].

É interessante o fato de que inicialmente, o VxWorks não foi projetado para aplicações *"safety critical"* [Romanski, 2002] e embora atualmente ele seja consolidado na indústria de aviação, ele foi reprovado ao ser submetido pela primeira vez ao processo de certificação para estar de acordo com a norma DO-178B [Romanski, 2002].

O VxWorks usa um *microkernel*, mas não é baseado em mensagens como o QNX. Todo espaço de endereçamento de memória é único, de forma que o *kernel* não oferece proteção de memória com relação ao isolamento entre as tarefas [Barabanov, 1997]. Para obter-se este recurso, é necessário comprar uma extensão chamada VxVMI para que a memória seja segmentada e protegida [Martin Timmerman, 2000f], ou comprar o VxWorks versão AE que inclui está funcionalidade, pensando no mercado de aviação, onde este recurso é obrigatório.

Com esta arquitetura onde não há divisão entre espaço de usuário e de *kernel*, as tarefas não precisam fazer chamadas de sistema para o *kernel*, e podem facilmente compartilhar dados na memória, já que cada tarefa pode acessar qualquer região de memória. Estas características tornam o desenvolvimento mais fácil, e o sistema bem mais rápido, graças a trocas de contexto muito mais simples, e eliminando mecanismos de proteção de memória. Contudo, estes recursos fazem com que o sistema seja muito mais frágil, pois erros em um módulo, podem afetar outros, ou mesmo todos módulos do sistema [Barabanov, 1997].

Nos testes da *Dedicated Systems* em um sistema embarcado com um processador Pentium de 200MHz, obteve-se um pior caso de latência de 2,7μs, entretanto eles observaram que a interrupção de clock (a de maior prioridade no sistema) leva 4μs para ser tratada, e chegou a levar 20μs em uma ocasião especial, onde o sistema teve que recuperar uma inversão de prioridade, de forma que a conclusão é que o VxWorks é rápido e previsível, com críticas à proteção de memória [Dedicated Systems, 2001g].

Com relação ao VxWorks AE, ele recebeu nota 5, de uma máxima 10 na avaliação da *Dedicated Systems*, devido a alguns *bugs* encontrados durante os testes [Dedicated Systems, 2002f]. A Wind River reparou rapidamente estes *bugs*, e divulgou uma carta dizendo não concordar com a avaliação, pois alguns comentários da avaliação dependem da configuração feita pelo desenvolvedor [Dedicated Systems, 2003a].

Um resultado muito interessante desta interação, é que graças a uma crítica da *Dedicated Systems* dizendo que o VxWorks poderia em alguns casos levar mais de $60\mu s$ apenas para processar uma interrupção de *timer*, levou a Wind River a disponibilizar um *patch* permitindo que a própria interrupção de *clock tick* seja preemptada por alguma de importância maior, o que melhorou estes tempos, entretanto em alguns casos pode até ser perigoso.

Em uma comparação realizada em 2001, Ip conclui que o VxWorks é mais determinístico e previsível que o Real Time Linux [Ip, 2001]. Em outro estudo, o VxWorks foi o sistema operacional que apresentou menor latência para tratar interrupções [Yearraballi, 2000].

Um fato curioso ocorreu com um robô que foi para Marte, e ajudou a demonstrar uma qualidade incrível do VxWorks. Quando os robôs chegaram no planeta Marte [NASA, 2004], um *bug* de *software* acabou ocorrendo, e o sistema tornou-se instável reiniciando constantemente. Através de um sistema de depuração remoto do VxWorks, foi possível diagnosticar e corrigir o problema no *software* do sistema a 70 milhões de quilômetros de distância. O problema consistia de uma inversão de prioridade que foi facilmente resolvido ativando o protocolo de herança de prioridades disponível no sistema, mas que não havia sido utilizada previamente pelos desenvolvedores.

Com relação aos testes realizados, o sistema utilizado foi o VxWorks 6.2, e a programação foi feita com o novo IDE da Wind River, o *Workbench* 2.4, que é baseado no Eclipse. Houve diversas dificuldades para iniciar o VxWorks em um PC convencional utilizando seu BSP para PCs. A primeira dificuldade foi com relação ao processo de inicialização do sistema que não é muito bem documentada, sendo que foi necessário compilar uma imagem de *boot* do sistema e copiá-la para um disquete com ferramentas especiais que tornam este disquete um disco de inicialização. Após o computador ser iniciado com este disquete, toda interação passa a ser via porta serial, já que supõe-se que sistemas embarcados de tempo real não têm teclado e monitor. Se necessário existe uma opção para redirecionar a saída da porta serial para a tela do computador.

Após iniciar com o disquete, um *prompt* de comando é exibido para configurar como será o *boot* do sistema. A forma mais fácil seria escolher um *kernel* do VxWorks que está no disco, mas uma imagem do sistema não cabe neste disco de inicialização, e não existem opções para iniciar a partir de um CD. Dessa forma, é necessário utilizar a opção de descarregar o *kernel* via rede a partir de um servidor FTP. A dificuldade nesta etapa, foi em obter uma placa de rede compatível com o sistema. Em primeiro lugar, a placa a ser utilizada precisa ser escolhida diretamente editando um arquivo de *include (.h)* fonte do VxWorks. Neste arquivo, é possível descomentar a linha referente à placa que deseja-se utilizar. Como o sistema somente suporta 9 modelos diferentes de placas de rede, foi necessário obter uma placa de rede da 3COM que fosse compatível.

Após o *kernel* ser iniciado com sucesso, é possível utilizar o *shell* do VxWorks via rede IP com o comando telnet ou via *console* serial, e o sistema de arquivos do servidor de FTP fica automaticamente

Listagem 6 Erro resultante ao gerar uma freqüência superior a 280KHz

```
workQPanic: Kernel work queue overflow.
Exception at interrupt level:
Page Fault Page
Dir Base : 0x01443000
Esp0 0x00000000 : 0x00000000, 0x00000000, 0x00000000, 0x00000000
Esp0 0x00000010 : 0x00000000, 0x00000000, 0x00000000, 0x00000000
Program Counter : 0x0042cbd4
Code Selector : 0x00000020
Eflags Register : 0x00010246
Error Code : 0x00000000
Page Fault Addr : 0x00000040
Task: 0xbb7db00 "tRootTask"
Regs at 0x48a54c
```

disponível, e é possível listar os arquivos com o comando *ls* ou carregar qualquer programa na memória para execução que esteja no servidor através do comando *ld*.

Ao sobrecarregar o sistema para fazer os testes no cenário de sobrecarga, várias características interessantes foram observadas. A primeira é que após a primeira conexão via *telnet* ser realizada, o sistema não aceitou mais conexões, respondendo com a mensagem *"Sorry, session limit reached"*. Ao executar o *ping flood*, o VxWorks detectou uma quantia excessiva de requisições via rede, e exibiu no consolde uma mensagem dizendo que as respostas seriam limitadas (*Limiting icmp ping response from 280 to 200 packets per second)*. Com isto, observou-se no *host* que 21% das requisições ICMP (*ping)* foram perdidas. Estas precauções e limites, que podem ser alterados se necessário, certamente aumentam a confiabilidade e determinismo do sistema.

Como o VxWorks possui 256 níveis de prioridade [Cedeno e Laplante, 2007], optou-se por sobrecarregar o sistema criando cerca de 400 tarefas com prioridades entre 150 e 250 (várias tarefas podem ter prioridades iguais) que imprimem dados na tela e dormem uma pequena quantia de *ticks*. A execução destas tarefas pôde ser monitorada com os comandos *w*, *i* e *ti* que exibem informações detalhadas sobre o estado da tarefa.

O único problema encontrado em todos os testes realizados foi que ao gerar uma freqüência de entrada superior a 280KHz o sistema travou por completo, exibindo a mensagem da listagem 6. O erro persistiu todas as vezes que valores de freqüência superiores a este foram gerados. Após o erro, o *kernel* se reiniciou automaticamente, voltando à operação normal, sem ser necessário reiniciar o computador, entretanto as tarefas em execução foram perdidas.

Para realizar o teste de maior freqüência de entrada, uma tarefa periódica executada a cada um segundo foi criada com prioridade dez (10), e uma rotina de tratamento de interrupções foi implementada para incrementar o contador que é utilizado pela tarefa periódica para medir a freqüência de entrada.

Os resultados deste teste podem ser vistos na tabela 14, sendo que foi possível medir freqüências de

até 260KHz com bastante precisão, levando a um pior caso de resposta de 3,85μs. No pior cenário onde o sistema estava executando mais de 400 tarefas e recebendo o *ping flood*, o *shell* parou de responder a comandos, mas as freqüências foram medidas adequadamente. Neste pior cenário, uma única vez foi medida uma freqüência incorreta de 979695Hz, entretanto este valor foi eliminado do conjunto de dados por ser um *outlier*.

Estado do sistema	Máximo	Mínimo	Média	Desvio Padrão
Operação Normal	260029Hz	259447Hz	259806Hz	258Hz
Sistema Sobrecarregado	263808Hz	258413Hz	259850Hz	750Hz
Ping Flood	263808Hz	258413Hz	259848Hz	741Hz

Tabela 14: Freqüências medidas pelo VxWorks a partir de um sinal de entrada de 260KHz

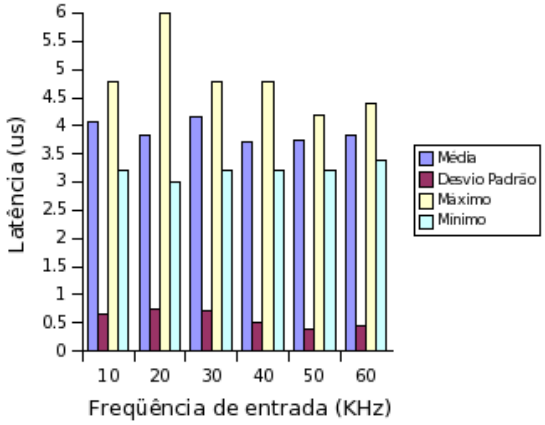
Com relação ao teste de latências para tratamento de interrupções o impacto da prioridade da tarefa é notável. Ao executar os testes de entrada e saída com prioridade 150, uma única chamada para leitura ou escrita em portas de entrada e saída chegou a levar 8μs, enquanto que com prioridade 10 os tempos não ultrapassaram 2,5μs.

Os resultados dos testes podem ser vistos na tabela 15, sendo que para implementar a sonda de *software* foi utilizada a função *PentiumTscGet* da biblioteca do VxWorks com funções especificas para processadores Pentium.

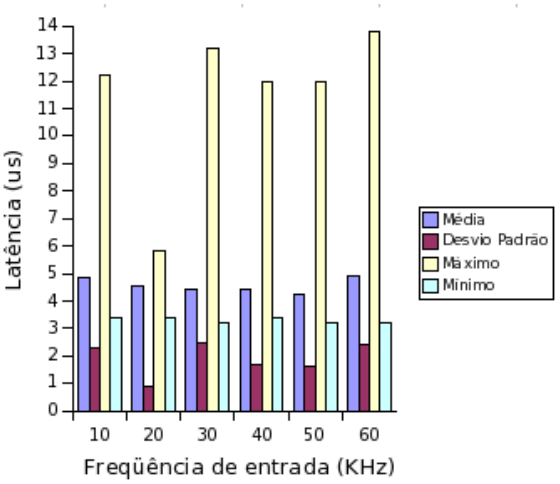
Operação/Condição de Execução	Dado	*IN* (μs)	*OUT* (μs)
Sistema em operação normal	Máximo	2,15	1,67
	Mínimo	1,39	1,4
	Média	1,44	1,42
	Desvio Padrão	0,15	0,04
Sistema sobrecarregado	Máximo	1,6	1,66
	Mínimo	1,45	1,42
	Média	1,49	1,48
	Desvio Padrão	0,06	0,1
Sistema sobrecarregado com *ping flood*	Máximo	1,6	1,66
	Mínimo	1,45	1,42
	Média	1,49	1,49
	Desvio Padrão	0,06	0,11

Tabela 15: Tempo para execução de funções de acesso a portas de entrada e saída no VxWorks 6.2

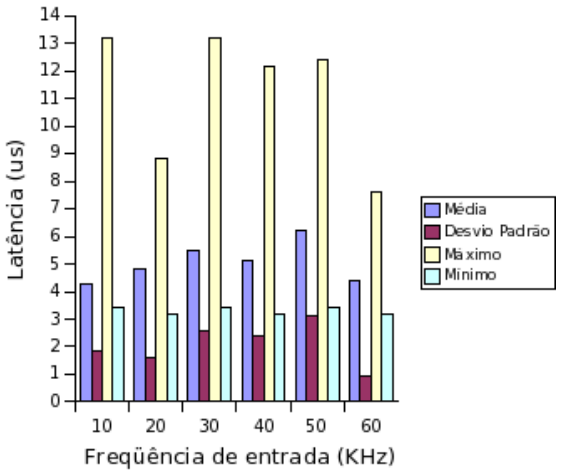
As latências observadas em seus vários cenários para tratar interrupções podem ser observadas na figura 36, sendo o pior *jitter* no cenário (a) de 4,4μs, no cenário (b) de 10,4μs, e no cenário (c) de 8,7μs. Além disto, a pior latência observada foi de 13,8μs.



(a) Operação normal



(b) Sistema sobrecarregado



(c) Com *ping flood*

Figura 36: Latências para tratar interrupções no VxWorks

A análise constatou que realmente o VxWorks é um sistema operacional de tempo real entre àqueles com tempo de resposta mais rápido, além de demonstrar variações de latência bastante baixas e compatibilidade com folga com as recomendações da OMAC. O único problema de instabilidade observado provavelmente é relativo ao pacote de suporte à placa (BSP) do Pentium e não ao sistema operacional em si, e só foi observado em altas freqüências.

7.11 Resumo

Neste capítulo, os sistemas operacionais de tempo real selecionados foram analisados em diversos aspectos. Os piores casos dos resultados obtidos podem ser vistos na tabela 16.

Nos testes realizados, todos sistemas operacionais analisados demonstraram um bom comportamento temporal de acordo com o critério sugerido pela OMAC [Hatch, 2006], onde o sistema não pode ter um *jitter* para tratamento de interrupções superior a $100\mu s$ em ciclos de 1ms, de forma que todos são adequados para executar tarefas de tempo real *hard*.

A única excessão seria o Windows XP, que apresentou um alto *jitter*, mas mesmo assim demonstrou bastante qualidade e robustez ao se utilizar a prioridade mais alta de seu escalonador (prioridade de Tempo Real), sendo possível executar tarefas *soft real time* com certa qualidade no Windows XP.

Alguns sistemas como o QNX e o Windows CE apresentaram valores um pouco acima dos demais provavelmente pela excessiva proteção do sistema e alto isolamento entre as tarefas, aumentando seu *overhead*, mas aumentando também sua confiabilidade.

	Windows XP	Windows CE	QNX Neutrino	μC/OS-II	Linux	RTAI	VxWorks
A	$200\mu s$	$20\mu s$	$20\mu s$	$1{,}92\mu s$	$13{,}89\mu s$	$5\mu s$	$3{,}85\mu s$
B	$700\mu s$	$88{,}8\mu s$	$32\mu s$	$2{,}32\mu s$	$77{,}6\mu s$	$7.01\mu s$	$10{,}4\mu s$
C	$848\mu s$	$99\mu s$	$35{,}2\mu s$	$3{,}2\mu s$	$98\mu s$	$11{,}4\mu s$	$13{,}4\mu s$
D	132ms	$524\mu s$	$62{,}95\mu s$	$2{,}12\mu s$	$5{,}26\mu s$	$5{,}26\mu s$	$1{,}67\mu s$
E	152ms	8,1ms	$3{,}77\mu s$	$1{,}74\mu s$	$5{,}72\mu s$	$5{,}72\mu s$	$2{,}15\mu s$

A: Pior tempo de resposta (1/freqüencia máxima sustentada)
B: Pior variação da latência (*jitter)*
C: Pior latência para tratar uma interrupção
D: Pior tempo para executar um OUT em uma porta de saída
E: Pior tempo para executar um IN em uma porta de entrada

Tabela 16: Comparação dos resultados obtidos

Outro fato curioso demonstrado no inicio do capítulo, é que um *hardware* mais antigo conseguiu executar algumas tarefas de tempo real com melhor performance que um mais moderno. De fato, em testes realizados em um processador 80386 de 20MHz, concluiu-se que sistemas operacionais compatíveis com Unix são adequados para para tarefas de controle embarcado até mesmo em processadores com relativa baixa performance [Kleines e Zwoll, 1996].

Com relação à estabilidade, os únicos sistemas que não travaram em nenhuma ocasião, dispensando a

necessidade de *reboot* foram o QNX, o Windows CE e o RTAI, sendo que eles resistiram e recuperaram-se plenamente de todas situações de *stress* impostas pelos testes. Um fato importante, é que os outros sistemas não são necessariamente inferiores por este motivo, pois eles também conseguiram antender a altas freqüências de entrada e responder às latências rapidamente. As falhas observadas podem estar relacionadas à implementação dos *device drivers* para a arquitetura PC nestes sistemas operacionais.

8 Considerações finais

8.1 Tendências e novos desafios para sistemas de tempo real

8.1.1 Gerenciamento de energia

Uma necessidade já atual e cada vez maior, é que os sistemas computacionais, e conseqüentemente os sistemas embarcados e de tempo real economizem energia devido a questões ambientais e econômicas. Dessa forma, os fabricantes embutem nos processadores e computadores cada vez mais recursos para gerenciamento e economia de energia, sendo que em alguns casos estes sistemas já estão ativados por padrão de fábrica.

Atualmente, os processadores podem ajustar dinamicamente sua velocidade conforme a carga de processamento, para esquentar menos e consumir menos energia. A velocidade do processador também volta a aumentar automaticamente quando existe demanda de processamento.

As arquiteturas modernas também contemplam mecanismos que desligam partes do sistema quando não estão em uso, e religam estes subsistemas conforme a necessidade de sua utilização.

Estes recursos têm impacto na performance de tempo real. O ajuste da velocidade do processador torna a medida do tempo usando TSC instável, e a latência até ligar os componentes que foram desligados pode prejudicar a resposta dos sistemas.

8.1.2 Multicore e paralelismo

Analisando os últimos 30 anos de pesquisa em tempo real, Laplante discute nove desafios atuais [Laplante, 2004], sendo 7 deles relativos a como fazer os escalonadores cumprirem seus prazos com confiabilidade quando utiliza-se duas ou mais unidades de processamento.

Esta necessidade torna-se ainda mais eminente com a grande tendência de utilização de processadores com vários núcleos, já que a velocidade dos processadores parou de crescer rapidamente. Isto ocorre pois o consumo de energia para aumentar poucos Hertz na CPU é alto, tornando cada vez mais complicado aumentar a velocidade com um bom custo beneficio de energia. Dessa forma, a solução adotada pelas líderes Intel e AMD tem sido desenvolver processadores com vários núcleos [Ross, 2008].

O processador *Cell* desenvolvido em conjunto pela IBM, Hitachi e Sony, foi apelidado de supercomputador em um chip, e possui nove núcleos de processamento [Taurion, 2005]. Para utilizar todos recursos disponíveis por este processador e outros processadores de vários núcleos, não somente os sistemas operacionais, mas em alguns casos os *softwares* dos usuários também devem se adaptar a esta nova realidade.

Um tipo de falha que pode ocorrer em *chips multicore*, ainda pouco estudados está na forma com que os componentes estão interconectados. Muitas vezes uma falha em uma unidade interna de um chip *multicore*, pode causar a falha do chip todo, comprometendo tarefas que não tinham relação com aquela que causou o problema [Aggarwal et al., 2007].

Outros problemas incluem como sincronizar vários processos, mantendo suas prioridades e recursos compartilhados, sendo executados realmente de forma simultânea em dois ou mais núcleos. Os semáforos e outros mecanismos do sistema operacional precisam garantir a integridade dos recursos entre vários processadores, e o uso de referências de tempo como o TSC podem se tornar não confiáveis, pois cada processador tem seu próprio TSC.

Um dos primeiros fornecedores a oferecer um RTOS com escalonamento para multiprocessamento simétrico (SMP) é a QNX [Geer, 2007]. O Linux também já suporta multiprocessamento, bem como o Windows e o VxWorks.

Uma abordagem interessante para multiprocessamento é utilizada no VxWorks desde 1990, em que o VxWorks trata um sistema multiprocessado ou em rede da mesma forma, de maneira que tanto faz o processador estar na mesma máquina ou em uma máquina distante conectado via rede TCP/IP. O VxWorks oferece primitivas que permitem que o processamento seja distribuído entre os processadores do sistema independente de qual a arquitetura de paralelismo do *hardware* [Fiddler et al., 1990].

Novos produtos também oferecem a possibilidade de executar um sistema operacional de tempo real e um de propósito geral simultaneamente em um processador *dual core*, melhorando a latência para tratamento de interrupção em um fator de dez vezes, quando comparado com um processador de apenas um núcleo [Fische, 2006].

8.1.3 Sistema operacional em hardware

Outra tendência interessante para sistemas operacionais de tempo real, seria executar parte do sistema operacional em *hardware*. Da mesma forma que atualmente o *hardware* do computador já executa parte do trabalho do sistema operacional, como no caso da MMU ao verificar se as partições de memória estão sendo respeitadas pelos programas, existe também a possibilidade de executar o próprio escalonador do sistema operacional em *hardware*. Esta possibilidade não só é possível como já demonstrou-se eficiente em estudos experimentais [Agron et al., 2006, Vetromille et al., 2006].

Kohout e Ganesh propõem o uso de subsistemas de *hardware* para auxiliar o RTOS ou até mesmo substituir partes do RTOS, liberando o processador apenas para o processamento das tarefas. Seus experimentos demonstraram ser possível diminuir em até 90% o consumo do processador por tarefas do RTOS, diminuindo o tempo máximo de resposta em até 81% [Kohout et al., 2003].

8.1.4 Microcontroladores

Em anos recentes, o uso de microcontroladores tornou-se muito popular, e estes dispositivos evoluíram a ponto de hoje terem mais capacidade de processamento e recursos que um computador pessoal possuía dez anos atrás.

Alguns microcontroladores além de baixo custo (custos inferiores a dez dólares), possuem recursos

que apenas os computadores mais modernos possuem, como vários níveis de *pipeline,* MMU e memória *cache*, por exemplo.

Entretanto, com grande número de recursos e alto poder de processamento, os programadores de microcontroladores começam a enfrentar o problema de ter que conhecer e dominar detalhes de *hardware* específicos do microcontrolador para o qual se está desenvolvendo alguma aplicação. Além disso, o paradigma de programação em microcontroladores normalmente consiste de *polled loops* ou sistemas reativos, que somente respondem mediante interrupções, porém as novas aplicações vêm exigindo a execução de várias tarefas simultâneas e gerenciamento de uso concorrente de recursos. Estes fatos, tornam cada vez mais interessante e prático utilizar-se um RTOS em um microcontrolador.

A empresa *Intellibot Robotics*, passou anos fornecendo um robô autônomo para limpeza de chão baseado em microcontrolador. Um de seus desenvolvedores admite que a transição do robô baseado em um programa baseado em microcontrolador para um RTOS, mais especificamente o Linux, tornou o processo de desenvolvimento muito mais barato e conveniente [Daly e Knuth, 2006]. Além de flexibilidade e facilidade de expansão do sistema, a empresa ainda ganhou a comodidade de poder facilmente migrar a plataforma computacional do sistema, sem ter que realizar nenhum tipo de reprogramação.

Sistemas como o VxWorks, QNX e Linux já estão disponíveis para microcontroladores ARM. Outros sistemas operacionais como o FreeRTOS e μC/OS-II estão disponíveis para processadores mais simples como AVR, PIC e processadores ARM com menos recursos.

Outra tendência é o uso de microcontroladores de 32 bits, como o ARM, PIC-32 e o AVR-32, onde é muito importante utilizar um sistema operacional, para aproveitar todos os recursos destes poderosos dispositivos.

8.2 Trabalhos futuros

Pelas tendências discutidas nas seções anteriores, propõe-se trabalhos futuros na área de sistemas operacionais de tempo real relacionados a gerenciamento de energia mantendo as características de um sistema de tempo real confiáveis, a implementação de partes dos sistemas operacionais em *hardware*, e novas pesquisas na área de processamento paralelo para tempo real, já que esta área possui demandas imediatas dada a grande popularização de processadores com vários núcleos.

8.3 Conclusões

Nesta pesquisa foram estudados diversos aspectos construtivos e internos de sistemas operacionais de tempo real, que fornecem a infra-estrutura básica para construir-se sistemas de tempo real com confiabilidade. Sem dúvida, os sistemas operacionais de tempo real são parte importante de um sistema de tempo real. Contudo, eles não são a panacéia dos sistemas de tempo real. De fato, existem situações em que o RTOS pode ser dispensado quando tem-se um projeto bem feito, e um desenvolvimento organizado

e de qualidade.

Entretanto, o simples uso de um RTOS não garante que o sistema irá possuir qualidade e confiabilidade. Exemplos trágicos incluem o equipamento de terapia por Raios-X chamado Therac-25, que causou a morte de pelo menos 5 pessoas por falhas de programação [Leveson et al., 1993]. Todo *software* do equipamento havia sido desenvolvido por uma única pessoa, sendo adaptado de uma versão anterior do produto, além do produto ter sido desenvolvido sem testes ou documentação. Problemas semelhantes podem acontecer com o uso de um sistema operacional, sendo sempre importante não apenas utilizar ferramentas de qualidade, mas também seguir critérios de desenvolvimento formais e organizados.

Por melhor que sejam os sistemas operacionais de tempo real e os sistemas de controle, eles não podem ajudar se não forem utilizados adequadamente. Embora esta afirmação possa parecer simples, sabe-se de muitos casos de catástrofes como o da usina nuclear de Chernobyl ocorreram por erro humano.

Além do estudo de características teóricas de *hardware* e dos sistemas operacionais para suportar sistemas de tempo real, uma análise metódica também foi realizada em diversos sistemas para obter-se os seus piores tempos de resposta em diversos cenários, já que o pior tempo de resposta é uma das medidas mais indicadas para avaliar a qualidade de sistemas de tempo real. Está análise possibilitou determinar se os sistemas estudados são confiáveis para executar o controle de sistemas mecatrônicos, como robôs e veículos.

O estudo demonstrou que sistemas para usuários finais como o Windows XP não são adequados para tarefas de tempo real *hard*, entretanto podem ter boa qualidade se utilizados com bom senso para tarefas de tempo real *soft*. Os demais sistemas estudados (VxWorks, QNX Neutrino, Linux, RTAI-Linux, μC/OS-II e Windows CE) demonstraram bom determinismo, estabilidade e confiabilidade nos testes realizados, sendo adequados para o controle de sistemas mecatrônicos. Uma observação importante realizada é que versões de Linux para tempo real mostraram-se como alternativas viáveis para alguns sistemas de tempo real em relação a sistemas comerciais como o VxWorks, o que também foi demonstrado em outro estudo recente [Barbalace et al., 2008].

O trabalho desenvolvido também resultou em um ambiente prático e de fácil utilização para implementar e aprender sistemas de tempo real, sem a necessidade de instalar nenhum sistema operacional, bastando ligar o computador com o *Live* CD desenvolvido. Outro resultado adicional, foi a implementação da plataforma de tempo real para o robô Scara, que demonstrou bastante confiabilidade e determinismo.

Neste tipo de estudo é comum colegas perguntarem "Qual é o melhor sistema?". A resposta é que não existe melhor sistema. Cada um tem diversas vantagens e desvantagens quando comparados a seus concorrentes, além de possuírem curvas de aprendizado e tempo de desenvolvimento diferentes também. Uma boa resposta seria que o melhor sistema operacional de tempo real, é aquele que o desenvolvedor do sistema domina.

Referências

[Aarno, 2004] Aarno, D. (2004). Control of a puma 560 using linux real-time application interface (rtai). *On-Line: http://www.nada.kth.se/ bishop/rtcontrol.pdf, Consultado em Jan/2008*.

[Adams, 2005] Adams, C. (2005). Product focus: Cots operating systems: Boarding the boeing 787. on-line. disponível em: http://www.avtoday.com/av/categories/commercial/832.html. consultado em agosto/2008. *Avionics Magazine*.

[Aggarwal et al., 2007] Aggarwal, N., Aggarwal, N., Ranganathan, P., Jouppi, N. P., e Smith, J. E. (2007). Isolation in commodity multicore processors. *Computer*, 40(6):49–59.

[Agron et al., 2006] Agron, J., Peck, W., Anderson, E., Andrews, D., Komp, E., Sass, R., Baijot, F., e Stevens, J. (2006). Run-time services for hybrid cpu/fpga systems on chip. *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, p. 3–12.

[Ambike et al., 2005] Ambike, A., jong Kim, W., e Ji, K. (8-10 June 2005). Real-time operating environment for networked control systems. *American Control Conference, 2005. Proceedings of the 2005*, p. 2353–2358 vol. 4.

[Amianti, 2008] Amianti, G. (2008). Arquitetura de software aviônico de um vant com requisitos de homologação. Dissertação de Mestrado, Escola Politécnica da Universidade de São Paulo.

[Aroca et al., 2007] Aroca, R. V., Aroca, R. V., Tavares, D. M., e Caurin, G. (2007). Scara robot controller using real time linux. In Tavares, D. M., editor, *Proc. IEEE/ASME International Conference on Advanced Intelliget Mechatronics*, p. 1–6.

[Bach, 1986] Bach, M. J. (1986). *The Design of The Unix Operating System*. Prentice-Hall International Editions.

[Barabanov, 1997] Barabanov, M. (1997). A linux-based realtime operating system. Dissertação de Mestrado, New Mexico Institute of Mining and Technology.

[Barbalace et al., 2008] Barbalace, A., Luchetta, A., Manduchi, G., Moro, M., Soppelsa, A., e Taliercio, C. (2008). Performance comparison of vxworks, linux, rtai, and xenomai in a hard real-time application. *Nuclear Science, IEEE Transactions on*, 55(1):435–439.

[Baskiyar e Meghanathan, 2005] Baskiyar, S. e Meghanathan, N. (2005). A survey of contemporary real-time operating systems. *Informatica*, 29(29):233–240.

[Basumallick e Nilsen, 1994] Basumallick, S. e Nilsen, K. (1994). Cache issues in realtime systems. In *ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*.

[Beal, 2005] Beal, D. (2005). Linux® as a real-time operating system. Relatório técnico, freescale semiconductor.

[Beck et al., 1997] Beck, M., Böhme, H., Dziadzka, M., Kunitz, U., Magnus, R., e Verworner, D. (1997). *Linux Kernel Programming*. Addison-Wesley.

[Beneden, 2001] Beneden, B. V. (2001). Windows ce 3.0: Breathing down rtos vendors' necks. *Dedicated Systems Magazine*.

[Bergman, 1991] Bergman, G. (1991). The accuracy of interrupt driven time measurements. In *Software Engineering for Real Time Systems, 1991., Third International Conference on*, p. 217–220.

[Binns, 2001] Binns, P. (2001). A robust high-performance time partitioning algorithm: the digital engine operating system (deos) approach. *Digital Avionics Systems, 2001. DASC. The 20th Conference*, 1:1B6/1–1B6/12 vol.1.

[Bollinger, 2003] Bollinger, T. (2003). Use of free and open-source software (foss) in the u.s. department of defense. Relatório técnico, MITRE Corporation.

[Bouyssounouse, 2005] Bouyssounouse, B. (2005). *Embedded Systems Design*, chapter Real-Time Operating Systems, p. 258–286. Springer Berlin / Heidelberg.

[Bruyninckx, 2002] Bruyninckx, H. (2002). Orocos: design and implementation of a robot control software framework. ICRA 2002.

[Bruyninckx et al., 2001] Bruyninckx, H., Troyer, P. D., e Gadeyne, K. (2001). An open source hands-on course with real-time linux. In *Third Real-Time Linux Workshop*.

[Brylow e Palsberg, 2004] Brylow, D. e Palsberg, J. (2004). Deadline analysis of interrupt-driven software. *IEEE Transactions on Software Engineering*, 30(10):634–655.

[Burns, 1991] Burns, A. (1991). Scheduling hard real-time systems: A review. *Software Engineering Journal*, p. 116+.

[Caccavale et al., 2005] Caccavale, F., Lippiello, V., Siciliano, B., e Villani, L. (2-6 Aug. 2005). Replics: an environment for open real-time control of a dual-arm industrial robotic cell based on rtai-linux. *Intelligent Robots and Systems, 2005. (IROS 2005). 2005 IEEE/RSJ International Conference on*, p. 2493–2498.

[Calandrino e Anderson, 2006] Calandrino, J. e Anderson, J. (2006). Quantum support for multiprocessor pfair scheduling in linux. In *Proceedings of the Second International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, p. 36–41.

[Carlow, 1984] Carlow, G. D. (1984). Architecture of the space shuttle primary avionics software system. *Commun. ACM*, 27(9):926–936.

[Cawlfield, 1997] Cawlfield, D. W. (1997). Achieving fault-tolerance with pc-based control. Relatório técnico, OMNX Open Control, Olin Corporation, OMNX Open Control, Olin Corporation Charleston, TN 37310-0248 dwcawlfield@corp.olin.com.

[Cedeno e Laplante, 2007] Cedeno, W. e Laplante, P. A. (2007). An overview of real-time operating systems. *Journal of the Association for Laboratory Automation*, 12:40–45.

[Centioli et al., 2004] Centioli, C., Iannone, F., Mazza, G., Panella, M., Pangione, L., Vitale, V., e Zaccarian, L. (2004). Open source real-time operating systems for plasma control at ftu. *Nuclear Science, IEEE Transactions on*, 51(3):476–481.

[Chatterjee e Strosnider, 1996] Chatterjee, S. e Strosnider, J. K. (1996). Quantitative analysis of hardware support for real-time operating systems. *Real-Time Systems*, 10(2):123–142.

[Chelf, 2001] Chelf, B. (2001). A peek inside the clock. *Linux Magazine*.

[Cherry, 2007] Cherry, S. (2007). Robots incorporated. *IEEE Spectrum*, 44(8):24–29.

[Christini, 2008a] Christini, D. (2008a). Real-time linux biological-experiment control project. On-Line. Disponível em: http://rtlab.org/index.php. Acesso em Maio/2008.

[Christini, 2008b] Christini, D. (2008b). Rtxi - real-time experiment interface. On-Line. Disponível em: http://rtxi.org/index.jsp. Acesso em Maio/2008.

[Cinkelj et al., 2005] Cinkelj, J., Mihelj, M., e Munih, M. (20 May 2005). Soft real-time acquisition in windows xp. *Intelligent Solutions in Embedded Systems, 2005. Third International Workshop on*, p. 110–116.

[Communications, 2007] Communications, S. (2007). Measuring jitter accurately. On Line: http://www.spirentcom.com/documents/4814.pdf, Consultado em Julho/2007.

[Culianu e Christini, 2003] Culianu, C. e Christini, D. (2003). Real-time linux experiment interface system: Rtlab, bioengineering conference, ieee 29th annual, proceedings of , vol., no.pp. 51-52, 22-23.

[Cvetkovic e Jevtic, 2003] Cvetkovic, M. e Jevtic, M. (2003). Interprocess communication in real-time linux. *Telecommunications in Modern Satellite, Cable and Broadcasting Service. TELSIKS 2003. 6th International Conference on*, 2:618–621 vol.2.

[Daly e Knuth, 2006] Daly, D. e Knuth, D. (2006). Porting an existing embedded system to linux. Relatório técnico, Intellibot Robotics, LLC,.

[de Oliveira, 2003] de Oliveira, R. S. (2003). Aspectos construtivos dos sistemas operacionais de tempo real. In *21o Simpósio Brasileiro de Redes de Computadores (SBRC) - V Workshop de Tempo Real (WTR)*.

[de Oliveira et al., 2001] de Oliveira, R. S., da Silva Carissimi, A., e Toscani, S. S. (2001). *Sistemas Operacionais*. Editora Sagra Luzzatto, 3 edition.

[Dedicated Systems, 2001a] Dedicated Systems (2001a). Comparison between hyperkernel4.3, rtx4.2 and intime1.20. Relatório técnico, Dedicated Systems.

[Dedicated Systems, 2001b] Dedicated Systems (2001b). Evaluation report definition. Relatório técnico, Dedicated Systems.

[Dedicated Systems, 2001c] Dedicated Systems (2001c). Hyperkernel 4.3. Relatório técnico, Dedicated Systems.

[Dedicated Systems, 2001d] Dedicated Systems (2001d). Intime 1.20. Relatório técnico, Dedicated Systems.

[Dedicated Systems, 2001e] Dedicated Systems (2001e). Qnx 4.25. Relatório técnico, Dedicated Systems.

[Dedicated Systems, 2001f] Dedicated Systems (2001f). Rtx 4.2. Relatório técnico, Dedicated Systems.

[Dedicated Systems, 2001g] Dedicated Systems (2001g). Vxworks/x86 5.3.1. Relatório técnico, Dedicated Systems.

[Dedicated Systems, 2001h] Dedicated Systems (2001h). Windows nt workstation 4.0. Relatório técnico, Dedicated Systems.

[Dedicated Systems, 2002a] Dedicated Systems (2002a). Comparison between qnx rtos v6.1, vxworks ae 1.1 and windows ce .net. Relatório técnico, Dedicated Systems.

[Dedicated Systems, 2002b] Dedicated Systems (2002b). Qnx neutrino 6.2, vxworks ae 1.1, windows ce .net and elds 1.1 compared. Relatório técnico, Dedicated Systems.

[Dedicated Systems, 2002c] Dedicated Systems (2002c). Qnx neutrino rtos v6.2. Relatório técnico, Dedicated Systems.

[Dedicated Systems, 2002d] Dedicated Systems (2002d). Qnx rtos v6.1. Relatório técnico, Dedicated Systems.

[Dedicated Systems, 2002e] Dedicated Systems (2002e). Red hat embedded linux developer suite (elds) v1.1 (x86). Relatório técnico, Dedicated Systems.

[Dedicated Systems, 2002f] Dedicated Systems (2002f). Vxworks ae 1.1 (x86). Relatório técnico, Dedicated Systems.

[Dedicated Systems, 2002g] Dedicated Systems (2002g). Windows ce 3.0. Relatório técnico, Dedicated Systems.

[Dedicated Systems, 2002h] Dedicated Systems (2002h). Windows ce .net x86. Relatório técnico, Dedicated Systems.

[Dedicated Systems, 2003a] Dedicated Systems (2003a). Amendment to the vxworks ae 1.1 (x86) evaluation resport.

[Dedicated Systems, 2003b] Dedicated Systems (2003b). Montavista linux professional edition 2.1. Relatório técnico, Dedicated Systems.

[del Foyo et al., 2006] del Foyo, L. E. L., Mejia-Alvarez, P., e de Niz, D. (2006). Real-time scheduling of interrupt requests over conventional pc hardware. *enc*, 0:27–36.

[Dinkel et al., 2002] Dinkel, W., Niehaus, D., Frisbie, M., e Woltersdorf, J. (2002). *KURT-Linux User Manual*. Kansas University.

[Dougan, 2008] Dougan, C. (2008). Building a test harness for rtos. *Dr. Dobbs Journal*.

[Dougan e Mwaikambo, 2004] Dougan, C. e Mwaikambo, Z. (2004). Lies, misdirection, and real-time measurements. *C/C++ Users Journal*.

[Dudek et al., 2007] Dudek, G., Dudek, G., Giguere, P., Prahacs, C., Saunderson, S., Sattar, J., Torres-Mendez, L.-A., Jenkin, M., German, A., Hogue, A., Ripsman, A. R. A. A., Zacher, J. Z. A. J., Milios, E. M. A. E., Liu, H. L. A. H., Zhang, P. Z. A. P., Buehler, M. B. A. M., e Georgiades, C. G. A. C. (2007). Aqua: An amphibious autonomous robot. *Computer*, 40(1):46–53.

[Dupré e Baracos, 2001] Dupré, J. K. e Baracos, P. (2001). Benchmarking real-time determinism. Relatório técnico, Instrumentation, Systems &. Automation Society (ISA).

[Farines et al., 2000] Farines, J.-M., da Silva Fraga, J., e de Oliveira, R. S. (2000). *Sistemas de Tempo Real*. Escola de Computação 2000, Florianópolis.

[Feofiloff, 2008] Feofiloff, P. (2008). Introdução informal à complexidade de problemas. On Line. Disponível em: http://www.ime.usp.br/ pf/analise_de_algoritmos/aulas/NPcompleto.html. Consultado em Agosto/2008.

[Fiddler et al., 1990] Fiddler, J., Wilner, D., e Wong, H. (26 Feb-2 Mar 1990). Multiprocessing: an extension of distributed, real-time computing. *Compcon Spring '90. Intellectual Leverage. Digest of Papers. Thirty-Fifth IEEE Computer Society International Conference.*, p. 216–218.

[Fische, 2006] Fische, P. (2006). About hardware virtualization features and real-time hypervisor software. *Embedded Control Europe - ECE*, p. 33–34.

[Foundation, 2007] Foundation, F. S. (2007). Gnu general public license.

[Franke, 2007] Franke, M. (2007). A quantitative comparison of realtime linux solutions. Relatório técnico, Chemnitz University of Technology.

[FreeRTOS, 2003] FreeRTOS (2003). *FreeRTOS - Memory Management. On Line. Disponível em: http://www.freertos.org/a00111.html, Consultado em 01/Maio/2008.*

[Ganssle, 2004] Ganssle, J., editor (2004). *The Firmware Handbook*. Elsevier.

[Ganssle, 2006] Ganssle, J. (2006). Rtos dissatisfaction. *Embedded.com*.

[Garcia e Silly, 2003] Garcia, T. e Silly, M. (10-12 June 2003). Scheduling a robotic real-time application. *Control and Automation, 2003. ICCA '03. Proceedings. 4th International Conference on*, p. 600–604.

[Geer, 2007] Geer, D. (2007). For programmers, multicore chips mean multiple challenges. *Computer*, 40(9):17–19.

[Gerum, 2005] Gerum, P. (2005). Life with adeos.

[Ghodoussi et al., 2002] Ghodoussi, M., Butner, S., e Wang, Y. (2002). Robotic surgery - the transatlantic case. *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, 2:1882–1888 vol.2.

[Gleick, 1999] Gleick, J. (1999). The way we live now: 3-21-99 – product broken windows theory. *The New York Times*.

[González et al., 2003] González, J., Baeyens, E., Gayubo, F., Turiel, J., e Fraile, J. (2003). Open architecture controller for a scara yamaha yk7000 industrial robot. In *Fifth Real-Time Linux Workshop*.

[Grier, 2007a] Grier, D. A. (2007a). The boundaries of time. *Computer*, 40(7):5–7.

[Grier, 2007b] Grier, D. A. (2007b). Working class hero. *Computer*, 40(5):8–10.

[Grottke e Trivedi, 2007] Grottke, M. e Trivedi, K. S. (2007). Fighting bugs: Remove, retry, replicate, and rejuvenate. *Computer*, 40(2):107–109.

[Guimarães, 2008] Guimarães, K. S. (2008). Introdução à np-completude. On Line. Disponível em: http://www.cin.ufpe.br/ katiag/cursos/pos/ComputCientifica_ModuloAlgoritmos/NocoesNPC.ppt. Consultado em Agosto/2008.

[Hajdukovic et al., 2003] Hajdukovic, M., Suvajdzin, Z., Zivanov, Z., e Hodzic, E. (2003). A problem of program execution time measurement. *Novi Sad Journal of Mathematics*, 33(1):67–73.

[Halang et al., 2000] Halang, W. A., Gumzej, R., Colnaric, M., e Druzovec3, M. (2000). Measuring the performance of real-time systems. *Real-Time Systems*, 18:59–68.

[Hardison, 2006] Hardison, D. (2006). Open source linux operating system for onboard spacecraft use. On-Line. Disponível em: http://flightlinux.gsfc.nasa.gov/, Consultado em: Maio/2006.

[Haridasan, 2003] Haridasan, M.; Pfitscher, G. (2003). Use of the parallel port to measure mpi inter-task communication costs in cots pc clusters. *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, p. 6 pp.–.

[Hatch, 2006] Hatch, J. (2006). Windows ce real-time performance architecture. In *Windows Hardware Engineering Conference*.

[Hatton, 2007] Hatton, L. (2007). Language subsetting in an industrial context: A comparison of misra c 1998 and misra c 2004. *Inf. Softw. Technol.*, 49(5):475–482.

[Henzinger et al., 2007] Henzinger, T. A., Henzinger, T. A., e Sifakis, J. (2007). The discipline of embedded systems design. *Computer*, 40(10):32–40.

[Heursch et al., 2001] Heursch, A. C., Horstkotte, A., e Rzehak, H. (2001). Preemption concepts, rhealstone benchmark and scheduler analysis of linux 2.4. In *Real-Time & Embedded Computing Conference*.

[Holden, 2004] Holden, J. (2004). Notes on pc timing. On Line. Disponível em: http://www.psych.usyd.edu.au/staff/johnh/timing_notes.html. Consultado em Julho/2008.

[Hyde, 2003] Hyde, R. (2003). *The Art of Assembly Language*. No Starch Press.

[IEEE, 1990] IEEE (1990). Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, p. –.

[Ingram, 1999] Ingram, D. (1999). Soft real time scheduling for general purpose client-server systems. In *Proceedings of the Seventh Workshop on Hot Topics in Operating Systems*, number 7, p. 130–135. Cambridge University. ISBN: 0-7695-0237-7.

[Intel, 2002] Intel (2002). *Intel Low Pin Count (LPC) - Interface Specification. Disponível em http://www.intel.com/design/chipsets/industry/lpc.htm. Consultado em 13/01/2008.* Intel Corporation, 1.1 edition.

[Intel, 2004] Intel (2004). *IA-PC HPET (High Precision Event Timers) Specification. Disponível em: http://www.intel.com/technology/architecture/hpetspec.htm. Consultado em: 13/01/2008.*

[Intel, 2007] Intel (2007). *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1.* Intel Corporation.

[Ip, 2001] Ip, B. (2001). Performance analysis of vxworks and rtlinux. Relatório técnico, Columbia University, Department of Computer Science, New York. http://www.mecatronica.eesc.usp.br/ aroca/restrito/artigos/vxworkslinuxrt.pdf http://www1.cs.columbia.edu/ sedwards/classes/2001/w4995-02/reports/ip.pdf.

[Irwin et al., 2002] Irwin, P., Richard, L., e Johnson, J. (2002). Real-time control using open source rtos. In Lewis, H., editor, *Advanced Telescope and Instrumentation Control Software II*, volume 4848, p. 560–567. SPIE.

[Jacker, 2002] Jacker, K. H. (2002). Teaching real-time control using free systems software. In *Proceedings of the 4th Real-time Linux Workshop.*

[Jensen, 1994] Jensen, D. (1994). Adventures in embedded development. *IEEE Software*, 11(6):116–117.

[Jespersen, 1999] Jespersen, James. Fitz-Randolph, J. (1999). *From Sundials To Atomic Clocks - Understanding Time and Frequency.* Dover Publications.

[Jones, 1997] Jones, M. (1997). What really happened on mars? On Line. Disponível em: http://research.microsoft.com/ mbj/Mars_Pathfinder/Mars_Pathfinder.html. Consultado em Agosto/2008.

[jong Kim et al., 2006] jong Kim, W., Ji, K., e Ambike, A. (2006). Real-time operating environmentfor networked control systems. *Automation Science and Engineering, IEEE Transactions on [see also Robotics and Automation, IEEE Transactions on]*, 3(3):287–296.

[Junior et al., 2008] Junior, J. M., Junior, L. C., e Caurin, G. A. P. (2008). Scara3d: 3-dimensional hri integrated to a distributed control architecture for remote and cooperative actuation. In *SAC '08: Proceedings of the 2008 ACM symposium on Applied computing*, p. 1597–1601, New York, NY, USA. ACM.

[Kailas e Agrawala, 1997] Kailas, K. K. e Agrawala, A. K. (1997). An accurate time-management unit for real-time processors. Relatório Técnico CS-TR-3768, University of Maryland Institute for Advanced Computer Studies.

[Kailas et al., 1997] Kailas, K. K., Trinh, B., e Agrawala, A. K. (1997). Temporal accuracy and modern high performance processors: A case study using pentium pro. Relatório Técnico CS-TR-3820, University of Maryland Institute for Advanced Computer Studies.

[Kalynnda, 2002] Kalynnda, B. (2002). Real-time linux evaluation. Relatório técnico, Glen Research center, NASA.

[Katuin, 2003] Katuin, J. (12-16 May 2003). Proton therapy treatment room controls using a linux control system. *Particle Accelerator Conference, 2003. PAC 2003. Proceedings of the*, 2:1068–1070 Vol.2.

[Kleines e Zwoll, 1996] Kleines, H. e Zwoll, K. (Feb 1996). Real time unix in embedded control-a case study within the context of lynxos. *Nuclear Science, IEEE Transactions on*, 43(1):13–.

[Klingsheim et al., 2007] Klingsheim, A. N., Klingsheim, A. N., Moen, V., e Hole, K. J. (2007). Challenges in securing networked j2me applications. *Computer*, 40(2):24–30.

[Knight, 2007] Knight, J. (2007). The glass cockpit. *Computer*, 40(10):92–95.

[Kohout et al., 2003] Kohout, P., Ganesh, B., e Jacob, B. (2003). Hardware support for real-time operating systems. In *Hardware/Software Codesign and System Synthesis, 2003. First IEEE/ACM/IFIP International Conference on*, p. 45–51.

[Kornecki et al., 1998] Kornecki, A., Wojcicki, H., Zalewski, J., e Kruszynska, N. (1998). Teaching device drivers technology in a real-time systems curriculum. *rtew*, 00:42.

[Kornecki et al., 2000] Kornecki, A., Zalewski, J., e Eyassu, D. (6-8 March 2000). Learning real-time programming concepts through vxworks lab experiments. *Software Engineering Education & Training, 2000. Proceedings. 13th Conference on*, p. 294–301.

[Krodel e Romanski, 2007] Krodel, J. e Romanski, G. (2007). Real-time operating systems and component integration considerations in integrated modular avionics systems report. Relatório técnico, U.S. Department of Transportation - Federal Aviation Administration.

[Kurki-Suonio, 1994] Kurki-Suonio, R. (Jun 1994). Real time: further misconceptions (or half-truths) [real-time systems]. *Computer*, 27(6):71–76.

[Köker, 2007] Köker, K. (2007). *Autonomous Robots and Agents*, chapter Embedded RTOS: Performance Analysis With High Precision Counters, p. 171–179. Springer Berlin / Heidelberg.

[L. Dozio, 2003] L. Dozio, P. M. (2003). Linux real time application interface (rtai) in low cost high performance motion control. In *Motion Control 2003, a conference of ANIPLA - Associazione Nazionale Italiana per l'Automazione*.

[Labrosse, 2002] Labrosse, J. (2002). *MicroC/OS-II - The Real Time Kernel*. CMP Books, 2 edition.

[Lamie e Carbone, 2007] Lamie, W. e Carbone, J. (2007). Measure your rtos's real-time performance. *Embedded Systems Design*.

[Laplante, 2004] Laplante, P. A. (2004). *Real-Time System Design and Analysis*. John Wiley & Sons.

[Laplante, 2005] Laplante, P. A. (2005). Criteria and an objective approach to selecting commercial real-time operating systems based on published information. *International journal of computers & applications*, 27:82–96.

[Laurich, 2004] Laurich, P. (2004). A comparison of hard real-time linux alternatives. on-line. disponível em: http://linuxdevices.com/articles/at3479098230.html. *LinuxDevices.com*.

[Lennon, 2001] Lennon, A. (2001). Embedding linux. *IEE Review*, 47(3):33 – 37.

[Leroux e Schaffer, 2006] Leroux, P. e Schaffer, J. (2006). Exactly when do you need real time? *Embedded.com*.

[Leveson et al., 1993] Leveson, N., Leveson, N., e Turner, C. (1993). An investigation of the therac-25 accidents. *Computer*, 26(7):18–41.

[Levi e Agrawala, 1990] Levi, S.-T. e Agrawala, A. K. (1990). *Real-Time System Design*. McGraw-Hill Publishing Company.

[Li et al., 1997] Li, Y., Potkonjak, M., e Wolf, W. (12-15 Oct 1997). Real-time operating systems for embedded computing. *Computer Design: VLSI in Computers and Processors, 1997. ICCD '97. Proceedings., 1997 IEEE International Conference on*, p. 388–392.

[Liu e Layland, 1973] Liu, C. L. e Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61.

[Lu et al., 2003] Lu, B., McKay, W., Lentijo, S., Monti, A., Wu, X., e Dougal, R. (2003). The real time extension of the virtual test bed. In *Proceedings of the Huntsville Simulation Conference*.

[LynxWorks, 2006] LynxWorks (2006). Embedded linux: Bluecat linux - robust embedded-linux operating system based on linux 2.6 kernel. On-Line. Disponível em http://www.lynuxworks.com/embedded-linux/embedded-linux.php. Acesso em: Maio/2006.

[Martin, 1967] Martin, J. (1967). *Design of Real-Time Computer Systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

[Martin Timmerman, 2000a] Martin Timmerman, B. B. (2000a). Executive summary of the evaluation report on windows nt 4.0 workstation - executive summary. *Dedicated Systems Magazine*.

[Martin Timmerman, 2000b] Martin Timmerman, B. B. (2000b). Hyperkernel 4.3 evaluation - executive summary. *Dedicated Systems Magazine*.

[Martin Timmerman, 2000c] Martin Timmerman, B. B. (2000c). Intime 1.20 evaluation - executive summary. *Dedicated Systems Magazine*.

[Martin Timmerman, 2000d] Martin Timmerman, B. B. (2000d). Qnx 4.25 evaluation - executive summary. *Dedicated Systems Magazine*.

[Martin Timmerman, 2000e] Martin Timmerman, B. B. (2000e). Rtx 4.2 evaluation - executive summary. *Dedicated Systems Magazine*.

[Martin Timmerman, 2000f] Martin Timmerman, B. B. (2000f). Vxworks/x86 5.3.1 evaluation- executive summary. *Dedicated Systems Magazine*.

[Martin Timmerman, 1998] Martin Timmerman, Bart Beneden, L. H. (1998). Windows nt real-time extensions - better or worse? *Real-Tima Magazine*, 3.

[McConnell, 2004] McConnell, S. (2004). *Code Complete*. Microsoft Press, Redmond, WA, USA, 2 edition.

[McKenney, 2007] McKenney, P. (2007). Smp and embedded real time. *Linux Journal*. http://www.linuxjournal.com/article/9361 http://www.linuxjournal.com/node/9361/print.

[McKenney, 2008] McKenney, P. E. (2008). Responsive systems: An introduction. *IBM Systems Journal*.

[Microsoft, 2007a] Microsoft (2007a). Real time and windows ce. On-Line. Disponível em: http://msdn2.microsoft.com/en-us/embedded/aa714541.aspx. Consultado em Julho/2007.

[Microsoft, 2007b] Microsoft (2007b). Windows xp embedded overview. On Line. Disponível em: http://www.microsoft.com/windows/embedded/products/wexpe/default.mspx. Consultado em Agosto/2008.

[MISRA-C-2004, 2004] MISRA-C-2004 (2004). *MISRA-C:2004 Guidelines for the use of the C language in critical systems*. MISRA - The Motor Industry Software Reliability Association.

[Montavista, 2008] Montavista (2008). Real-time linux. on-line: http://www.mvista.com/, consultado em: 03/fev/2008.

[NASA, 2004] NASA (Feb.-March 2004). Vxworks set for another stay on mars. *Electronics Systems and Software*, 2(1):44–44.

[Nass, 2007] Nass, R. (2007). Annual study uncovers the embedded market. *Embedded Systems Design*.

[National Instruments, 2007a] National Instruments (2007a). Quando preciso de um sistema real-time?. consultado em 05/jan/2008. Relatório técnico, National Instruments. Document Type: Tutorial NI Supported: Yes Publish Date: 29/Mai/2007.

[National Instruments, 2007b] National Instruments (2007b). Tutorial de real-time. Relatório técnico, National Instruments. Consultado em 06/Jan/2008. Document Type: Tutorial NI Supported: Yes Publish Date: 31/Mai/2007.

[NIST, 2002] NIST (2002). Introduction to linux for real-time control. Relatório técnico, National Institude of Standards and Technology (NIST).

[Obeland, 2001] Obeland, K. M. (2001). Posix in real-time. Relatório técnico, University of Southern California.

[Obenland, 2001] Obenland, K. (2001). Real-time performance of standards based commercial operating systems. In *Embedded Systems Conference*.

[Oberle e Walter, 2001] Oberle, V. e Walter, U. (2001). Micro-second precision timer support for the linux kernel. Relatório técnico, IBM Linux Challenge. Winner of an IBM Linux Challenge Award.

[O'Dowd, 2008] O'Dowd, D. (2008). Embedded linux: With friends like these, who needs enemies? *Embedded.com*.

[Ostroff, 1992] Ostroff, J. S. (1992). Formal methods for the specification and design of real-time safety critical systems. *Journal of Systems and Software*, 18:36–60.

[Parab et al., 2007] Parab, J. S., Shelake, V. G., Kamat, R. K., e Naik, G. M. (2007). *Exploring C for microcontrollers - A Hands On Approach*. Spring.

[Piccioni et al., 2001] Piccioni, C. A., Tatibana, C. Y., e de Oliveira, R. S. (2001). Trabalhando com o tempo real em aplicações sobre o linux. Relatório técnico, UFSC - Universidade Federal de Santa Catarina, CTC - Centro Tecnológico, DAS - Departamento de Automação e Sistemas.

[Pieper et al., 2007] Pieper, S. M., Pieper, S. M., Paul, J. M., e Schulte, M. J. (2007). A new era of performance evaluation. *Computer*, 40(9):23–30.

[Proctor, 2006] Proctor, F. (2006). Real-time linux. On-Line. Disponível em http://www.isd.mel.nist.gov/projects/rtlinux/. Acesso em: Maio/2006.

[Proctor, 2001] Proctor, F. M. (2001). Measuring performance in real-time linux. In *Third Real-Time Linux Workshop*.

[Proctor e Shackleford, 2001] Proctor, F. M. e Shackleford, W. P. (2001). Real-time operating system timing jitter and its impact on motor control. In *Proceedings of the SPIE Conference on Sensors and Controls for Intelligent Manufacturing II*.

[Proctor, 2002] Proctor, M. (2002). Embedded real- time linux for cable robot control. In *Proceedings of the ASME Design Engineering Technical Conference*, volume 1, p. 851 – 856.

[Puthiyedath et al., 2002] Puthiyedath, L. K., Cota-Robles, E., Keys, J., Aggarwal, A., e Held, J. P. (2002). The design and implementation of the intel® real-time performance analyzer. In *Proceeding of the Eighth Real-Time and Embedded Technology and Applications Symposium*. Intel.

[RadiSys, 1998] RadiSys (1998). Intime interrupt latency report - measured interrupt response times - technical paper. Relatório técnico, RadiSys.

[Reis, 1992] Reis, M. C. (1992). *Micros XT-AT Circuitos*. Letron Livros.

[Relf, 2007] Relf, C. G. (2007). Deterministic data acquisition and control with real-time. On-Line: , Consultado em Julho/2007.

[Romanski, 2002] Romanski, G. (2002). Certification of an operating system as a reusable component. In *Digital Avionics Systems Conference*, volume 1, p. 5D3–1 – 5D3–9.

[Rosen, 2007] Rosen, L. (2007). The new qnx hybrid software model: Combining open source and proprietary benefits for embedded systems. on-line. disponível em: http://www.qnx.com/download/feature.html?programid=16868. Relatório técnico, QNX Software Systems.

[Rosenquist, 2003] Rosenquist, C. (2003). Hard realtime rapid prototyping development platform. Dissertação de Mestrado, Linköpings universitet.

[Ross, 2008] Ross, P. (2008). Why cpu frequency stalled. *IEEE Spectrum*, 45(4):72–72.

[Rubini e Corbet, 2001] Rubini, A. e Corbet, J. (2001). *Linux Device Drivers*. OReilly, 2 edition.

[Sacha, 1995] Sacha, K. (1995). Measuring the real-time operating system performance. In *Euromicro Workshop on Real-Time Systems*, volume 7, p. 34–40.

[Sandborn, 2008] Sandborn, P. (2008). Trapped on technology's trailing edge. *IEEE Spectrum*, 45(4):42–58.

[Sangiovanni-Vincentelli et al., 2007] Sangiovanni-Vincentelli, A., Sangiovanni-Vincentelli, A., e Di Natale, M. (2007). Embedded system design for automotive applications. *Computer*, 40(10):42–51.

[Santhanam, 2003] Santhanam, A. (2003). Towards linux 2.6 - a look into the workings of the next new kernel. on-line. disponível em: http://www.ibm.com/developerworks/linux/library/l-inside.html. consultado em: 20/01/2008. Relatório técnico, IBM.

[Sarolahti, 2001] Sarolahti, P. (2001). Real-time application interface. Relatório técnico, University of Helsinki.

[Sato e Asari, 2006] Sato, K. e Asari, K. (2006). Characteristics of time synchronization response of ntp clients on ms windows os and linux os. In *38th Annual Precise Time and Time Interval (PTTI) Meeting*.

[Schmidt, 2007] Schmidt, K. U. (2007). Ieee 1588 on windows xp powered measurement devices - mastering the trigger challenge. In *Proc. IEEE International Symposium on Precision Clock Synchronization for Measurement, Control and Communication ISPCS 2007*, p. 92–95.

[Schneider, 2002] Schneider, J. (2002). Why you can't analyze rtoss without considering applications and vice versa. In *In Proc. of the 2nd International Workshop on Worst-Case Execution Time Analysis (WCET 2002)*.

[Schumacher et al., 2004] Schumacher, G., Heathcote, S., e Krabbendam, V. (2004). Soar tcs: from implementation to operation. In Lewis, H. e Raffi, G., editors, *Proceedings of SPIE Advanced Software, Control, and Communication Systems for Astronomy*, volume 5496, p. 32–37. SPIE.

[Sebek, 2001] Sebek, F. (2001). Measuring cache related pre-emption delay on a multiprocessor real-time system. In *Proceedings of IEEE Workshop on Real-Time Embedded Systems (RTES'01)*, London.

[Selig, 2006] Selig, M. A. (2006). Em ponto. *Linux Magazine*, p. 68–71.

[Shaw, 2003] Shaw, A. (2003). *Sistemas e Software de Tempo Real*. Bookman.

[Sohal, 2001] Sohal, V. (2001). How to really measure real-time. In *Embedded System Conference*.

[Sorton e Kornecki, 1998] Sorton, E. e Kornecki, A. (Apr/May 1998). Hands-on software design [real-time programming]. *Potentials, IEEE*, 17(2):42–44.

[Spector e Gifford, 1984] Spector, A. e Gifford, D. (1984). The space shuttle primary computer system. *Commun. ACM*, 27(9):872–900.

[Stallings, 2003] Stallings, W. (2003). *Arquitetura e Organização de Computadores*. Prentice Hall.

[Stankovic, 1988] Stankovic, J. A. (1988). Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19.

[Stankovic e Ramamritham, 1990] Stankovic, J. A. e Ramamritham, K. (1990). What is predictability for real-time systems? *Real-Time Syst.*, 2(4):247–254.

[Stankovic et al., 1995] Stankovic, J. A., Spuri, M., Natale, M. D., e Buttazzo, G. C. (1995). Implications of classical scheduling results for real-time systems. *IEEE Computer*, 28(6):16–25.

[Statskontoret, 2004] Statskontoret (2004). Free and open source software. Relatório técnico, Statskontoret.

[Stewart, 2001] Stewart, D. (2001). Measuring execution time and real-time performance. In *Embedded System Conference*.

[Stiennon, 2008] Stiennon, R. (2008). Top ten worst uses for windows. On Line. Disponível em: http://www.networkworld.com/community/node/29644?ts. Consultado em Agosto/2008.

[Stodolsky et al., 1993] Stodolsky, D., Chen, J. B., e Bershad, B. N. (1993). Fast interrupt priority management in operating system kernels. In *In Proceedings of the Second Usenix Workshop on Microkernels and Other Kernel Architectures*, number CS-93-152, p. 105–110.

[Stumpf, 2003] Stumpf, M. (2003). Wind river and nasa - embedded development for the extreme demands of space exploration. *Dedicated Systems Magazine*.

[Sun MicroSystems, 2007] Sun MicroSystems (2007). Binary code license agreement. On Line. Disponível em: http://www.java.com/en/download/license.jsp.

[Sutter, 2002] Sutter, E. (2002). *Embedded Systems Firmware Demystified*. CMP Books.

[Sweet, 1995] Sweet, W. (1995). The glass cockpit [flight deck automation]. *IEEE Spectrum*, 32(9):30–38.

[Systems, 2007] Systems, W. R. (2007). Wind river acquires hard real-time linux technology from fsmlabs - frequently asked questions. On-Line: , Consultado em Julho/2007.

[Tacke e Ricci, 2002] Tacke, C. e Ricci, L. (2002). Real-time determinism in windows ce. *Intel Developer Network*, 4:63–68. http://www.windowsfordevices.com/articles/AT6761039286.html.

[Tanenbaum, 2001] Tanenbaum, A. (2001). *Modern Operating Systems*. Prentice Hall.

[Taurion, 2005] Taurion, C. (2005). *Software Embarcado - A nova onda da informação*. Brasport.

[Tennenhouse, 2000] Tennenhouse, D. (2000). Proactive computing. *Commun. ACM*, 43(5):43–50.

[Theberge et al., 2006] Theberge, M., Theberge, M., e Dudek, G. (2006). Gone swimmin' [seagoing robots]. *IEEE Spectrum*, 43(6):38–43.

[Timmerman, 2000a] Timmerman, M. (2000a). Rtos market overview - a follow up. *Dedicated Systems Magazine*.

[Timmerman, 2000b] Timmerman, M. (2000b). Rtos market survey - preliminary results. *Dedicated Systems Magazine*.

[Timmerman, 2001] Timmerman, M. (2001). What makes a good rtos. Relatório técnico, Dedicated Systems.

[Timmerman et al., 1998] Timmerman, M., Beneden, B. V., e Uhres, L. (1998). Rtos evaluations kick off! *Real-Time Magazine*, 98-3:6–10.

[Timmerman e Perneel, 2001] Timmerman, M. e Perneel, L. (2001). Rtos state of the art. Relatório técnico, Dedicated Systems.

[Tron, 1998] Tron (1998). Introduction to the itron project - open real-time operating system standards for embedded systems. on-line. disponível em: http://www.ertl.jp/itron/survey00/graph-e.pdf. consultado em: Maio/2008. Relatório técnico, TRON Association.

[Tsoukarellas et al., 1995] Tsoukarellas, M., Gerogiannis, V., e Economides, K. (1995). Systematically testing a real-time operating system. *Micro, IEEE*, 15(5):50–60.

[van Genuchten, 2007] van Genuchten, M. (2007). The impact of software growth on the electronics industry. *Computer*, 40(1):106–108.

[Venkateswaran, 2005] Venkateswaran, S. (2005). The passage of time. *Linux Magazine*.

[Vera et al., 2003] Vera, X., Lisper, B., e Xue, J. (2003). Data caches in multitasking hard real-time systems. In *IEEE Real-Time Systems Symposium*.

[Versweyveld, 1999] Versweyveld, L. (1999). Zeus robot system reverses sterilization to enable birth of baby boy. On Line. Disponível em: http://www.hoise.com/vmw/99/articles/vmw/LV-VM-11-99-1.html. Consultado em Agosto/2008.

[Vetromille et al., 2006] Vetromille, M., Ost, L., Marcon, C., Reif, C., e Hessel, F. (2006). Rtos scheduler implementation in hardware and software for real time applications. In *Rapid System Prototyping, 2006. Seventeenth IEEE International Workshop on*, p. 163–168.

[Viswanathan, 2006] Viswanathan, S. (2006). Understanding the windows ce variable tick timer. Relatório técnico, Microsoft Corp.

[VMWare White Paper, 2005] VMWare White Paper (2005). Timekeeping in vmware virtual machines. Relatório técnico, VMWare Inc., 3145 Porter Drive Palo Alto, CA.

[Wang e Lin, 1998] Wang, Y.-C. e Lin, K.-J. (1998). Enhancing the real-time capability of the linux kernel. In *Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on*, p. 11–20.

[Weinberg, 2001] Weinberg, B. (2001). Embedded linux - ready for real time. *Third Real-Time Linux Workshop*, (3).

[Weiss et al., 1999] Weiss, K., Steckstor, T., e Rosenstiel, W. (1999). Performance analysis of a rtos by emulation of an embedded system. In *Rapid System Prototyping, 1999. IEEE International Workshop on*, p. 146–151.

[Williston, 2008] Williston, B. K. (2008). What's the deal with embedded linux? *DSP DesignLine*.

[Wind River, 2006] Wind River (2006). Wind river linux center. on-line. disponível em http://www.windriver.com/linux/, acesso em: Maio/2006.

[Wind River, 2007] Wind River (2007). Wind river linux powers device innovation. On-Line. Disponível em: http://www.windriver.com/linux/Linux_Cust_Snapshots.pdf. Acesso em Fev/2008.

[Windl et al., 2006] Windl, U., Dalton, D., Martinec, M., e Worley, D. R. (2006). The ntp faq and howto - understanding and using the network time protocol (a first try on a non-technical mini-howto and faq on ntp). Relatório técnico, Hewlett-Packard.

[Windows For Devices, 2006] Windows For Devices (2006). Special report: Windows ce 6 arrives with 100% kernel source. On Line. Disponível em: http://www.windowsfordevices.com/news/NS2632317407.html. Consultado em Agosto/2008.

[Wolf, 2007a] Wolf, W. (2007a). The embedded systems landscape. *Computer*, 40(10):29–31.

[Wolf, 2007b] Wolf, W. (2007b). The good news and the bad news. *Computer*, 40(11):104–105.

[won Lee; Sung-Ho Hwang; Jae Wook Jeon, 2006] won Lee; Sung-Ho Hwang; Jae Wook Jeon, S.-H. S. S. (2006). Analysis of task switching time of ecu embedded system ported to osek(rtos). *SICE-ICASE, 2006. International Joint Conference*, p. 545–549.

[Yaghmour, 2002] Yaghmour, K. (2002). Adaptative domain environment for operating systems. Relatório técnico, Opersys.

[Yearraballi, 2000] Yearraballi, R. (2000). Real-time operating systems: An ongoing review. In *The 21st IEEE Real-Time Systems Symposium*.

[Yodaiken, 1999] Yodaiken, V. (1999). The RTLinux manifesto. In *Proc. of The 5th Linux Expo, Raleigh, NC*.

[Yodaiken e Barabanov, 1997] Yodaiken, V. e Barabanov, M. (1997). A real-time linux. In *Proceedings of the Linux Applications Development and Deployment Conference (USELINUX)*. The USENIX Association. Online at http://rtlinux.cs.nmt.edu/rtlinx/u.pdf.

[Yodaiken, 1997] Yodaiken, V. J. (1997). Adding real-time support to general purpose operating. Registro de Patente Americana Número 5995745.

[Zhang et al., 2006] Zhang, G., Chen, L., e Yao, A. (2006). Study and comparison of the rthal-based and adeos-based rtai real-time solutions for linux. *imsccs*, 2:771–775.

[Zhang e West, 2006] Zhang, Y. e West, R. (2006). Process-aware interrupt scheduling and accounting. *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*, p. 191–201.