Pós-Graduação em Ciência da Computação

ANÁLISE DE SISTEMAS OPERACIONAIS DE TEMPO REAL

Por

Anderson Luiz Souza Moreira

Dissertação de Mestrado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE, MAIO/2007

UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ANDERSON LUIZ SOUZA MOREIRA

ANÁLISE DE SISTEMAS OPERACIONAIS DE TEMPO REAL

ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR: Prof. Sérgio V. Cavalcante, PhD.

RECIFE, MAIO/2007

Em memória de meu tio Luiz José Souza um homem que se importava, sempre esteve do meu lado em todos os momentos de minha vida e agora me protege.

Agradecimentos

Agradeço primeiramente a minha mãe, Dona Maria do Carmo, por me incentivar desde pequeno em buscar o caminho da verdade e do respeito ao próximo.

Ao meu orientador Sérgio Cavalcante, incentivador e guia sempre atento a minha formação e por ter acreditado em minha capacidade mesmo sabendo das limitações que tinha no início do curso.

A Lilian Ramires, minha grande amiga, pelo carinho, apoio, atenção e sempre lutou para que eu não desistisse perante as adversidades e continuasse seguindo em frente.

Minha família, que mesmo longe, sei que poderei contar com eles em todos os momentos da vida.

Também agradeço aos meus novos amigos, principalmente a Fernando, Guilherme, André, Alan, Jenny e Gláucya que me ajudaram em diversos momentos durante o período que estive na Instituição e Augusto Pedroza por me ajudar na codificação do *benchmark* utilizado nesse trabalho e nas idéias de como realizar os testes.

“Antes de os relógios existirem, todos tinham tempo. Hoje, todos têm relógios”.

Eno Wanke, poeta.

“Se não fosse para cometer erros, eu nem tentaria fazer”.

Robert Wood Johnson, empresário.

Sumário

Listas de Acrônimos

API – *Application Program Interface*

CPU – *Central Processing Unit*

CSMA/CD – *Carrier Sense Multiple Access/Collision Detection*

CTA – Controle de Tráfego Aéreo

DM – *Deadline Monotonic*

DVS – *Dynamic Voltage Scale*

EDF – *Earliest Deadline First*

EDRES – *Earliest Deadline Relative*

EEMBC – *Embedded and Microprocessor Benchmark Consortium*

FIFO – *First-In First-Out*

GB – *Giga Byte*

GCC – *Gnu C Compiler*

HAL – *Hardware Abstraction Layer*

IDE – *Integrated Development Environment*

IEEE – *Institute of Electrical and Electronic Engineers*

IPC – *Interprocess Communication*

ISR – *Interrupt Service Routine*

KB – *Kilo Byte*

LTT – *Linux Trace Toolkit*

LX/RT – *Linux Real-Time*

Mb – *Mega bit*

MB – *Mega Byte*

MMU – *Memory Manager Unit*

NASA – *National Aeronautics and Space Administration*

OCERA – *Open Components for Embedded Real-time Applications*

OEM – *Original Equipment Manufacturer*

POSIX – *Portable Operating Systems Interface*

PTR – Processo de Tempo Real

QoS – Qualidade de Serviço

RAM – *Random Access Memory*

RM – *Rate Monotonic*

ROM – *Read Only Memory*

RTAI – *Real-time Application Interface*

SMP – *Symmetric Multiple Processor*

SOPG – Sistemas Operacionais de Propósito Geral

SOTR – Sistemas Operacionais de Tempo Real

STR – Sistemas de Tempo Real

TLSF – *Two Level Segregated Fit*

WCET – *Worst Case Execution Time*

Índice de Figuras

Índice de Tabelas

Resumo

Sistemas críticos de tempo real estão se tornando cada vez mais utilizados. A grande complexidade na criação de aplicações para tempo real favorece o surgimento de novos recursos que facilitam ao projetista adequar tais sistemas às suas necessidades, entre eles os Sistemas Operacionais de Tempo Real (SOTR).

Conseqüência disso é que a adoção dos SOTR têm aumentado significativamente, principalmente entre os que são baseados em Sistemas Operacionais de Propósito Geral (SOPG), mais precisamente os que são baseados em *Linux*. Existem também aqueles sistemas que foram desenvolvidos especificamente para o tratamento de tarefas temporais, sendo um dos mais famosos o *VxWorks* da *Wind River* como também o *Windows* CE da *Microsoft*.

Contudo o SOTR é uma peça integrante em um sistema de tempo real. O presente trabalho tem como ideal facilitar o projetista de tempo real a escolher entre os diversos tipos de SOTR, entre eles o sistema proprietário da *Microsoft*, *Windows* CE e os sistemas de código aberto *RTLinux* e RTAI.

Elementos chaves, como previsibilidade, tamanho, modularidade e adaptabilidade são verificados para ter o conhecimento necessário que auxilie o projetista no desenvolvimento de sistemas mais confiáveis.

O resultado final será a verificação da real capacidade e maturidade de tais sistemas, auxiliando o projetista de tempo real na criação de aplicações. Também será analisado se os mesmos podem ter seus escalonadores modificados.

Abstract

Critical Real-Time Systems are becoming more used each day. The great complexity in creating real-time applications favors the sprouting of new resources that facilitate the designer to adapt such systems to his/her needs. Among them there are the Real-Time Operating Systems (RTOS).

A consequence of this is that RTOS adoption have significantly increased, mainly among those that are based on General Purpose Operating Systems (GPOS), more specifically the ones that are based on Linux. There are also systems that have been developed specifically for the treatment of temporal tasks: one of the most famous are the VxWorks by Wind River and Windows CE from Microsoft.

However the RTOS is an integral part in a real-time system. The present work tries to facilitate the designer of real-time system to choose from some well known RTOS, among them the one developed by Microsoft, Windows CE and the source system open RTLinux and RTAI. Key elements, as predictability, size, modularity and adaptability are verified and compared in order to assist the designer in the development of more reliable systems.

The final result will be the verification of the real capability and maturity of such RTOS, assisting in the creation of real-time systems. Also the scheduler of each system is analyzed concerning its modification flexibility.

Capítulo 1 - Introdução

Este capítulo apresenta o contexto no qual esta dissertação encontra-se inserida, apresentando as motivações para a sua realização, seus principais objetivos, trabalhos relacionados, assim como sua estruturação em capítulos.

1.1. Apresentação

Analisar e verificar diferentes soluções de sistemas operacionais são importantes áreas da Ciência Computacional e mostram-se como importantes campos exploratórios tanto comerciais quanto acadêmicos. Mensurações ou execução de *benchmarks*, em aplicações de tempo real, não são tarefas triviais e muitas vezes conduzem a análises errôneas. Ainda assim é um caminho capaz de prover uma vasta gama de informações aos desenvolvedores de tais sistemas. Por exemplo, qual a melhor solução em SOTR para uma determinada aplicação crítica e se esta é a ideal para uma determinada arquitetura de *hardware* e *software*.

Muitas controvérsias sobre o uso de sistemas operacionais como componentes importantes de sistemas de tempo real vem sendo travadas, principalmente no meio acadêmico. O número de SOTR existente é muito grande e cada um com uma característica que mais lhe favorece. Porém, nessa competição, algumas das funcionalidades e serviços dos SOTR requerem um comportamento previsível de um ou mais eventos que os envolvam.

Essa é a principal característica e razão porque muitos dos sistemas operacionais de tempo real têm suporte tanto à execução de tarefas críticas como também para tarefas não críticas, auxiliando o projetista de tempo real com uma solução adequada a cada tipo de problema.

O presente trabalho foca no comportamento de *software*, porém o leitor deve ter em mente que o conhecimento necessário em *hardware* também se faz com similar importância, mas que não terá uma pesquisa tão ampla quanto a primeira.

1.2. Objetivos / Motivação

O ambiente de sistemas embarcados, principalmente na área de controle automotivo industrial e de robótica, vem se tornando cada vez mais complexo. Tipicamente isso consiste em muitos computadores operando em múltiplos níveis de controle ou supervisionando diferentes dispositivos eletrônicos, cada um com uma função diferente. A partir de uma determinada plataforma de *hardware*, a execução eficiente de uma aplicação de tempo real requer do desenvolvedor conhecimento necessário para um gerenciamento eficiente dos códigos fontes, trabalho com tarefas, escalonamento, carga do sistema e programação concorrente. No entanto um SOTR

deve prover ao desenvolvedor primitivas que casem com esses conhecimentos citados além de comunicação entre tarefas e operações com dispositivos, entre outros.

A escolha por usar um determinado SOTR em que as características devam oferecer suporte a um domínio específico de sistemas de tempo real é muitas vezes feita de forma não condizente ao que realmente o desenvolvedor quer, ou por motivos de indicações de terceiros ou por reputação no mercado.

Por outro lado os SOTR estão cada vez mais completos e com uma série de recursos que muitas vezes não são utilizados pelo projetista. Assim, também são importantes fatores como flexibilidade para alterações, confiança no desenvolvedor, garantia de funcionamento, facilidade de suporte, entre outros.

O presente trabalho tem como ideal facilitar o projetista de tempo real a escolher entre os diversos tipos de SOTR, entre eles o sistema proprietário da *Microsoft*, *Windows* CE, e os sistemas de código aberto *RTLinux* e RTAI, sendo que estes dois últimos têm tido uma importância muito grande nos últimos tempos já que estes fazem uso do *Linux* como sistema hospedeiro e o mesmo carrega consigo grandes facilidades de desenvolvimento. Elementos chave como previsibilidade, tamanho, modularidade e adaptabilidade são verificados para ter conhecimento necessário que auxilie o projetista no desenvolvimento de sistemas mais confiáveis.

O resultado final será a verificação da real capacidade e maturidade de tais sistemas, além de um conjunto de regras para a escolha de um SOTR para o projeto de sistemas de tempo real, auxiliando na criação de tais sistemas, com uso considerável de recursos de cada sistema operacional em questão. Também será analisado se os mesmos podem ter seus escalonadores modificados.

1.3.Trabalhos relacionados

O trabalho escrito por Ghosh *et al* [22] discute se é melhor construir um SOTR ou comprar um já pronto. Um modelo sistemático de escolha é definido para a seleção de um SOTR. Esses critérios são: performance, componentes de *software* que compõe o pacote do SOTR, dispositivos suportados pelo sistema operacional, suporte técnico, licença e reputação no mercado.

O *benchmark Hartstone*, e o que ele tenta mensurar é brevemente descrito no artigo de Weiderman e Kamenoff [65]. As medidas e métricas do *benchmark* são o tempo de execução de seis diferentes operações. Essas operações são cruciais no

trabalho, pois servem para medir performance de um SOTR no que diz respeito ao tempo de troca de contexto entre as tarefas. Além também de avaliar tempo de preempção, tempo de interrupção, tempo entre troca de mensagens e latência no uso de semáforos.

O artigo de Stankonvic e Rajkumar [60], descreve uma série de características que podem ser encontradas em SOTR. Essas características são agrupadas seguindo diferentes classificações. Também são descritas técnicas de escalonamento que podem ser utilizadas em sistemas de tempo real. No entanto o trabalho deles não fala de um SOTR específico e nem de características que possam ajudar a escolher um determinado sistema.

Já no trabalho de Weinberg e Cesati [66] encontramos a descrição de uma aproximação para medir e analisar o tempo de execução de um *kernel* de tempo real em uma plataforma de *hardware* específica. Além de descrever os passos necessários para mudar de uma solução proprietária de SOTR para *Linux* com funções de tempo real. Porém nenhuma análise é descrita para mostrar a importância de fazer tal mudança e o motivo não é bem explicado, levando apenas em consideração a questão do valor e de que o *Linux* é um sistema muito utilizado mundialmente.

1.4. Organização da dissertação

Esta dissertação contém mais cinco capítulos organizados da seguinte forma:

- **Capítulo 2 – Estado da Arte**: mostra estudos atuais de SOTR proprietário e os que são baseados em *Linux*, além de mostrar atuais pesquisas na área de escalonamento de processos;

- **Capítulo 3 – Avaliação comparativa das características dos sistemas operacionais de tempo real**: avalia os SOTR escolhidos para o presente estudo levando em consideração os pontos em destaque de suas características: Tamanho, Modularidade, Adaptabilidade e Previsibilidade além das premissas levadas em consideração para análise do presente trabalho;

- **Capítulo 4 – Avaliação comparativa a partir dos resultados obtidos com o *benchmark*:** mostra a idéia do *benchmark* desenvolvido e das técnicas de avaliação de latência e troca de contexto com definições formais. Também são avaliados os resultados obtidos durante a execução do *benchmark*;

- **Capítulo 5 – Estudo de caso:** Nesse Capítulo foram implementados três exemplos de diferentes situações de aplicações de tempo real, críticas ou não, em que envolvam o uso de SOTR. A representação do funcionamento de tais situações é necessária mostrando como a restrição de tempo imposta e o ambiente são importantes. Mostra também que SOTR estão ganhando espaço não somente em universidades como também em outras áreas como na indústria e em aplicações governamentais;

- **Capítulo 6 – Conclusões**: este capítulo conclui o trabalho com um breve resumo do que foi feito bem como as principais contribuições do trabalho e sugestões para trabalhos futuros, com o objetivo de mostrar novas direções que podem estender este trabalho.

Capítulo 2 - Estado da Arte

O objetivo desse capítulo é fazer um estudo do estado da arte de SOTR proprietários e daqueles baseados em *Linux* e Algoritmos de Escalonamento disponíveis pela comunidade de pesquisa e por empresas particulares, e determinar que tipos de sistemas e mecanismos realmente podem ser os mais úteis para aplicações de tempo real. De outra forma, este capítulo analisará as características de tempo real dos SOTR e de algumas técnicas atuais de escalonamento, extraindo as características principais que serão incluídas no desenvolvimento dessa dissertação. O projeto é integrar protótipos de várias técnicas inovadoras de tempo real.

2.1. Sistemas de Tempo Real

Sistemas de Tempo Real (STR) são, em geral, reativos ao ambiente que estão inseridos, o que faz com que a prova dos requisitos críticos (lógica e temporal) sejam extremamente complexos [28]. Atualmente encontramos STR em uma gama bem diversificada de aplicações, desde eletrodomésticos e videogames até complexos sistemas de controle de tráfego aéreo (CTA).

2.1.1. Descrição

De acordo com Farines *et al* [17] o problema de tempo real é fazer com que as aplicações tenham um comportamento previsível, atendendo as características temporais impostas pelo ambiente ou pelo usuário, mesmo com recursos limitados.

O objetivo principal dos STR não é executar o aplicativo o mais rápido possível e sim executar no tempo estabelecido e momento correto [59]. Em alguns casos o sistema deve esperar até um determinado estímulo para ser executado. Observe o exemplo da Figura 1, que demonstra um sistema de *airbag*. Caso o *airbag* seja inflado tarde demais, o motorista irá se chocar com o volante quando ocorrer à colisão, por outro lado se inflado cedo demais, o sistema de *airbag* irá desinflar antes de proteger o motorista do impacto. O momento ideal para que o sistema de *airbag* proteja o motorista é compreendido entre os tempos *t* 1 e *t* 2.

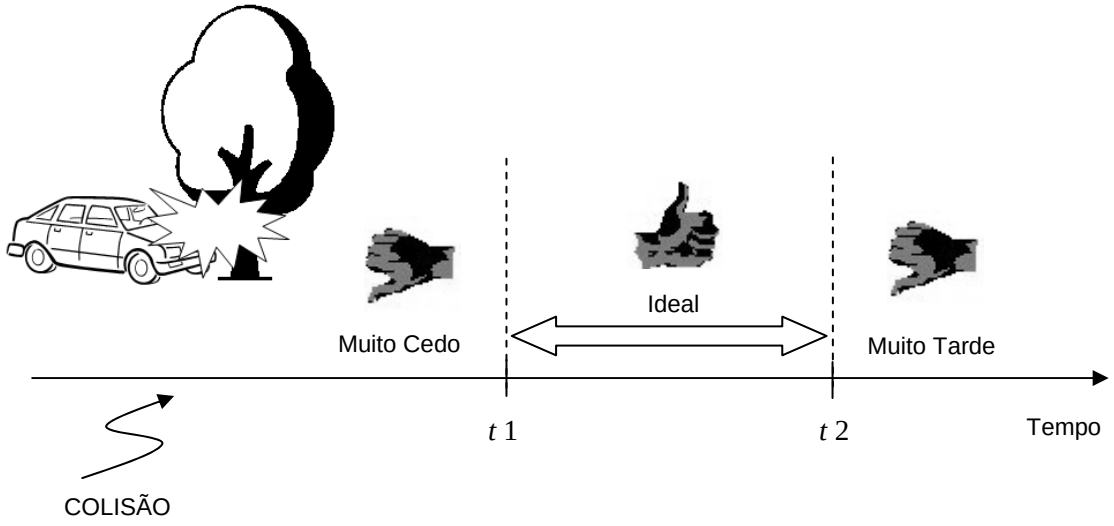


Figura 1 - Exemplo de um STR – *airbag* de um automóvel em uma colisão [56]

Ferramenta indispensável para atender este ponto é a previsibilidade do sistema, ou seja, em se tratando de tempo, todas as tarefas devem ser executadas dentro de tempos limitados e conhecidos. Tradicionalmente, algoritmos de escalonamento baseados em técnicas propostas por Liu e Layland [32] como o *Rate Monotonic* (RM), *Deadline Monotonic* (DM) e o *Earliest Deadline First* (EDF) vêm sendo bastante utilizados no contexto de aplicações para tempo real. Isso é devido a inúmeros trabalhos e pesquisas científicas que permitiram a tais técnicas obter a previsibilidade ainda em tempo de projeto (*off-line*) ou a análise da escalonabilidade *a priori* sem a necessidade da construção do escalonamento propriamente dita.

2.1.2. Características de Sistemas de Tempo Real

Quanto a criticidade os STR se classificam em: **críticos** (*hard real-time*) na quais as aplicações caso não cumpram suas restrições temporais podem ter conseqüências catastróficas; e **não críticos** (*soft real-time*), onde o não cumprimento das restrições temporais pode ser aceitável, desde que dentro de certos limites. Em sistemas críticos podemos ainda subdividir os sistemas em: ***Fail Safe*** em que um sistema identifique uma falha, rapidamente atinge um estado seguro. Um exemplo de tal sistema é o de sinalização ferroviária; e ***Fail Operational***, em que um estado seguro não pode ser identificado, tendo que manter um nível mínimo de serviço para não ocorrer uma catástrofe, exemplo são sistemas de controle de vôo e de CTA.

Uma outra classificação de sistemas não críticos é: **Sistemas com Elevada Disponibilidade** o qual deve ter como garantir que está na maior parte do tempo operacional, como sistemas de comutação telefônica; e **Sistemas com Elevada Integridade**, o qual tem pouca probabilidade de erro por falha de algum componente. Podemos exemplificar tais sistemas como os sistemas bancários que utilizam transações *on-line*.

Existem outras classificações para os STR, como a de Kolano e Demmerer [27] e Kopetz [28]. Sendo que a de Kopetz oferece uma abrangência maior, classificando-os em:

- Resposta Garantida (*Guaranteed-Response*) e Melhor Esforço (*Best-Effort*): os sistemas de Resposta Garantida são calculados e planejados seguindo hipóteses de carga e falha, podendo ser considerados sistemas quase perfeitos enquanto

que os sistemas de Melhor Esforço são aqueles que em sua maioria não requerem um estudo minucioso das falhas do sistema e quanto este sistema pode ter de carga, sendo mais adequados a sistemas não-críticos;

- Recursos-Adequados (*Resource-Adequate*) e Recursos-Inadequados (*Resource-Inadequate*): sistemas com Recursos-Adequados seguem o paradigma semelhante ao de resposta garantida, ou seja, existem recursos suficientes para atender o sistema em caso de falhas ou em caso de alta na sua carga. O inverso pode ser considerado para sistemas que possuem a implementação de Recursos-Inadequados, na qual consideram os sistemas sem recursos suficientes para a aplicação, sendo que devem ser usadas técnicas de adequação de recursos dinâmicas, ou seja, em execução;
- Orientado a Evento (*Event-Triggered*) e Orientado a Tempo (*Time-Triggered*): Para sistemas que são Orientados a Eventos todo seu processamento ocorre quando surge uma mudança brusca no ambiente em que o sistema está inserido modificando seu estado. Podemos dizer que o sistema é inicializado quando ocorre um evento. Porém em sistemas Orientados a Tempo os sistemas são processados de acordo com tempos pré-estabelecidos.

2.2.Algoritmos de Escalonamento

Enquanto em especificação e verificação funcional têm-se interesse maior na integridade do sistema, teoria de escalonamento foca no problema de cumprir com os requisitos de tempo especificados. Para satisfazer os requisitos de tempo de sistemas de tempo real, a figura do escalonador utiliza algoritmos que ajudam no tratamento do comportamento temporal das tarefas.

2.2.1. Conceitos de escalonamento para tempo real

Teoria de escalonamento não se limita apenas ao estudo de sistemas de tempo real ou de sistemas de propósito geral, mas também em sistemas industriais, de transportes ou sistemas de controle de processos. Contudo é importante frisar que estudos realizados na área de escalonamento de sistemas de tempo real têm problemas distintos em relação às outras teorias de escalonamento.

Os problemas de escalonamento de sistemas de tempo real, em sua maioria, são *NP-hard* [70], ou seja, não há algoritmos polinomiais para achar soluções sub-ótimas,

porém se considerarmos alguns recursos que são inerentes a sistemas reais como preempção, o problema de escalonamento de tempo real passa a ser NP-completo. Mais informações podem ser obtidas em [17] e [41].

O termo escalonamento de processos se refere em que ordem de execução processos no estado de pronto vão utilizar recursos do processador. A figura central no escalonamento é o escalonador o qual implementa uma escala de execução de acordo com uma política de escalonamento específica (Figura 2).

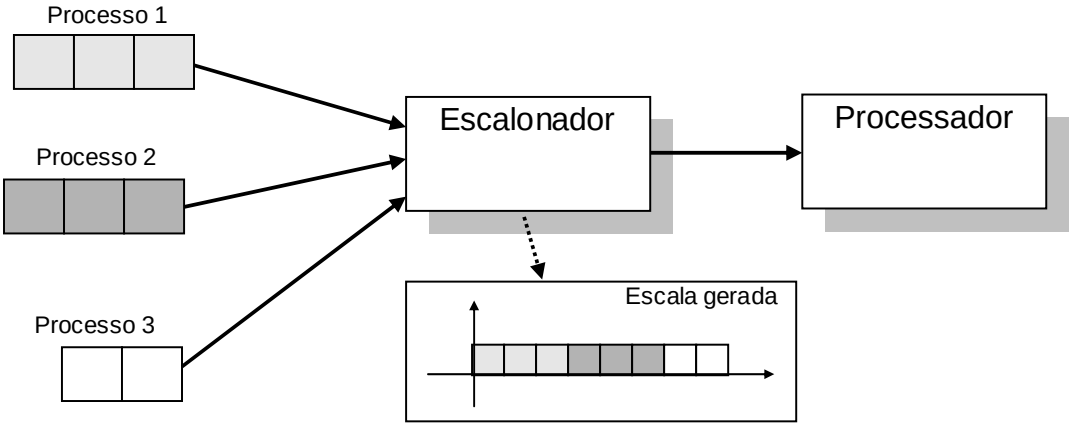


Figura 2 - Escalonamento de tarefas [39]

Algumas propriedades relacionadas ao tempo devem ser consideradas quando trabalhamos com escalonamento de tarefas de tempo real [39]:

- *Release Time* (Tempo de liberação): Tempo em que o processo está pronto para ser processado;
- *Deadline* (Tempo limite): Tempo máximo permitido para que um processo complete sua execução;
- *Worst case execution time* (Tempo de Execução no Pior Caso): Tempo máximo, estabelecido em tempo de projeto, para a completa execução de um processo;
- Período: É o intervalo em que um determinado processo deverá se repetir;
- Prioridade: Urgência relativa de um processo para ser executado.

Essas propriedades obedecem a algumas abordagens de especificações de processos em tempo real, sendo que podem ser: **Periódicos**, em que os processos são ativados regularmente em taxas de período fixas; **Aperiódicas**, são processos que são ativados irregularmente em um período desconhecido; e **Esporádicas**, em que os

processos são ativados irregularmente também, porém com um intervalo mínimo entre duas invocações consecutivas do mesmo processo.

2.2.2. Classificação de escalonamento

O problema geral de escalonamento é criar uma ordem em que cada processo no estado de pronto irá executar e se as várias restrições impostas pelo sistema serão satisfeitas. Para que essa ordenação seja válida para determinado conjunto de processos, o escalonador deve atribuir prioridades a estes; geralmente tarefas críticas são computadas antes das não-críticas.

Como exemplo podemos citar um conjunto de processos em que o mais prioritário será o que executa mais vezes dentro de uma determinada faixa de tempo, o algoritmo usado pelo escalonador classifica os processos pela ordem de prioridade. Seguindo essa linha, essa classificação pode ser **Estática**, em que todos os processos são classificados em tempo de projeto (*off-line*), em que fatores como *deadline*, tempo de execução, tempo de ativação entre outros são levados em conta; ou **Dinâmica**, onde os processos são classificados em tempo de execução (*on-line*). Outras classificações são estabelecidas para algoritmos de tempo real:

- **Sistemas Monoprocessados** e **Sistemas Multiprocessados**: O número de processadores é um dos principais fatores para se escolher um determinado algoritmo de escalonamento em sistemas de tempo real. Sistemas monoprocessados são mais simples de estabelecer regras e provar se a escala gerada pelo escalonador é válida ou não [17]. Em sistemas de tempo real multiprocessados, o algoritmo de escalonamento deverá prevenir acesso simultâneo a recursos compartilhados e dispositivos de forma mais eficiente do que em sistemas monoprocessados.
- **Sistema Preemptivo** e **Sistema Não preemptivo**: Em alguns modelos de algoritmos de escalonamento, podemos dizer que um sistema é preemptivo quando um processo em execução pode ser interrompido por outro processo de maior prioridade. Já em sistemas não preemptivos um processo deverá cumprir a sua execução sem ser interrompido, desde que seja iniciada sua execução.
- **Processos Dependentes** e **Processos Independentes**: O conceito de dependência pode ser exemplificado pelo seguinte exemplo: dado um sistema de

tempo real, um processo que está pronto para executar, só poderá ser inicializado, se e somente se, não exista processo inicializado anteriormente do qual o primeiro dependa, e que ainda não tenha finalizado sua execução. Isso ocorre porque um processo pode depender de informações ou recursos que outros processos também utilizem para sua execução. Processos dependentes usam memória compartilhada ou dados de comunicação para transferir informações geradas por um processo e requerido por outro. Já em sistemas com processos independentes, estes não dependem das informações ou recursos usados por outros processos.

2.2.3. Modelos de escalonamento para tempo real

Muitas pesquisas têm sido realizadas no campo de escalonamento para sistemas de tempo real, tanto para sistemas críticos quanto não-críticos. As referências mais freqüentes de algoritmos de escalonamento, que servem como base para a maioria das pesquisas atuais, são os modelos RM e EDF de Liu e Layland [32], os quais são considerados ótimos para classes específicas de problemas em tempo real.

Para o RM ser ótimo o ambiente deve ser monoprocessado, com prioridade fixa e preemptivo, devendo se assumir que todas as tarefas são independentes, que possuem seus *deadlines* iguais aos períodos e que o tempo para troca de contexto entre os processos é considerado nulo. O EDF é um exemplo de algoritmo orientado a prioridade com atribuição dinâmica, em que quanto mais próximo está um *deadline* de um determinado processo, maior será a sua prioridade.

Também Liu e Layland [32], provaram que EDF é ótimo em um modelo de processos preemptivos executando em um único processador e sem recursos compartilhados. Esses algoritmos, no entanto não são muito úteis em aplicações reais em que a troca de contexto entre as tarefas não possa ser desprezada e que tenham interdependência.

Já para sistemas multiprocessados o EDF é demonstrado como ótimo no trabalho feito por Baker [2] que apresenta um novo teste de escalonabilidade baseado no conceito de análise chamada de intervalo μ-*busy* em que o um conjunto de processos é escalonável pelo EDF para *m*-processadores se a utilização total não passar de $m(1-u_{\max})+u_{\max}$, onde *u* é a máxima utilização individual de um processo. Porém o mesmo só é válido para sistemas multiprocessados homogêneos, ou seja, que tenha

todos os processadores idênticos, não sendo válido para sistemas heterogêneos, que são aqueles em que os processadores são diferentes, além de que o estudo é voltado para sistemas periódicos e o conjunto de processos são independentes.

Outro trabalho feito por Andersson *et al* [1] também utiliza o EDF para sistemas multiprocessados, porém para carga aperiódica em que os tempos de chegada são desconhecidos. Um algoritmo previamente estabelecido pelos autores para escalonamento dirigido a prioridades para carga periódica foi estendido para que pudessem utilizá-lo para carga aperiódica. Esse algoritmo recebeu o nome de *Global Priority-driven Scheduling* em que a escala é completamente definida pelo número de processadores e outros atributos como tempo de chegada, *deadlines* e tempo de execução. É considerado que o sistema possui processos independentes entre si e possui um alto *overhead* gerado na criação da escala, devido à grande quantidade de processos.

Uma técnica de escalonamento baseada em recursos virtuais de tempo real é um meio de controlar recursos compartilhados e processos que sejam dependentes entre si. A noção de recursos virtuais foi introduzida para abstrair os diversos recursos compartilhados como memória e processadores, principalmente quando são utilizados por um grupo de processos que possuem diferentes tempos de chegada, *deadlines*, tempo entre processos (*jitter*), etc. O compartilhamento é feito seguindo um esquema de tempo em que as aplicações não interfiram umas nas outras e que a execução seja feita no tempo mais rápido possível. Uma descrição mais clara do trabalho nessa área pode ser obtida com o trabalho de Feng e Mok [18].

Outros modelos de escalonamento mais atuais levam em consideração o consumo de energia dos dispositivos móveis, como o descrito no trabalho de Pillai e Shin [43], também conhecido como *Dynamic Voltage Scaling* (DVS). Já que, com a vida limitada das baterias, gerenciamento de energia é crucial em certas situações, os atuais processadores como o *xScale* da *Intel*[1] ou o *Crusoe* da *Transmedia* utilizam recursos de gerenciamento de energia, porém, poucos são os aplicativos que tiram proveito desse recurso. Na Figura 3 podemos observar um esquema simples de como funciona o DVS.

---

[1] A *Intel* vendeu sua linha de processadores *xScale* para a *Marvell Technology Group* (notícia disponível em http://marvell.com/press/pressNewsDisplay.do?releaseID=680 no dia 27 de junho de 2006).

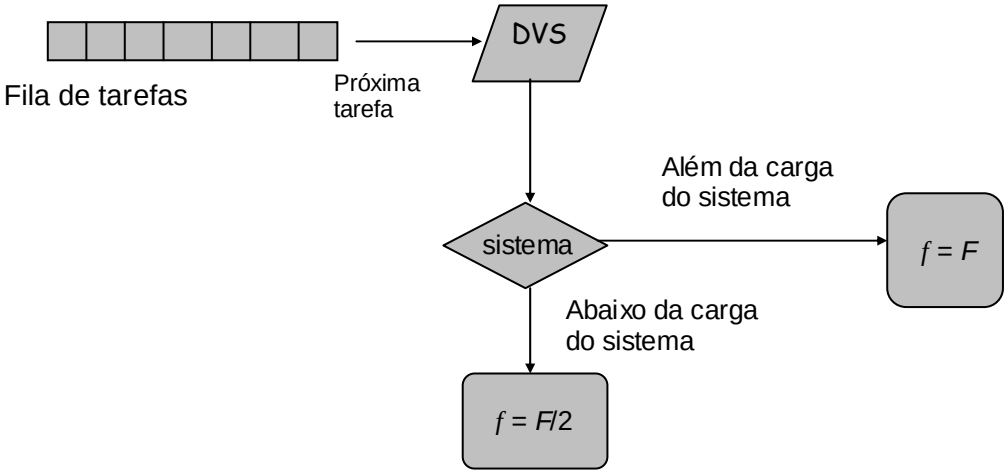


Figura 3 - Esquema simples do uso do DVS [43]

Nesse modelo, a escala gerada para determinar a execução das tarefas no processador opera junto com a freqüência do relógio do sistema. Desde que a energia é proporcional a f^2, DVS pode potencialmente fornecer um controle de energia através da freqüência e a escala. Para um melhor entendimento verifique o exemplo. Uma tarefa (T_1) leva 20 unidades de tempo para ser executada e seu *deadline* é de 30 unidades de tempo. Se a energia é proporcional à freqüência, podemos subentender que obedece a seguinte expressão [43]:

$E_n = K \times C \times f$, onde E_n é a energia utilizada pela tarefa, K é uma constante para calculo de interferência, C é o tempo de computação da tarefa e f é a freqüência relativa desta tarefa.

A energia gasta pela tarefa T_1 sem o uso do DVS é $E_1 = K \times 20 \times f^2$ e com o uso do DVS é $E_2 = K \times 20 \times \left(f/2 \right)^2$, isso implica que $E_2 = E_1/4$.

No entanto, se reduzir à freqüência, o sistema consome menos energia, mas gasta mais tempo para executar a tarefa. Se a carga do sistema estiver alta T_1 executará em sua freqüência normal e terminará a sua execução em tempo previsto, por outro lado se T_1 estiver executando em uma carga baixa do sistema a escala DVS será utilizada, fazendo com que freqüência do processador caia a metade e leve 40 unidades de tempo para ser executada, perdendo seu *deadline*.

Não se deve utilizar DVS em sistemas de tempo real sem um estudo aprofundado de como as tarefas serão executadas e qual a carga máxima que o sistema

irá utilizar. Para sanar essa limitação do DVS em sistemas de tempo real, foi proposto por Shin e Kim [57] um esquema conhecido por *intra-task* em que as tarefas críticas possam ser utilizadas com um esquema de economia de energia e poder cumprir com os seus *deadlines*.

Nesse contexto vemos que o processador é a figura crucial em sistemas embarcados e ele por si só consome uma grande parte dos recursos de energia destinada a esses dispositivos. Existem dois modos básicos de economizar energia no processador em dispositivos embarcados: um é o desligamento (*shutdown*) e o outro é a execução lenta (*slowdown*) do processador. Estudos feitos por Lee *et al* [31] apresentam um algoritmo *off-line* para escalonar um dado conjunto de processos com seus respectivos tempos de chegada e *deadlines*, utilizando alternativas de deixar lacunas entre a execução dos processos para que o processador possa entrar no modo *shutdown* ou *slowdown*. A solução é ótima se considerarmos uma faixa de energia contínua e adotar o modelo EDF, porém não é válido se forem consideradas sincronizações entre os processos e para alguns modelos de tarefas o algoritmo não gera uma escala válida.

Isso foi demonstrado por Jejurikar e Gupta [25] que, no mesmo trabalho, propuseram uma alteração no modelo de Lee *et al* [31] que junto com o algoritmo de escalonamento *Procrastination Scheduling*, atrasam ao máximo o tempo de execução dos processos prontos, considerando um sistema de prioridades fixas e que os processos sejam independentes. Com os processos sofrendo atrasos, nesse espaço de tempo em que não são usados os recursos do processador o mesmo utiliza-se de um dos dois meios de economia de energia. Nota-se, porém que nesse novo modelo não se considerou novamente processos que sejam dependentes e que concorram a um determinado recurso compartilhado.

2.2.4. Modelos de Algoritmos de Escalonamento

No trabalho de Kim *et al* [26], os autores demonstraram através de uma experimentação, em que criaram um serviço que funciona com o escalonador do *Unix* para que o sistema pudesse trabalhar com processos de tempo real. Nesse trabalho eles consideraram como métrica base o *deadline* dos processos de tempo real para que pudessem criar o método de implementação do escalonador no *Unix*. Eles observaram que as escalas geradas pelo escalonador do sistema operacional são apenas para processos que estejam no estado de pronto. Com base nessa observação implementaram

um serviço do *Unix* que sempre aparece como pronto antes de qualquer processo que não seja de tempo real. Isso faz com que o escalonador veja antes de quaisquer processos do sistema, as tarefas de tempo real.

No entanto o fator limitante do trabalho de Kim *et al* [26] é que utilizam o *Unix* que não possui nenhuma extensão de tempo real na sua API, não possuindo nenhuma garantia concreta que em sistemas de uso prático mostre-se satisfatório, ou que venha a cumprir os requisitos temporais dos processos. O algoritmo que eles modificaram foi o EDREL (*Earliest Deadline Relative*) que estabelece os níveis de prioridade baseado no *deadline* dos processos e seus tempos de chegada. O uso de um serviço de sistema operacional também não é ideal, já que o sistema para poder escalonar tarefas de tempo real tem de esperar que o sistema como um todo inicialize e após isso carregue o aloque no espaço de memória do *kernel* as funcionalidades de tempo real. Poderia ser mais bem flexível de implementar se no lugar de usar o *Unix* fosse utilizado o *Linux* e criasse uma modificação do *kernel*, assim garantiria que os processos de tempo real fossem prontamente inicializados junto com o sistema.

Pedro *et al* [42] fizeram uma análise de escalonabilidade de tarefas no sistema de tempo real S.Ha.R.K. [52], para comprovar o funcionamento do escalonador do sistema operacional de tempo real. A análise consistiu em fazer um teste de escalonabilidade usando os algoritmos RM e EDF, dentro de um conjunto de processos que fossem escalonáveis pelo EDF e não escalonáveis pelo RM. A prova envolveu dois experimentos simples.

No primeiro experimento três processos fazem um conjunto de funções: o primeiro processo faz um cálculo exponencial de um valor *float*, o segundo processo realiza um cálculo para determinar a posição em que um texto será exibido e em seguida o imprime na tela do monitor e o último processo exibe um texto simples em uma posição estática sem nenhum cálculo. No segundo experimento eles consideram um conjunto de cinco processos que fazem o mesmo procedimento, em que uma esfera se movimenta pela tela do monitor, porém dentro de um espaço pré-definido. Para os dois experimentos foram considerados os processos periódicos e com *deadline* igual ao período.

A idéia dos dois experimentos é atribuir valores de periodicidade aos processos, fazendo com que tenham uma taxa de utilização do processador entre 0,7798 e 1 para o primeiro experimento e de 0,7435 e 1 para o segundo. Esses valores estão de acordo com o cálculo para fator de escalonamento segundo o RM [32]. De acordo com essa

equação, para que um conjunto de três processos seja escalonável pelo RM tem de ter taxa de utilização menor ou igual a 0,7798 para o primeiro experimento e de 0,7435 no segundo.

Através de funções de estimativas de tempo de execução, da biblioteca `kern.h` disponível no SOTR, obteve-se os seguintes tempos de computação: para o primeiro experimento, processo 1 de 8 µs, processo 2 de 21 µs e processo 3 de 21 µs; já no segundo experimento por se tratar de processos idênticos o mesmo tempo de computação foi atribuído sendo igual a 28 µs. Os valores de periodicidade atribuídos às tarefas foram para o primeiro experimento: processo 1 de 50 µs, processo 2 de 60 µs e processo 3 de 60 µs; e para o segundo experimento: processo 1 de 125 µs, processo 2 de 200 µs, processo 3 de 250 µs, processo 4 de 140 µs, processo 5 de 112 µs. Sendo assim a taxa de utilização do processador será mais elevada do que o fator de escalonamento para os dois experimentos. Portanto não são escalonáveis pelo RM e sim pelo EDF em que a taxa de utilização é menor que 1. Ao executar o conjunto de processos pelo algoritmo EDF no sistema operacional de tempo real não ocorreu nenhuma interrupção da aplicação, o mesmo não ocorreu quando executada pelo escalonamento RM em que as aplicações foram interrompidas, pois as mesmas não eram suportadas pelo escalonador.

No entanto o estudo de Pedro *et al* [42] não levou em consideração outros fatores como a periodicidade da tarefa escalonadora, além de fatores que causam interferência do sistema como um todo, a exemplo de uso de *pipeline*, memória cache, entre outros. Outro fator que gera dúvida é que o teste de escalonabilidade é suficiente, seguindo a abordagem do RM de acordo com a taxa de utilização do processador, sendo que, entre os processos descartados por esse teste pode haver algum escalonável.

2.3. Sistemas Operacionais de Tempo Real

2.3.1. Definições

Quanto à solução os STR podem ser classificados em **Sistemas Operacionais de Tempo Real** e **Núcleos de Tempo Real**. Basicamente Sistemas Operacionais têm como função básica gerenciar os recursos de *hardware* e de *software* do computador, além de servir de interface com o usuário que o utiliza. Núcleos de Tempo Real serão explicados mais adiante.

Para Farines *et al* [17] Sistemas Operacionais de Propósito Geral (SOPG) possuem dificuldades em atender as demandas específicas das aplicações de tempo real, não possuindo em nenhum momento preocupação com a previsibilidade temporal, requisito indispensável para o bom funcionamento de tais aplicações. Porém sistemas atuais como o *Linux* já na versão 2.6.8.1, já possuem em seu escalonador propriedades para a utilização de aplicações não-críticas, especificadas pelo padrão POSIX[2] 1003.1b e 1003.1c[3] [13].

Os SOPG tipicamente são projetados para um melhor desempenho médio, ou seja, para que a maioria das tarefas sejam atendidas justamente entre os processos e que estas utilizem os recursos da máquina de forma igualitária. Certas funções do SOPG são escondidas dos projetistas e/ou programadores de sistemas, fazendo com que certas funcionalidades não possam ser alteradas de forma direta.

Em SOTR as mesmas funcionalidades de auxílio dos SOPG são encontradas, porém são, em geral, sistemas abertos onde os mecanismos que antes eram escondidos nos SOPG agora estão acessíveis. Isso se deve às propriedades inerentes aos sistemas de tempo real que são a corretude lógica (*correctness*) e a corretude temporal (*timeliness*), sendo as duas de igual importância para garantir a previsibilidade que pode ser considerada o coração de sistemas dessa natureza.

Essa característica se aplica devido ao tempo estabelecido, pois as tarefas que estão sendo executadas devem ser cumpridas em prazos previamente conhecidos. Visibilidade e controle de recursos de *hardware* são fundamentais em tais sistemas. Várias funcionalidades facilitadoras encontradas em SOPG tornam o comportamento dos SOTR pouco previsíveis, tais como o uso de memória *cache* que gera um tempo adicional pouco previsível no processamento (*overhead*) quando o acesso a segmentos de dados não é correta e produz uma perda de *cache* (*cache miss*).

Outros recursos extras de *hardware* também comprometem a previsibilidade como placas de rede que usam o protocolo de *Ethernet* CSMA/CD (*Carrier Sense Multiple Access/Collision Detection*), no qual as estações observam o barramento de dados e somente realizam a transmissão da informação quando o barramento estiver livre, caso ocorra colisão, o pacote de dados é retransmitido após um intervalo aleatório.

---

[2] POSIX (*Portable Operating Systems Interface*). Usamos o termo como referência tanto para a POSIX quanto suas extensões de tempo real.

[3] Mais adiante serão explicados tais padrões.

Esse comportamento é fundamentalmente não determinístico, pois não garante em quanto tempo esse intervalo será realizado.

Já em Núcleos de Tempo Real encontramos uma solução que tenha garantias temporais bem maiores do que em SOTR. Podemos dizer que temos funcionalidades mínimas em um *microkernel*, porém com ótima garantia temporal, ideal para ser usado em aplicações críticas de pequeno/médio porte e que tenham uma interface simples com o usuário (não gráfica), como controle de plantas industriais, sistemas automotivos, etc. A classificação, quanto à solução, pode ser mais facilmente entendida de acordo com a Figura 4:

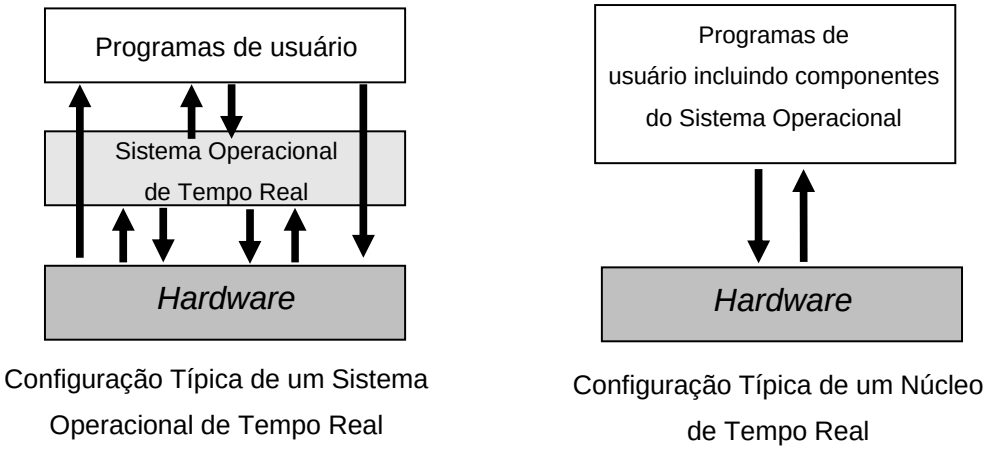


Figura 4 - SOTR e Núcleo de Tempo Real

Em sistemas embarcados, geralmente o uso de SOTR é desnecessário, já que muitas das funcionalidades podem ser encontradas e implementadas apenas pela linguagem de programação em que o sistema foi originalmente escrito. Exemplo de tais linguagens é ADA, na qual várias características interessantes, inovadoras e influentes para tempo real e aplicações deste tipo são implementadas [5], como o *programming-in-the-large* que é a capacidade de reuso.

2.3.2. O padrão POSIX

Muitos dos atuais SOTR possuem conformidades com o padrão POSIX nas extensões 1003.1-1003.13 PSE51 e PS52 para tempo real [46], total ou parcial. Este padrão desenvolvido pelo *Institute of Electrical and Electronic Engineers* (IEEE) define a interface e ambiente de sistemas operacionais, incluindo os interpretadores de comando e programas. A iniciativa da IEEE para este padrão de definição é importante,

pois homogeneíza os diversos projetos para tempo real, não sendo, porém fundamental para a construção de um SOTR completo, seguindo todas as restrições temporais.

Algumas empresas possuem seu próprio padrão para tempo real [21], [37] e [51], sendo que a maioria deixa que seus sistemas sejam moldados conforme descrições dos projetistas.

O padrão POSIX é composto basicamente pelas Definições de Base que inclui termos gerais, conceitos e interfaces comuns; Interfaces do Sistema, sendo estes que definem os serviços de funções do sistema para a linguagem de programação C, portabilidade, erros e recuperação; *Shell* e Utilitários, contêm as definições para os padrões de códigos no nível de interface para serviços que utilizam interpretadores de comando e seus utilitários; *Rationale* que possui informações que não são encaixadas em nenhuma das estruturas anteriores.

A versão atual da revisão do POSIX é o *Open Group Base Specifications Issue* 6 e as extensões para tempo real estão definidas em *IEEE Std* 1003.1-2001. O sub-padrão para tempo real seguindo essas definições são:

- 1003.1b: Extensão para Tempo Real (filas, também conhecidos como *queues*);
- 1003.1c: *Threads*;
- 1003.1d: Extensões para Tempo Real adicionais;
- 1003.1j: Extensões para Tempo Real com características avançadas;
- 1003.1q: *Tracing*;
- 1003.5: Própria para Linguagem Ada para 1003.1;
- 1003.5a: Atualizações para Ada;
- 1003.5b: Ada para Tempo Real.

Padrões para outros tipos de linguagem de programação, como C, são definidos no documento POSIX 2003.1b 2000 [44]. As extensões de tempo-real no padrão POSIX fornecem funcionalidades que são desejáveis em um SOTR. Entre essas funcionalidades podemos citar: filas de mensagens (comunicação entre tarefas - *intertask communication*), sistemas multitarefa (*multitasking*), memória compartilhada, semáforos (usado como forma de sincronismo entre tarefas), escalonamento (principalmente baseado em prioridades), sinais e temporizadores de alta resolução (que podem gerar intervalos de tempo com resolução de menos de um microssegundo).

SOTR modernos são baseados em conceitos complementares de comunicação *multitasking* e *intertask*. Um ambiente *multitasking* permite a uma aplicação de tempo real ser construída como sendo um conjunto de tarefas[4] independentes, cada uma com sua linha de execução (*threads*) e conjunto de recursos do sistema. Já a comunicação *intertask* facilita que tarefas sejam sincronizadas e comunicadas em uma ordem de acordo com sua ativação.

Como o padrão POSIX se preocupa basicamente com a interface em que os sistemas de tempo real devem ter, é difícil considerar esta padronização como básica e completa para um SOTR. Aspectos relacionados à previsibilidade não são tão claros, como a forma de implementação do *kernel* do sistema, tempo de troca de contexto entre as aplicações, tempo máximo que uma chamada de sistema (*system call*) deve ter, entre outras restrições que devem ser observadas pelos projetistas do SOTR.

2.3.3. Uso de Sistemas Operacionais de Tempo Real para aplicações embarcadas

Na medida em que ocorre a diversificação das áreas de aplicação de tempo real, os sistemas computadorizados tornam-se mais complexos e conseqüentemente os respectivos sistemas que os controlam também. A competitividade existente no mercado requer produtos sempre com mais funcionalidades e custos menores, o que leva ao uso de controle de tempo real baseado em microprocessadores, entre eles o MIPS, *xScale*, ARM, etc [49].

Entre os sistemas de tempo real podemos destacar aqueles identificados como sistemas de tempo real embarcado (*embedded real-time systems*). Um sistema computacional embarcado corresponde a um ou mais microprocessadores, um sistema operacional e um software aplicativo que ficam inseridos em um produto maior para processar as funções de controle deste produto.

Um sistema computacional embarcado deve suportar apenas um conjunto restrito de funções, definidas pelo equipamento maior no qual ele está inserido. Por exemplo, um computador de mão (*hand-held*), televisores e o controle computadorizado do motor de um automóvel têm de estar disponíveis a qualquer hora e o mais rápido possível sempre que o sistema é inicializado. Observe o gráfico da Figura 5.

---

[4] No presente trabalho as palavras tarefa e processo são consideradas sinônimos.

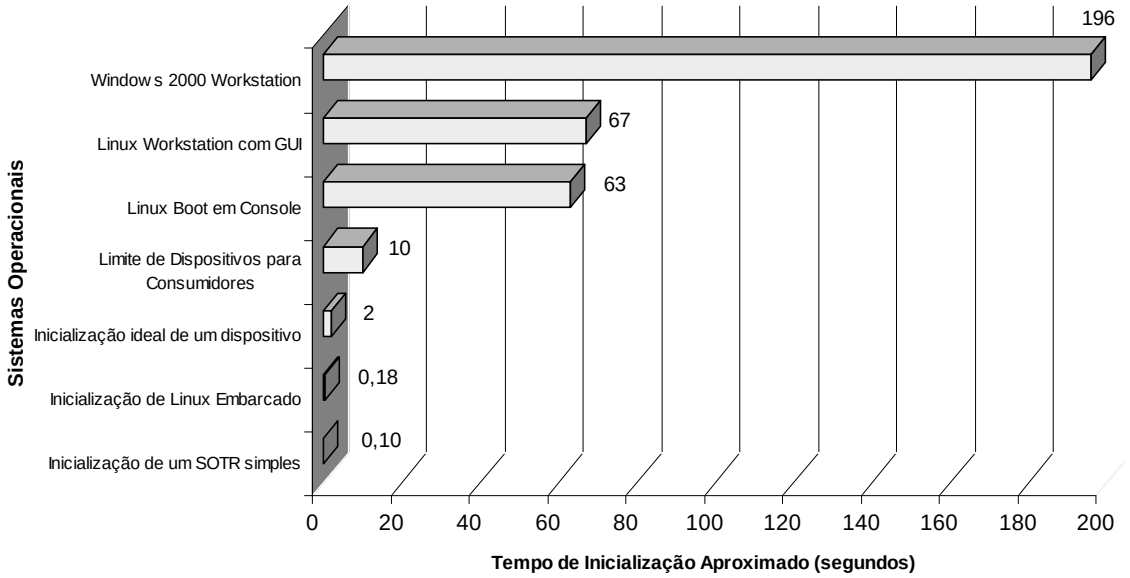


Figura 5 - Comparação entre os tempos de inicialização de SOPG, sistemas embarcados e sistemas de tempo real[5].

No gráfico exposto pela Figura 5 o tempo de inicialização de alguns SOPG é comparado com o tempo desejado de inicialização de sistemas embarcados e SOTR. Observa-se que no caso dos SOPG como *Windows* e *Linux* o seu uso em aplicações embarcadas se torna inviável, já que o tempo de inicialização dos sistemas é bem superior ao comparado a outras soluções, o que pode ser notado no caso do limite máximo aceitável para inicialização de dispositivos eletrônicos que deve ser por volta de 2 segundos. No caso de uma inicialização simples de um SOTR esse tempo deve ser preferivelmente abaixo de 1 segundo.

2.3.4. Exemplos de SOTR

O mercado de SOTR vem crescendo continuamente e a cada dia novas tecnologias são incorporadas a tais sistemas. Alguns destes sistemas se destacam mais que os outros ou por causa de suas funcionalidades e compatibilidades ou por causa de algumas características específicas, como suporte, facilidade de uso, preço, etc.

A seguir serão mostrados alguns dos principais SOTR: *VxWorks* 6, *Windows* CE.NET, OCERA 1.0, QNX *Neutrino*. As principais características como arquitetura, manipulação de processos e estruturas serão mostradas. Uma lista com uma grande

[5] Referência: Notas de aula do Professor Sérgio Cavalcante, da disciplina Sistemas de Tempo Real.

variedade de SOTR pode ser obtida no endereço da *Wikipedia* [48] e testes específicos podem ser obtidos nos documentos da *Dedicated Systems Encyclopedia* [14].

VxWorks 6

O sistema operacional *VxWorks*, desenvolvido pela empresa *Wind River*, é utilizado freqüentemente em sistemas em que envolve robótica, aviação, sistemas de controle médico, simuladores aeroespaciais e controle bélico [21]. Entre seus clientes estão a NASA (*National Aeronautics and Space Administration*), o Departamento de Defesa dos Estados Unidos, Agência Espacial Européia, além de algumas montadoras de automóveis como BMW, *Nissan* e *Hyundai*.

A base do sistema é constituída por um *microkernel* que provê suporte a multitarefa, suporte a interrupção por software e hardware, mecanismos de comunicação entre tarefas, das quais podemos citar memória compartilhada, trocas de mensagens, semáforos, chamada remota de procedimentos e sinais. As tarefas do SOTR são executadas quase que totalmente em modo *kernel*, porém podem ser executadas em modo usuário também.

No modo *kernel* as tarefas são inicializadas na mesma área que o *kernel* central do SOTR, ficando protegidas a acessos não privilegiados. Já em modo usuário as tarefas possuem cada uma um espaço de endereçamento próprio.

A diferença básica entre os dois modos é que caso ocorra falha em uma tarefa no modo *kernel* o sistema como um todo fica comprometido. Isso não ocorre no modo usuário, em que a tarefa comprometida pode ser excluída sem maiores danos ao sistema central. Observando a Figura 6 podemos ver a arquitetura geral do SOTR em questão.

Apesar desta vantagem, tipicamente os processos de tempo real (PTR) do *VxWorks* utilizam o modo *kernel* como o preferencial e quase sempre este modo de execução se torna mais confiável por que utiliza a mesma área que o núcleo central, evitando, desta forma, que seja interrompido por outros processos.

Cada PTR pode fazer acesso a uma área compartilhada para troca de informações com outros PTR que estejam no modo *kernel* e também que estejam em modo usuário, porém memória compartilhada não é a única forma de comunicação entre os processos.

Um conceito de proteção denominado de **domínio protegido** é incluso no *VxWorks*. Um domínio protegido pode ser descrito como diversas caixas em que cada

uma possui o seu próprio espaço de endereçamento virtual, que pode ou não ser visível em outros domínios. Essa divisão em vários espaços virtuais faz com que o sistema se torne mais robusto já que cada área tem suas regras próprias e não influenciam diretamente outros PTR. O *kernel* do SOTR deve ser executado em seu próprio domínio protegido, porém com regras de acesso mais restritas.

A execução de aplicação em áreas isoladas do *VxWorks* faz com que esta seja independente e provê áreas para código, dados e outras estruturas separadamente. A tarefa ganha todos os recursos associados com a aplicação, além de proteção de memória. O SOTR *VxWorks* suporta os processadores: Motorola 68k/CPU32, *ColdFire*, *Power*PC, *Intel* x86/*Pentium*/IA-32, ARM/*Strong*ARM, Hitachi, *Super*H, SPARC, i960, DSP, *xScale* e MIPS [21].

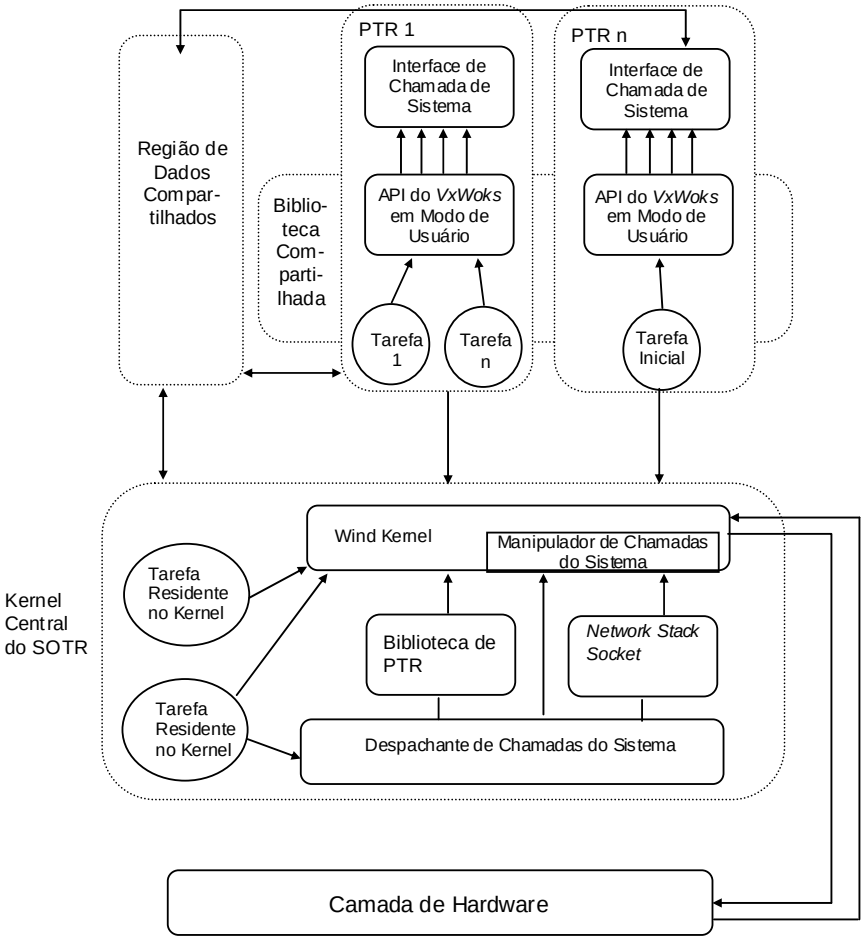


Figura 6 - Arquitetura geral do *VxWorks* [21]

O modelo de programação de um PTR tem compatibilidade com o modelo de programação do POSIX, especificamente do perfil PSE51 e PSE52. O ambiente de desenvolvimento pode ser o GCC (*Gnu C Compiler*) ou outro proprietário como o

ambiente Tornado que vem com o próprio sistema, que é um IDE (*Integrated Development Environment*) gráfico para plataformas *Win32*, *Linux* ou *Solaris*. Este também suporta carregamento dinâmico de módulos. No tratamento de tarefas pelo *VxWorks* um PTR não é considerado uma entidade escalonável. A unidade de execução de um PTR é uma tarefa do sistema comum, e poderá ter múltiplas tarefas executando, seguindo o mesmo conceito de *threads*. Tarefas em um PTR compartilhado possuem o mesmo espaço de endereçamento e memória e não podem existir além deste espaço, pois assim mantêm a integridade de diversas partes do SOTR.

As criações dos PTR são separadas do carregamento da aplicação, e obedecem aos seguintes passos: A fase da criação de um PTR, que é uma atividade mínima para verificar se a aplicação não possui erros ou falhas, cria os objetos associados com o objeto, o contexto inicial de mapeamento da memória e a tarefa inicial que funciona dentro do PTR. Este estágio funciona com chamadas de sistema. A segunda fase é o carregamento e instanciamento do PTR que é executado dentro do contexto da memória de um novo processo e funciona em uma prioridade especificada pelo usuário. Não há nenhuma demanda de paginação dentro do *VxWorks*, assim os processos são carregados inteiramente quando são criados. Há uma latência para carregar e começar um PTR que pode ser relativamente longa. Se isso for considerado um problema, o PTR pode ser carregado antes do tempo, e sua tarefa inicial suspensa. Sem a demanda por paginação, as páginas nunca são carregadas do disco, assim não ocorre um atraso não determinístico durante a execução.

Existem 256 níveis de prioridade (de 0 até 255), sendo que 0 (zero) é o de mais alta prioridade. O limite de tarefas é definido pela quantidade de memória disponível. Contudo o sistema *VxWorks* não possuí nenhum serviço nativo que ofereça qualidade de serviços (*QoS*) e a compatibilidade com outros SOTR é limitada, baseando-se apenas no padrão POSIX 1003.1b. Outra limitação do sistema é a não inclusão de nenhuma função de *mailboxes* e o controle de inversão de prioridades não é feito diretamente pelo sistema mas pode ser feito com *mutex* e semáforos. Um outro fator limitante do sistema é que existem apenas duas políticas de escalonamento disponíveis: O padrão é o escalonamento de prioridade preemptivo e o outro é escalonamento *Round-Robin*.

Enquanto processos tradicionais são criados automaticamente pelo sistema, os domínios protegidos precisam ser configurados pelo projetista do sistema de tempo real. Esse tipo de configuração coloca uma responsabilidade extra nas mãos dos desenvolvedores, aumentando as chances de erros e de possíveis enganos.

Windows CE versão .NET

A fabricante de *software Microsoft*, possui uma família de produtos dedicada a sistemas embarcados, composta pelo *Windows* CE, *Windows* XP *Embedded* e *Windows Embedded for Point of Service*. O sistema operacional para aplicações de tempo real com recursos de conectividade é o *Windows* CE versão .NET. O sistema é aberto, escalonável, de 32-bits que integra recursos de tempo real com tecnologias já conhecidas como a API (*Application Program Interface*) dos sistemas operacionais *Windows*, usada como interface de programação de aplicativos. A arquitetura da plataforma pode ser visualizada pela Figura 7.

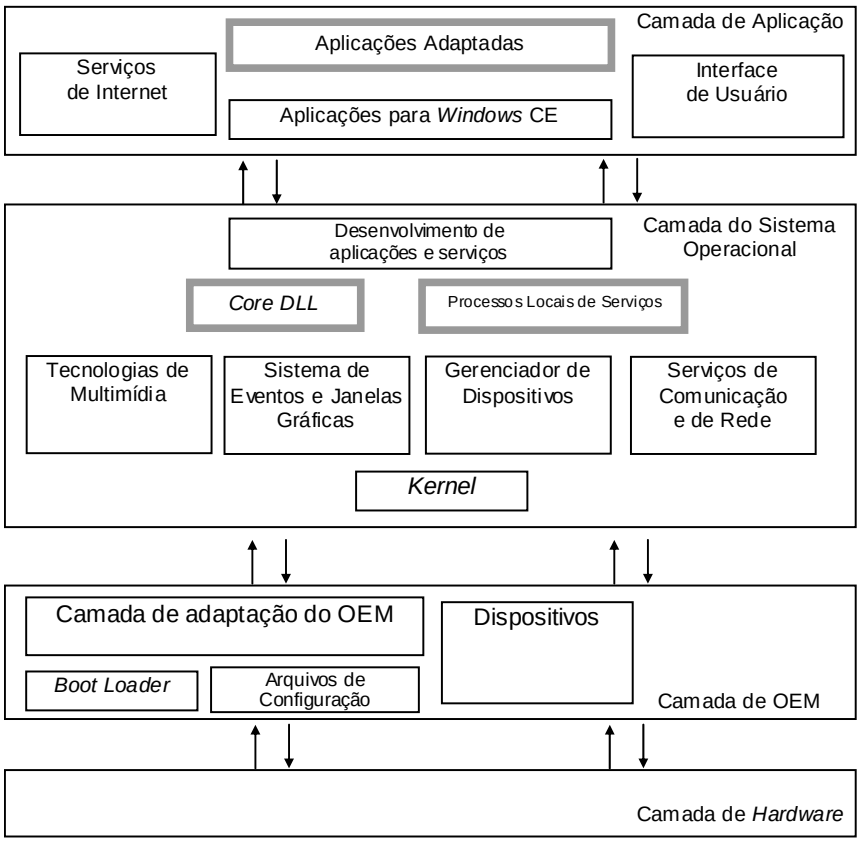


Figura 7 - Arquitetura do *Windows* CE [37]

O projeto modular da arquitetura faz com que o SOTR isole cada processo, assim não interfere em nenhum outro processo em caso de falhas ou mau funcionamento do sistema. Essa característica é típica de sistemas que executam em modo de usuário. A **Camada de *hardware*** é onde o sistema operacional faz interação

com a plataforma, na qual tem relação direta com a camada de OEM (*Original Equipment Manufacturer*), que trata dos Controladores de Dispositivos (*Devices Drivers*), o inicializador do sistema operacional (*Boot Loader*) e os arquivos de configuração além da **Camada de adaptação do OEM**. Esta última é a responsável pelo carregamento do sistema operacional na memória e o processo de inicialização. Ainda de acordo com o modelo da arquitetura, a **Camada do Sistema Operacional** é a que contém o *kernel* do SOTR, além dos principais serviços de multimídia, comunicação e gerenciamento de dispositivos.

A fabricante do sistema operacional representa o núcleo básico pelo módulo **nk.exe** (do acrônimo *new kernel*), que provê a base de todas as funcionalidades de qualquer dispositivo baseado na tecnologia do *Windows* CE versão .NET. Essas funcionalidades incluem processos, *threads* e gerenciamento de memória. O *kernel* do *Windows* CE versão .NET usa um sistema de paginação baseado em memória-virtual [49] para gerenciar e alocar memória para os programas. Esse sistema de memória-virtual utiliza blocos de memória de 1.024 *bytes* ou 4.096 *bytes* paginados em regiões de 64 *Kilo Bytes* (KB).

O SOTR é bastante flexível pois o sistema é construído sobre um conjunto de módulos em que cada um possui uma funcionalidade específica dependendo da arquitetura em que esteja funcionando o SOTR, em sua versão mais compacta o SOTR requer apenas 200 KB de ROM. Recursos como multimídia, *Internet*, *bluetooth* e gerenciador de dispositivos fazem parte do conjunto de objetos do sistema operacional que podem ou não serem utilizados pelo projetista. Quanto à sincronização o *Windows* CE versão .NET disponibiliza seções críticas, *mutex*, eventos e semáforos [51].

A camada mais externa do modelo de arquitetura do sistema operacional é a **Camada de Aplicação** em que são alocadas as interfaces gráficas com usuário e aplicativos, além de serviços de *Internet*. Muitos recursos personalizáveis são feitos nessa camada como serviços de rede, suporte a multimídia, construtores gráficos e recursos de segurança.

O escalonador do SOTR é baseado no *Round-robin* com fatia de tempo (*time-slice* ou *quantum*) ajustável pelo projetista do sistema. Possui 256 níveis de prioridade e pode ter 32 processos sendo executados simultaneamente.

Contudo o sistema *Windows* CE versão .NET possui algumas limitações, como número baixo de processos sendo executados por vez quando comparados a outros SOTR, não possui nenhuma função nativa para controle de inversão de prioridade,

sendo que para resolver este problema o fabricante aconselha usar herança de prioridades [37] que também não é o ideal já que processos podem entrar em bloqueio mortal (*deadlock*). Outro fator limitante no sistema do *Windows* CE versão .NET é a não padronização com o padrão POSIX de seu sistema, o que inviabiliza a compatibilidade com outras APIs. Além de que, segundo Chart *et al* [12] a maioria dos dispositivos comerciais que utilizam o sistema operacional *Windows* CE versão .NET são desenhados especificamente para este sistema fazendo com que alguns dispositivos de terceiros, que queiram usar esta plataforma, não sejam suportados.

Uma atualização da versão .NET, chamada *Windows Embedded* CE 6.0 foi lançada em abril de 2007 e algumas melhorias significativas foram acrescentadas para um melhor aproveitamento em aplicações de tempo real críticas, tais como: aumento de 32 para 32.000 processos executando em paralelo; otimização das chamadas do sistema deixando os recursos com uma maior previsibilidade em áreas separadas do *kernel*; separação do *kernel* em partes que facilita atualizações de módulos específicos do sistema e não ele como um todo; e atualização da biblioteca da linguagem C para maior compatibilidade com ambiente de desenvolvimento baseados em PC´s.

OCERA 1.0

OCERA é o acrônimo de *Open Components for Embedded Real-time Applications*, sendo um projeto da comunidade Européia para o desenvolvimento de um SOTR de código livre fundamentado no *RTLinux*. O projeto foi basicamente criado de dois núcleos: um do *kernel Linux* na versão 2.4.18 com uma série de melhorias (*patches*) para que pudesse ter comportamento de tempo real; e o segundo núcleo foi baseado na versão *beta* do *RTLinux Free* na versão 3.2 PRE2 da empresa FSMLabs.

O resultado do programa é a criação de uma série de bibliotecas que facilitam o comportamento previsível do *Linux*. Essas bibliotecas fazem parte de diversos componentes que podem ser utilizados fora do ambiente do SOTR, pois é uma forma de gerar flexibilidade, melhorias de configuração, portabilidade e robustez. O SOTR coloca funcionalidades extras no *Linux*, permitindo que os projetistas de sistemas embarcados possam criar aplicações utilizando o sistema de forma mais eficiente.

Ao todo o projeto possui cinco atividades principais divididas em treze grupos especiais, cada um com uma função específica dentro do projeto, para a construção de componentes específicos ou de análise do próprio projeto, que são divididas em [64]:

- WP1 – Estado da Arte e análise de SOTR;
- WP2 – Especificação da Arquitetura;
- WP3 – Análise de Mercado;
- WP4 – Desenvolvimento de componentes para gerenciamento de recursos;
- WP5 – Desenvolvimento de componentes de escalonamento de tempo real;
- WP6 – Desenvolvimento de componentes de tolerância a falhas;
- WP7 – Desenvolvimento de componentes para comunicação;
- WP8 – Estimativa dos resultados do projeto OCERA;
- WP9 – Validação da plataforma;
- WP10 – Treinamento e suporte técnico;
- WP11, WP13 – Gerenciamento de recursos de suporte tais como correio eletrônico, página na *Internet*;
- WP12 – Verificação dos resultados dos outros grupos.

Cada componente criado no projeto OCERA é projetado para abranger uma grande variedade de aplicações incluindo sistemas críticos ou não. Uma visão da arquitetura do SOTR pode ser observada na Figura 8:

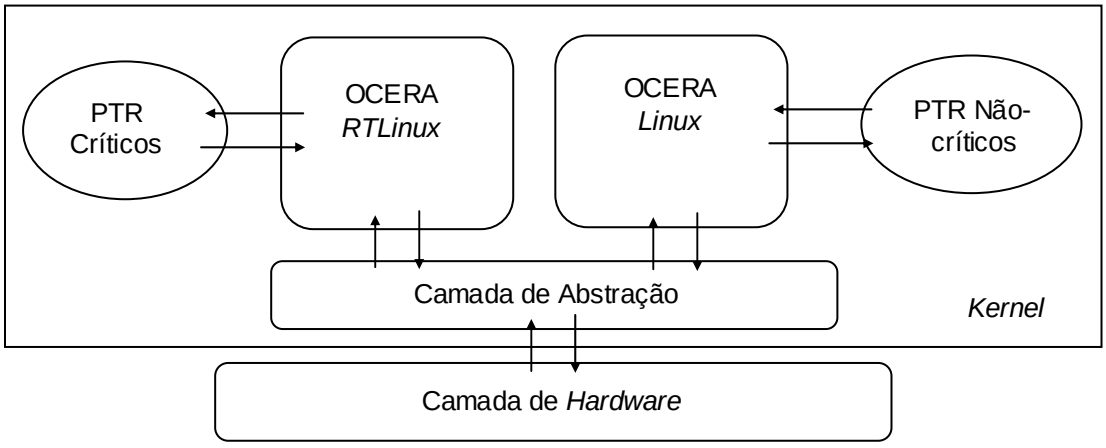


Figura 8 - Arquitetura do OCERA 1.0 [64]

As *threads* são executadas diretamente por um escalonador de prioridade fixa, sendo que esse coloca como mais baixa prioridade o kernel padrão, que executa como sendo um processo independente. A Camada de Abstração trabalha interceptando todas

as interrupções do *hardware*. Interrupções de *hardware* que não são classificadas pelo sistema como sendo de tempo real, são tratadas e então passadas para o *kernel* do *Linux* como uma interrupção de *software*, fazendo com que o núcleo de tempo real fique em estado de espera (*idle*) e o núcleo padrão fique em execução. Porém para controle do que seja de tempo real ou não, a rotina de serviço de interrupção (*interrupt service routine*) fica em execução.

O *kernel* do SOTR não é preemptivo, o que pode causar atrasos não previsíveis em algumas situações, porém por causa do seu tamanho reduzido (cerca de 200 KB) e operações limitadas (interceptação de interrupções de *software* e *hardware*) não permite que atrasos decorrentes da não preempção possam prejudicar o sistema. Os processos no sistema possuem dois atributos: **privilegiado**, no qual os processos acessam diretamente o hardware e os que **não usam memória virtual**. A escrita dos processos é feita como módulos especiais do *Linux* que podem ser dinamicamente alocados na memória.

A inicialização de um processo de tempo real é feita por uma estrutura que informa ao SOTR o seu *deadline*, período e tempo de chegada. Existem três políticas de escalonamento no sistema: SCHED_FIFO, SCHED_SPORADIC e SCHED_EDF. A primeira é baseada em prioridade fixa e processos com a mesma prioridade são escalonados numa ordem FIFO (*First In First Out*). A política de SCHED_SPORADIC é uma implementação do servidor esporádico [39] usada para executar processos aperiódicos e a última política implementa a prioridade dinâmica EDF.

Não há limite para o número de *threads* em execução, mas o custo do escalonamento é proporcional ao número de *threads*. O número mínimo e máximo de prioridades é 0 e 1000000 respectivamente. Processos aperiódicos também são suportados pelo sistema através do uso de interrupções. Quanto aos processadores do *hardware* suportado pelo sistema operacional incluem o x86, *Power*PC e ARM (inclusive o *Strong*ARM) [64].

O sistema tem uma compatibilidade muito grande com o padrão POSIX, incluindo barreiras, sinais, temporizadores, filas de mensagens e relógios, sendo que alguns são implementados em um padrão não compatível com o POSIX. A grande variedade de componentes, como adicionar previsibilidade a sistemas de arquivos, memória dinâmica e utilização de protocolos de rede para tempo real, fazem o SOTR uma boa opção a projetistas que necessitam de tais funções.

No entanto o projeto possui muitos dos seus componentes em estados de desenvolvimento, possuindo poucas opções estáveis e confiáveis. O número de

processadores ainda é um pouco limitado e grande partes das falhas que são encontradas nos sistemas *Linux* e *RTLinux* ainda não foram completamente ajustadas para o OCERA, tais como adicionar preempção ao *kernel* do *Linux*, maior resolução do relógio do sistema e criação de tarefas em modo de usuário.

QNX Neutrino

Sistema Operacional desenvolvido pela empresa QNX *Software Systems* tem sido muito utilizado principalmente para aplicações militares há mais de 20 anos [47]. Possui suporte ao padrão POSIX 1003.1 – 2001, principalmente em barreiras, relógios, sincronização de processos, proteção de memória, escalonamento de processos, memória compartilhada, sinais com extensão para tempo real, semáforos, threads, temporizadores, entre outras funções que obedecem ao mesmo padrão.

A arquitetura do sistema também segue o uso de *microkernel* em que a comunicação é basicamente feita por troca de mensagens. Essa troca de mensagens é uma barramento virtual criado por *software* que permite acrescentar ou remover módulos ou aplicações dinamicamente, ou seja, não é necessário interromper o sistema para a inclusão de novos recursos.

Essa arquitetura baseada em microkernel se assemelha muito a um modelo cliente-servidor, em que a unidade central (servidor) é o *microkernel* do SOTR e os clientes são os serviços específicos tais como sincronização, escalonamento, sinais e controle de aplicações. Cada processo executa em seu próprio espaço virtual. Como o núcleo central executa a comunicação via troca de mensagens esse pode distribuir de forma eficiente as diversas solicitações dos projetistas de tempo real. A Figura 9 mostra a arquitetura do SOTR.

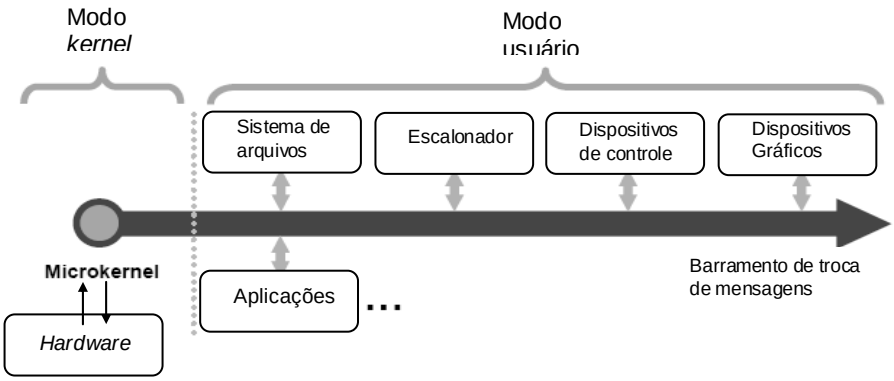


Figura 9 - Arquitetura do QNX Neutrino fundamentado em troca de mensagens [47]

Uma forma eficiente de controle, implementada no SOTR, faz com que caso ocorra falha em dispositivos de baixo nível, o sistema continue funcionando. Caso um componente ou dispositivo do sistema tenha alguma falha, o *kernel* poderá terminar a aplicação rapidamente ou então deixá-la em uma forma isolada do resto do sistema.

O sistema central pode gerenciar até 4095 processos sendo que cada um pode conter no máximo 32767 *threads*. As políticas de escalonamento são a FIFO, *Round-robin*, e um servidor esporádico que faz parte do padrão POSIX.

Funcionalidades adicionais são implementadas com um método de cooperação entre os processos, ou seja, um servidor de processos, definido pelo SOTR, atua como o responsável por responder requisições dos diversos serviços do sistema. Enquanto o núcleo central executa no mais alto nível de prioridade, definido por 0 (zero), os gerenciadores e outros servidores executam em níveis inferiores, definido por 1 ou 2.

Aplicativos podem executar em nível de privilégio 3 porém sem perder a sua funcionalidade temporal e previsibilidade. Essa forma de hierarquia faz com que o núcleo tenha um controle maior caso alguma aplicação venha a falhar.

Um ponto fraco relacionado ao QNX é que a inclusão de diversos componentes ao sistema faz com que o núcleo tenha por obrigação criar novos servidores de requisições, o que gera um *overhead* no processamento de aplicações. Conseqüência direta disso é um tempo maior de chaveamento entre os diversos processos do sistema.

O uso de recursos de rede e *Internet* não são tão desenvolvidos quanto o de outros concorrentes tais como o *VxWorks* e o *Windows* CE .NET. Suporta os seguintes processadores: x86, *Power* PC, ARM, MIPS e SH-4. Porém, alguns recursos adicionais disponível pela empresa [47] podem ser incluídos ao sistema operacional para que possa ter suporte ao processador *xScale*.

2.3.5.Quadro comparativo dos exemplos de SOTR

Quadro 1: Comparação dos SOTR citados como exemplos citando características.

SOTR	*Hardware* suportado	Suporte a SMP	Políticas de Escalonamento	Proteção de memória	Prioridades
VxWorks 6	Motorola 68k/CPU32, *ColdFire*, *Power*PC, *Intel* x86/*Pentium*/IA-32, ARM/*Strong*ARM, Hitachi, *Super*H, SPARC, i960, DSP, *xScale* e MIPS.	Sim.	Prioridade fixa e escalonamento *round-robin* (*time sharing*).	Não possui.	256 níveis de prioridades. Limite de tarefas executando limitado a quantidade de memória disponível.
Windows CE versão .NET	x86, MIPS, SH.	Não.	Prioridade fixa, *round-robin* (*time sharing*).	Sim, inclusive com alocação dinâmica de memória.	256 níveis de prioridades. 32 processos executando.
OCERA 1.0	X86, *Power*PC, ARM, MIPS.	Sim apenas a x86.	Prioridade fixa, servidor de tarefas aperiódicas e prioridade dinâmica EDF.	Não possui. Porém pode usar sistemas de terceiros para implementar tal recurso.	Processos com nível 0 (zero) possuem maior prioridade. Limite definido pelo desenvolvedor.
QNX Neutrino	X86, ARM, MIPS, *Power*PC, *Super*H. Pode ter suporte ao *xScale*.	Sim.	Prioridade fixa, servidor de tarefas aperiódicas e prioridade dinâmica EDF.	Sim, inclusive com alocação dinâmica de memória.	Três níveis de prioridades, sendo 0 (zero) de mais alta prioridade (recursos do sistema) e processos executando em nível 3.

Capítulo 3 - Avaliação Comparativa das Características dos Sistemas Operacionais de Tempo Real

---

Este capítulo irá avaliar os SOTR escolhidos para estudo, levando em consideração os pontos em destaque de suas características: Tamanho, Modularidade, Adaptabilidade e Previsibilidade além das premissas levadas em consideração para análise do presente trabalho.

Avaliar SOTR não é das tarefas mais fáceis isso porque a cada ano as aplicações de tempo real estão se tornando cada vez mais complexas e demandam por recursos que acompanhem também essa complexidade. Tais recursos, para esses sistemas, vão muito além de apenas um escalonador que supra as necessidades. Esses sistemas podem ser muitos caros e o custo de desenvolvimento incluído nesse tipo de projeto requer uma análise precisa e pode determinar se um sistema está apto ou não a uma solução segura.

O sistema operacional deve fornecer uma funcionalidade mínima suficiente para que possibilite a execução completa de aplicações de tempo real de acordo com suas restrições temporais. Quando avaliamos algumas características de vários sistemas operacionais, alguns critérios distintos de cada um têm um impacto maior na escolha de um determinado SOTR. Esses critérios são fontes necessárias de pesquisa por parte dos projetistas de tempo real que queriam utilizar um SOTR para suas aplicações.

A execução de um programa está diretamente ligada ao gerenciamento que o sistema operacional faz, pois é ele quem vai definir em que momento um dispositivo pode ser acessado por um programa, se um ou vários processos vão ser executados no computador ou qual a sua melhor forma de uso. Isso é genérico para qualquer sistema operacional seja ele *Windows*, *Linux* ou *MacOS*.

Também podemos dizer que um sistema operacional esconde dos projetistas e desenvolvedores de sistemas determinadas funcionalidades para otimizar seu trabalho. No caso específico de SOTR, adaptados ou não de SOPG, além dessa facilidade de transparência, devem possuir a característica de previsibilidade dos programas que irão executar nesse ambiente.

Um SOTR deve ser aberto, ou seja, define um conjunto de mecanismos apropriados e flexíveis mas não força políticas específicas sobre o usuário, auxiliando o comportamento de aplicações de tempo real. Também deve possibilitar a definição de um amplo domínio de políticas, por exemplo, políticas diferentes para escalonamento de tarefas, dependendo da aplicação. Caso contrário, é muito mais difícil construir sistemas de tempo real, cada um de um tipo e cada um sendo independente, que satisfaçam restrições específicas.

Com isso o SOTR que compõe em sua estrutura essas funcionalidades possibilita uma **visibilidade** e **controle** para todos os componentes do sistema. Muitos dos SOTR modernos possuem algum tipo de preempção, muito mais aprimorados do que em SOPG, minimizando o tempo entre a ocorrência de eventos e do escalonamento do sistema operacional. É mais comum observar que o recurso de preempção faz com que

um processo de mais alta prioridade do sistema possa interromper a execução de uma tarefa de mais baixa prioridade, a qualquer momento, para que dessa forma esta tarefa de mais baixa prioridade não possua os recursos compartilhados indefinidamente. Observe o exemplo da Figura 10.

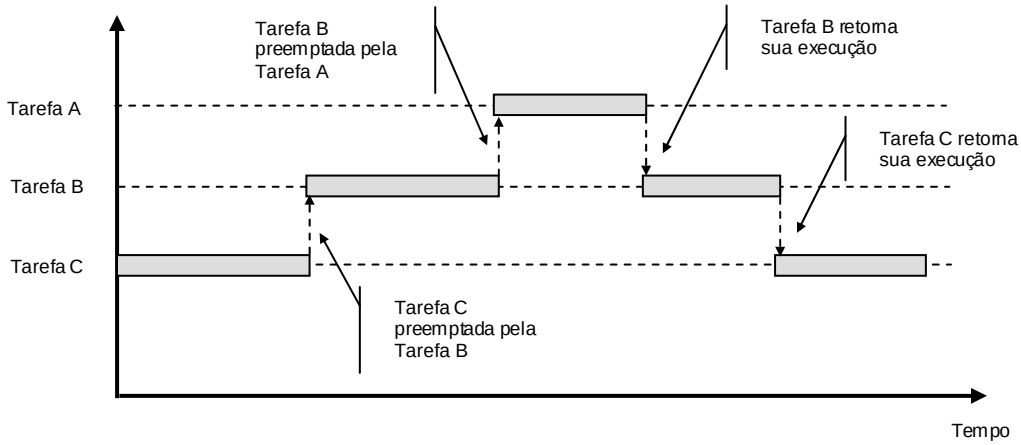


Figura 10 - Preempção de tarefas através do tempo

No exemplo exposto pela Figura 10, observamos três tarefas sendo que Tarefa A possui prioridade maior entre todas as tarefas, Tarefa C é de menor prioridade e a Tarefa B possui uma prioridade intermediária. A Tarefa B, ao entrar em execução, interrompe a Tarefa C e começa a sua execução, sendo que esta em um determinado momento também é interrompida para que a Tarefa A possa ser executada. Ao final da execução da Tarefa A as demais retornam a seu processo de execução e finalizam obedecendo a regra da prioridade.

A previsibilidade do SOTR faz com que as garantias para um sistema de tempo real que for incluso nesse ambiente seja executado de forma ideal dentro do prazo definido pelo seu Pior Caso do Tempo de Execução (*Worst Case Execution Time* ou simplesmente WCET). Determinismo, pior caso de latência de interrupção e tempo de troca de contexto são outras características que um SOTR deve ter. Um sistema operacional é dito como determinístico se o WCET de cada uma das chamadas de sistema possam ser calculadas. Esses tempos podem ser diferentes dependendo do tipo de processador utilizado ou fabricante do sistema operacional, no entanto é esperado que o sistema continue com a mesma funcionalidade independente do tipo de *hardware*.

A latência de interrupção é o intervalo total de tempo entre a chegada de um sinal de interrupção no processador e o início da rotina de serviço de interrupção (ISR,

de *Interrupt Service Routine*) associada. Quando uma interrupção ocorre, o processador deve seguir alguns procedimentos antes de executar a ISR. A primeira é que este deve finalizar qualquer instrução que esteja sendo processada, logo em seguida deve identificar o tipo de interrupção sendo que isso é feito pelo *hardware* e não interfere ou suspende qualquer tarefa sendo executada e, por último, caso as interrupções estejam habilitadas no sistema, o contexto é salvo e a ISR associada a esta interrupção é iniciada. Porém deve-se deixar claro que a maioria dos SOTR deve garantir que o tempo de resposta de uma interrupção seja o mais breve possível.

Outra característica diretamente associada à quantidade de tempo entre troca de processos é a quantidade de tempo necessária para que um sistema efetue a troca de contexto. Importante notar que essa característica representa grande parte do *overhead* gerado em todo sistema. Imagine que a média do tempo de execução de qualquer tarefa, antes dessa ser bloqueada por qualquer outra tarefa, seja de 100 ms e que o tempo da troca de contexto também seja de 100 ms. Nesse caso a metade do tempo do processador é gasto com a rotina de troca de contexto, mostrando dessa forma como essa característica deve ser bem estabelecida.

Para evitar tais problemas muitos dos SOTR possuem recursos que minimizam a troca de contexto, nesse caso podemos citar o uso do mesmo espaço de memória, fazendo assim que o tempo de troca entre tarefas seja muito baixo. Novamente o uso de um tipo específico de processador no sistema de tempo real se torna importante, pois dependendo do que for escolhido varia o número de registradores que devem ser salvos e aonde.

Alguns autores de trabalhos de análise em SOTR como Krishna e Shin [29] estabelecem quatro grandes grupos para comparar sistemas operacionais desse tipo, são eles: Performance, Segurança, Flexibilidade e Custo. O presente trabalho estuda quatro características específicas de SOTR, que são Tamanho, Modularidade, Adaptabilidade e Previsibilidade. De certa forma os grupos estabelecidos por Krishna e Shin [29] envolvem essas características, porém de forma subjetiva, sem a exibição de trabalhos concretos e também não estabelecem regras para este tipo de análise. No entanto tais características não proporcionam uma visão completa de um SOTR, devendo levar em consideração outros pontos.

3.1. Critérios de Avaliação de Sistemas Operacionais de Tempo Real

Quando comparamos diferentes *kernels* de tempo real é importante que a configuração do *hardware* escolhido seja equivalente ou igual, para que obtenha resultados compatíveis. A plataforma de *hardware* usada nos três sistemas escolhidos é uma arquitetura x86 e a sua configuração é: um computador do tipo PC, placa-mãe usada na avaliação é uma ASUS A7N8X-X, CPU AMD de 1.4 GHz (*Giga hertz*) com 256 KB de *cache* L2, placa gráfica de 32 MB (*Mega Byte*) no barramento PCI, memória RAM de 512 MB do tipo DDR 400 sem bufferização e disco rígido de 80 GB (*Giga Byte*), além de placa de rede nativa de 100 Mb/s (*Mega bit* por segundo).

Com o intuito de ajudar na análise um conjunto de premissas foram seguidas para que mostrassem características importantes sobre os SOTR:

- Deve possuir uma boa precisão de tempo;
- Os componentes necessários para implementar uma aplicação de tempo real devem ser leves e simples de implementar;
- Comportamento previsível sobre todos os cenários de carga do sistema, envolvendo tanto interrupções simultâneas e execução de várias tarefas;
- Uma tarefa de tempo real deve executar de tal maneira que, em caso de eventual falha, não interfira com outras tarefas e o sistema operacional como um todo;
- Recursos como paginação, *cache*, gerenciamento de energia e acesso a disco devem ser evitados e se possível, desabilitados a fim de impedir o comportamento anômalo do sistema e perdendo a previsibilidade;

No presente trabalho foi estabelecido que os pontos de análise devem representar uma eficiente pesquisa da informação relevante dentro de SOTR. Fornecendo uma base eficiente para verificação se os sistemas operacionais escolhidos são realmente projetados para sistemas de tempo real tanto tipo *soft,* como do tipo *hard*. Os critérios escolhidos foram Tamanho, Modularidade, Adaptabilidade e Previsibilidade.

3.1.1. Tamanho

Essa característica é levada em consideração em tais sistemas operacionais, pois define o quão grande ou pequeno este sistema operacional é, e se este tamanho pode ser ajustado para cada necessidade da aplicação de tempo real. Sistemas embarcados, em sua maioria, possuem tamanho limitado de armazenamento, não possibilitando que aplicações muito grandes sejam hospedadas, fazendo com que inviabilize a performance geral do sistema como um todo. Tendo isso em vista podemos dizer que alguns sistemas operacionais são capazes de se adaptar a plataformas de tamanho muito reduzido, porém possibilitando ainda uma funcionalidade mínima de sistemas de tempo real.

Alguns fabricantes de SOTR possibilitam que a imagem do sistema seja a mais simples e funcional possível. É muito difícil encontrar sistemas operacionais desse tipo com tamanhos muito grandes. Um ponto aqui é provavelmente único em sistemas operacionais mínimos e merece que sejam mais detalhados. Um sistema padrão possui seus executáveis em um determinado lugar e estes serão utilizados toda vez que o sistema for inicializado. Sistemas embarcados que executam na forma de discos de RAM (*Random Access Memory*) ficam executando uma cópia da imagem então não é necessário preservar esses arquivos se estes não são requeridos para operação geral do sistema. Estes arquivos sem uso podem ser removidos, entre estes arquivos podemos citar além da imagem do *kernel*, *scripts* de inicialização do sistema, módulos que não são utilizados em um SOTR e executáveis que só são usados na inicialização.

Porém reduzir o tamanho do sistema requer muito tempo e habilidade por parte de um projetista de aplicações de tempo real, porque é muito difícil dizer com exatidão o que realmente pode ser descartado e o que realmente é necessário em um sistema mínimo. Felizmente existem algumas regras que podem ser seguidas para reduzir o tamanho de tais sistemas como observação de redundâncias, compressão e redução de tamanho dos executáveis descartando bibliotecas que não são utilizadas.

Nos SOTR escolhidos, para verificação do tamanho teve em consideração a imagem gerada do sistema operacional com os módulos necessários para os controladores de disco, placa de vídeo, rede e demais dispositivos nativos na placa-mãe do computador. Também foram levados em considerações módulos dos aplicativos criados para teste dos sistemas e do *benchmark*. Módulos adicionais como de modem, sistemas de arquivos e de serviços opcionais não foram levados em consideração e não foram incluídos na imagem final.

3.1.2. Modularidade

Um sistema é dito modular caso partes separadas ou unidades que o compõe ou não, possam ser colocadas juntas para formar diferentes combinações mantendo o sistema funcional. O conceito de modularidade pode ser entendido também como sendo múltiplos programas que executam ao mesmo tempo como processos isolados e que cada processo executa em um espaço isolado sem influência direta do sistema como um todo.

A modularidade do sistema operacional permite que serviços do *kernel* executem de tal maneira que estes não influenciem em outros serviços, aplicações de tempo real ou dispositivos. A forma de isolamento das aplicações de tempo real depende da técnica de cada fabricante em poder manter essa característica, geralmente são carregados os módulos na área de modo usuário do sistema fazendo com que dessa forma proteja o restante da aplicação.

Em cada aplicação, o *kernel* do sistema operacional mantém uma trilha de cada contador do programa, *flags* da aplicação, registradores e estado para que caso seja necessário suspender a aplicação, esta possa retornar do ponto em que parou de forma dinâmica e transparente ao desenvolvedor.

Alguns autores como Ding *et al.* [15] classificam os sistemas altamente modularizáveis como Sistemas Operacionais de Tempo Real Orientados a Objetos, em que têm como principais características fundamentadas em três pontos: a primeira é que um sistema operacional básico de tamanho mínimo deve ser estabelecido; a segunda é aumentar o compartilhamento das funções do sistema operacional no nível de aplicação, fornecendo uma construção de aplicações altamente parametrizável; e por último é um suporte explícito de configurações em tempo de execução do sistema operacional ou de funções de aplicações de tempo real para facilitar a adaptação de tais aplicações a uma gama muito maior de variações do *hardware* ou de ambientes em que estes estejam inseridos.

Isso de certa forma é bem interessante, pois sistemas operacionais orientados a objetos são extensíveis em novas abstrações e funcionalidades podem ser adicionadas facilmente e eficientemente de tal forma que a estrutura interna do *kernel* do sistema se mantenha uniforme. É útil porque isso permite a implementação, de um domínio ou máquinas alvos, recursos específicos enquanto preserva a interface do *kernel* para programas que já existam.

No presente trabalho, foram verificadas as formas de isolamento criado pelos sistemas operacionais para aplicações de tempo real, o processo de gerenciamento de memória e proteção de dispositivos. Também se observou a modificação dos sistemas operacionais para diferentes plataformas e como é feita essa adaptação.

3.1.3. Adaptabilidade

Chamamos de Adaptabilidade, no presente trabalho, a capacidade de um sistema operacional se adaptar de forma a ter sucesso em uma nova ou diferente situação. Sistemas complexos podem passar por um vasto número de possíveis mudanças em suas condições de operações provenientes do ambiente de funcionamento. Como resultado, SOTR e aplicações que são hospedadas nestes devem ser escritas para funcionar em certas condições que sejam incertas, incluindo que tais sistemas possam se adaptar em performance e funcionalidade a mudanças oriundas do ambiente externo.

A pesquisa no contexto de adaptabilidade de sistemas de tempo real [8], [53], [55], diferencia dois tipos de adaptações em tais sistemas operacionais: aqueles que antecipam mudanças no ambiente operacional, conhecidos as vezes pelo termo **adaptação preventiva**; e aqueles que recaem com mudanças inesperadas, que também podem ser conhecidas como **adaptações reativas**. Adaptações preventivas tentam garantir certo nível de funcionalidade em suas aplicações prevendo hipóteses do comportamento futuro do sistema baseado em comportamentos passados e adaptações reativas por outro lado atuam em resposta a eventos externos ou excepcionais como falhas, carga temporariamente excessiva do processador, entre outras.

Além de verificar se os SOTR em estudo envolvem em uma das duas categorias citadas o presente trabalho também verifica como isso é feito, também estende o conceito para preventiva parcial e reativa parcial, e qual a estratégia gerada pelo sistema operacional para de adaptar em reações adversas.

Exemplo disso é tentar utilizar um módulo que não esteja nativo no sistema. Se uma aplicação que necessite de tal módulo é executada deve observar se esta influência no sistema como um todo. Além de que verifica se o código fonte de partes específicas do sistema operacional é aberto e possibilita a sua alteração. Com a possibilidade de alterar o código fonte, o projetista de tempo real, simplifica em alguns casos, o desenvolvimento de algum controlador de dispositivo e diagnóstico de problemas mais facilmente.

Considera-se também que o estudo do código fonte do sistema operacional permite que o desenvolvedor de um componente de *hardware* existente possa se beneficiar na portabilidade para um *hardware* novo. Modificações no código fonte do sistema permite uma forma eficiente de redistribuir aplicações em cenários não comerciais, como meios acadêmicos e de projetos de pesquisa, facilitando dessa forma a inovação entre os desenvolvedores.

3.1.4. Previsibilidade

A possibilidade de um sistema operacional ser previsível influencia diretamente na classificação dele ser considerado de tempo real ou não. Essa característica mostra quão previsível é um sistema operacional para que este possa, de forma eficiente, atender as restrições temporais, em outras palavras podemos dizer que um sistema operacional previsível é um sistema de tempo real que engloba tarefas do tipo *hard* em que qualquer que seja o processamento. É garantido que a sua execução completa em um específico intervalo de tempo.

O comportamento do sistema deve também possibilitar aos projetistas de tempo real informações detalhadas sobre as interrupções do sistema, chamadas de sistemas (*systems calls*) e temporizadores/relógios. Qualquer SOTR utiliza o recurso de interrupções para assegurar que este irá responder rapidamente a eventos oriundos do ambiente. Esta forma assíncrona de resposta a eventos é bem utilizada na maioria dos sistemas operacionais que são preemptivos pois, assim, aplicações que fazem parte de equipamentos médicos, controles industriais ou dispositivos automobilísticos podem ter garantias de que serão atendidos a quaisquer alterações que venham sofrer.

O profissional de tempo real sabendo dessas questões temporais poderá ter uma idéia do tempo máximo em que as interrupções são geradas e o tempo em que o controlador de dispositivo usa para processar uma determinada interrupção. O tempo em cada *system call* deve ser bem conhecido também e independente do número de objetos que possa existir e estejam executando ao mesmo tempo no sistema. As funcionalidades de tempo real oferecidas em sistemas como o POSIX ou qualquer outro padrão recai na principal função dos SOTR que é a de gerenciar os recursos do sistema de tal maneira que a corretude temporal seja sempre bem estabelecida.

Enquanto a maioria dos SOPG asseguram a eficiência e a justiça na alocação dos recursos no meio de vários aplicativos e programas, os SOTR devem seguir fielmente a

política de escalonamento definida e o comportamento temporal da aplicação, fazendo com que o tempo de execução das aplicações possa ser concluído e não tenha prejuízo, tanto para aplicações críticas como para não críticas.

Previsibilidade também implica que as métricas de análise de sistemas de tempo real sejam diferentes e mais minuciosas do que aqueles que não são. Um exemplo que podemos citar é o caso do sistema operacional *Linux*, enquanto sua versão para computadores pessoais é focada na performance média de todas as primitivas do sistema, sua versão de tempo real, *RTLinux*, fornece um nível aceitável para execução de aplicações de tempo real, devido a modificações das políticas de escalonamento e a possibilidade de modificação do comportamento das primitivas do sistema operacional.

Ao mesmo tempo em que a previsibilidade, em SOTR, é desejada nota-se que não é algo que possa ser facilmente medida, em parte por causa de correções de tempo da aplicação de tempo real que podem diferenciar dependendo do escalonamento criado.

As definições de previsibilidade de um sistema operacional devem ser centradas em alguns pontos como rigor dos *deadlines* das tarefas e granularidade do sistema – determinando assim o quanto uma tarefa pode se estender e a quantidade de tempo necessária para o escalonador tomar decisões, sendo que sistemas que possuem alta granularidade (o relógio do sistema possui uma resolução alta) possuem escalonadores mais adequados a aplicações de tempo real. Essas características podem variar de acordo com o sistema operacional em escolha.

Para análise de previsibilidade, no presente trabalho, foi utilizada uma aplicação que exercita os sistemas operacionais (*benchmark*) baseados em diferentes parâmetros e requisitos, usando principalmente para verificar a granularidade do sistema operacional e o tempo necessário para executar determinadas funções. A explicação sobre o uso, modelagem e descrição do *benchmark* serão descritas no Capítulo 4.

3.2. RTLINUX 3.2 RC1

O SOTR, *RTLinux* surgiu da idéia de usar o sistema UNIX como base para ser utilizado por aplicações de tempo real. O projeto foi inicialmente proposto por Yodaiken e logo depois implementado por Barabanov em seu projeto de Mestrado no Instituto de Tecnologia do Novo México, Estados Unidos. O princípio era criar um SOTR que fosse aberto para modificações, eficiente e que suportasse multitarefa para aplicações de tempo real do tipo *hard*.

3.2.1. Avaliação

A escolha do *Linux* se deu porque a licença de uso não era restrita como do UNIX, o seu código fonte se encontra disponível para alterações e em muitas situações, o sistema se mostrou eficiente e estável [4]. Também podemos citar que disponibilizar o código fonte dos componentes que formam o sistema operacional se mostra útil principalmente para a verificação, análise e correção de problemas do sistema.

O sistema *Linux* também pode ser executado na maioria dos computadores disponíveis, além de ser também utilizado em plataformas embarcadas usando poucos recursos de *hardware*.

Outras características do *Linux*, que possibilitou sua integração com recursos de tempo real, é que ele possui um amplo ambiente de desenvolvimento e uma variedade muito grande de aplicações abertas. Todos esses recursos citados fazem com o *Linux* seja uma plataforma de desenvolvimento muito interessante para uma grande variedade de aplicações de sistemas de tempo real.

No entanto o *Linux* tem muitos problemas que o incapacitam para ser usado como SOTR para aplicações do tipo *hard*, um dos mais importantes é o fato de que as interrupções são desabilitadas sempre que alguma aplicação utilize o modo kernel. Isso trás segurança para as estruturas internas do *kernel*, impedindo que possam ser alteradas de alguma forma, porém se retira a previsibilidade do sistema já que gera tempo não programado inicialmente para aplicações de tempo real.

Alguns outros recursos que encontramos em tais sistemas e que retiram previsibilidade de aplicações de tempo real é o uso de memória virtual, alta granularidade nos temporizadores do sistema e escalonador que tem como princípio repartir igualitariamente o uso do processador entre os processos.

O *RTLinux* é uma variante do *Linux* para execução de processos de tempo real do tipo *hard*. O sistema possibilita a execução de processos especiais e tratar interrupções de tempo real na mesma máquina, como um sistema *Linux* padrão. O sistema operacional é um *microkernel* de tempo real que trabalha junto com o *kernel* do *Linux* (ver Figura 11), para alguns autores esse esquema também é conhecido como *Dual kernel* [60]. Atualmente o SOTR possui duas vertentes: uma aberta para desenvolvimento à comunidade mundial, que é a escolhida no presente estudo, o *RTLinux/Open* e a versão comercial que possui várias outras funcionalidades que a versão aberta não possui, o *RTLinux/Pro*.

De acordo com Farines *et al.* [17], inicialmente todas as interrupções são tratadas pelo *microkernel*, para interceptar requisições de tempo real. Caso não existam tais requisições o *kernel Linux* volta a tratar as interrupções. O que ocorre é que o sistema operacional adiciona uma camada virtual entre o *kernel* padrão do *Linux* e o *hardware* do computador. Essa nova camada aparece como sendo o *hardware* do computador para o núcleo padrão do *Linux*. A camada do *RTLinux* não sofre interferência funcional do núcleo padrão, fazendo com que aplicações comuns possam ser utilizadas sem problemas no sistema.

A camada de abstração de *hardware* (HAL, *Hardware Abstraction Layer*) trabalha capturando todas as interrupções de controle de máquina geradas pelo *kernel* padrão do *Linux*. Isso é feito modificando o código fonte do sistema operacional padrão e substituindo todas as instruções de máquina `cli`, `sti` e `iret` que habilitam e desabilitam interrupções por instruções de *software* `S_CLI`, `S_STI` e `S_IRET`.

Quando uma interrupção desse tipo ocorre, a HAL verifica a variável. Se elas estiverem inicializadas para tempo real, ela passa para o controle do despachante (*dispatcher*) do *RTLinux* imediatamente, caso contrário passará para o controle do despachante do sistema operacional padrão.

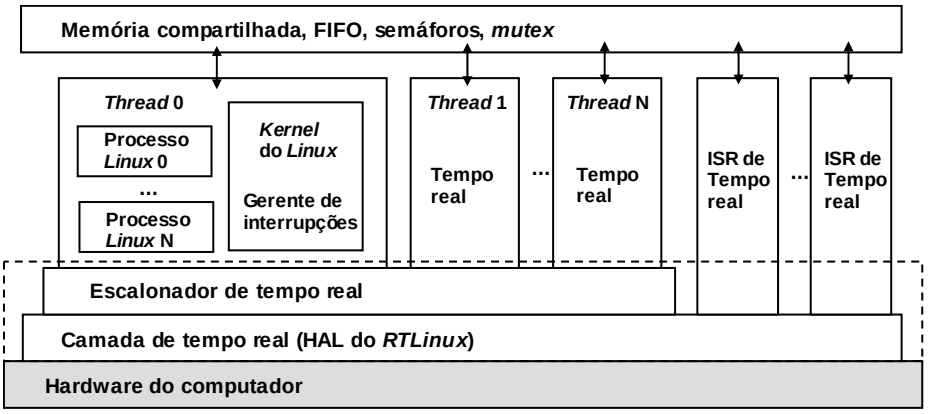


Figura 11 - Estrutura principal do *RTLinux* para sistemas de tempo real [20]

Um dos aspectos mais importantes do *RTLinux* é o modo em que ele é executado. A HAL é carregada em uma versão qualquer do *Linux* executando como módulo desse sistema. Quando o código do SOTR é carregado todas as funcionalidades para tempo real ficam prontas, mas ainda mantém o *kernel* do *Linux* executando como se fosse uma tarefa ociosa, ou seja, quando não há tarefas de tempo real executando o sistema operacional padrão executa.

Para entender a implementação do *RTLinux* e como os programas são executados nele, temos de entender os conceitos de módulos. Brevemente falando, módulos do *Linux*, são construídos pela compilação de seus códigos fontes em arquivos objetos. Os arquivos objetos são carregados no *kernel* onde eles estão. Quando o módulo é inicialmente carregado a função `init_module` () executa, apenas uma vez e quando todas as configurações para o código é feita.

Se a função `init_module` () estiver vazia o módulo irá carregar mas não fará nada. Todos os programas do SOTR são carregados dinamicamente, ou seja, quando o sistema está em execução, porém nada impede que os módulos possam ser adicionados diretamente durante a inicialização do sistema, contanto que tenha os módulos do HAL carregados inicialmente.

Tamanho

Para medir o tamanho da imagem gerada no *RTLinux* hospedado em um sistema *Linux* com *kernel* versão 2.4.30, observou-se o tamanho da imagem em dois momentos com e sem a inclusão da camada do SOTR. Este ficou com um tamanho de 1.2 MB sem a camada de tempo real, levando em consideração que a imagem final incluia os módulos básicos para inicialização do sistema operacional.

Após a geração da camada de tempo real, a imagem ficou com um tamanho aproximado de 1.7 MB, observando que a maioria das funcionalidades de tempo real do *RTLinux* foram habilitadas. Podemos concluir que a imagem do *microkernel* possui um tamanho aproximado de 500 KB. Também temos de levar em consideração os módulos necessários para que aplicações de tempo real sejam executadas no sistema operacional, observe a Tabela 1.

Tabela 1: Tamanho dos módulos do *RTLinux* para suporte de aplicações de tempo real

Módulo	Tamanho
mbuff.o	472 KB
rtl.o	236 KB
rtl_fifo.o	180 KB
rtl_posixio.o	180 KB
rtl_sched.o	1.4 MB
rtl_time.o	332 KB

Somando todos os módulos descritos na Tabela 1, encontramos aproximadamente 2.8 MB, que juntando aos 500 KB da camada de tempo real, temos 3.3 MB aproximado para o SOTR.

Modularidade

O SOTR *RTLinux* é fundamentado em um único bloco que compõe o seu sistema central, neste estão o *microkernel* que coexiste com o *kernel* do *Linux* hospedeiro. A unidade principal do sistema *RTLinux* é a *thread*. Processos de tempo real podem ter uma ou mais *threads* sendo que estas podem ou não ser comunicantes entre si. Como forma de compatibilidade o SOTR é implementado seguindo as definições dos padrões POSIX 1003.13 para mínimos recursos para sistemas operacionais de tempo real. O projeto interno foi direcionado para obedecer estes requisitos.

Quanto a tarefas de tempo real, cada unidade básica executa em uma instância modular diretamente com o *kernel* do sistema operacional, ou seja, esta é carregada diretamente no espaço do *kernel*. Um módulo do *Linux* nada mais é que um arquivo objeto, geralmente escrito na linguagem C e usualmente criado com um argumento que informe modularidade por parte do compilador, no caso do compilador gcc[6], para informar que um arquivo fonte deve ser compilado como módulo deve-se usar a opção –c na linha de comando.

Porém ao contrário dos arquivos fontes comumente escritos com a função `main()` responsável pela funcionalidade principal, um modulo do *kernel* possui duas funções principais a `int init_module()` e a `void cleanup_module()` em que a primeira é responsável pelo carregamento do módulo no sistema, que retorna 0 (zero) caso obtenha sucesso no carregamento ou um valor negativo em caso de falha e a segunda função é chamada para desalojar o módulo na memória.

Não existe um limite exato para o número de *threads* que fiquem executando no sistema operacional, porém esta é limitada pela quantidade de memória disponível no sistema, porém o custo de escalonamento é proporcional ao número de tais unidades. Um tamanho ideal de escalonamento que não implica muito *overhead* no sistema é em torno de 15 *threads* [17]. Por padrão o sistema aceita no máximo 128 *threads*, sendo que este número pode ser modificado durante a compilação na função `RTL_PTHREAD_THREADS_MAX`. Como conseqüência de utilizar *threads* em mesma área que o *kernel* implica no que foi dito anteriormente, na Seção 2.2.3 do Capítulo 2, em que caso ocorra uma falha no módulo este pode danificar o sistema como um todo

[6] GCC – É um compilador distribuído livremente para comunidades de desenvolvimento, mais informações podem ser encontradas em http://www.gnu.org

indisponibilizando recursos para outras tarefas. Outra impossibilidade de proteção no sistema operacional é no que diz respeito a dispositivos, como cada dispositivo é alocado em um espaço compartilhado junto com o *kernel*, caso ocorra falha de algum destes o sistema também sofrerá influência dessa falha.

O SOTR em questão caso necessite de algum novo módulo para controlar um novo dispositivo incluído no sistema tem a possibilidade do mesmo ser carregado dinamicamente enquanto o sistema esteja em funcionamento. Por outro lado a modularidade do *microkernel* de tempo real não oferece os mesmos benefícios, sendo que cada módulo que é nativo no sistema operacional e que não esteja habilitado, este deve ser compilado, parar os módulos de tempo real do sistema operacional e então carregado com os mesmos. Caso alguma aplicação de tempo real esteja sendo usada no sistema operacional, esta deve ser interrompida para que a nova funcionalidade possa ser acoplada.

No teste feito foi gerada uma imagem do SOTR em que não possuía o modulo de controle de filas de tempo real, `rtl_fifo.o`, para adicionar tal funcionalidade, uma nova imagem foi gerada, desalojado os módulos que estavam em execução no sistema e então acrescentado o novo recurso junto com os outros módulos que foram retirados. Isso mostra pouca modularidade por parte do sistema operacional. Tal problema não é encontrado na versão comercial do SOTR.

Para criar uma *thread* de tempo real no SOTR deve-se usar a função `pthread_create()`, que deve ser utilizada somente dentro da função `init_module()`. Para cancelar uma *thread* deve-se utilizar a função `pthread_cancel()` dentro da função `cleanup_module()` para que a mesma seja desalojada da memória.

Quanto ao gerenciamento de memória, é um ponto crítico no *RTLinux.* Como o sistema não fornece alocação dinâmica de memória e nem usa internamente no sistema operacional, tudo fica a cargo do sistema hospedeiro em gerenciar, ou seja, o *Linux*. O principal argumento pela não inclusão de tal recurso pelos desenvolvedores é que alocação de memória não é previsível, e para contornar tal situação e garantir a previsibilidade do sistema, geralmente pré-aloca a maioria dos recursos que as *threads* irão utilizar em tempo de execução, ou seja, antes da *thread* ser carregada na memória o sistema faz uma alocação prévia em um espaço já definido da memória principal para os recursos que serão utilizados. Com isso é possível alocar toda a memória que cada *thread* irá requisitar antes das mesmas serem criadas.

Uma solução para a alocação dinâmica de memória, que inclusive está disponível na versão em estudo do SOTR, é o algoritmo TLSF (*Two Level Segregated Fit*) proposto por [68] em que tenta resolver o problema da previsibilidade da alocação de memória. O algoritmo usa um mecanismo de alocação segregada para implementar uma boa política de alocação. A idéia é utilizar um *array* de listas livres em que cada unidade do *array* mantém blocos livres com tamanhos de classes predefinidas.

Para acelerar o acesso a estes blocos livres e também gerenciar um grande conjunto de listas segregadas, os *arrays* têm de ser organizados em dois níveis. O primeiro nível divide os blocos livres em classes que são separadas por tamanho (8 KB, 16 KB, 32 KB, etc) e o segundo nível subdivide cada primeiro nível linearmente, onde o número de divisões é um parâmetro configurável pelo projetista da aplicação.

Quando definido os valores o sistema operacional poderá definir qual o tempo máximo que cada estrutura pode ser acessada conforme o tamanho do bloco. Esse novo alocador de memória foi projetado para fornecer uma funcionalidade temporal às estruturas `malloc` e `free`, além de fornecer um eficiente gerenciamento de memória, deixando esta pouco fragmentada.

As seguintes funções e bibliotecas são adicionadas a API do SOTR para uso do TLSF:

```
1 void *rtl_malloc(tamanho)
2 void rtl_free(ponteiro)
3 void *realloc (ponteiro, novo_tamanho)
4 void *calloc (tamanho_t nelem, tamanho_t tamanho_elem)
```

A função definida na linha 1 aloca um espaço na memória com tamanho especificado e retorna um ponteiro para iniciar a alocação do bloco de memória, caso ocorra falha a função passa NULL, a função da linha 2 é a função definida para desalojar um bloco de memória indicado pelo ponteiro, na linha 3, a função modifica o tamanho de um bloco que esteja sendo indicado pelo ponteiro para o novo tamanho e na linha 4 é indicada a função que aloca em um determinado bloco de memória `n` elementos da matriz de tamanho `tamanho_t` e que cada elemento possui um tamanho definido por `tamanho_elem`.

Esta forma de gerenciamento de memória é instável e experimental, em testes realizados no SOTR, após o carregamento do módulo de tempo real na memória, o

sistema falhava e não obtinha resposta de outros serviços, foram considerados os valores padrões definidos no momento da compilação do *microkernel*.

Ainda sobre o gerenciamento de memória podemos dizer que o SOTR *RTLinux* foi projetado para ser executado em processadores com suporte a MMU, não tendo nenhuma proteção específica de memória entre as *threads* de tempo real e o *kernel* e também entre as próprias *threads*. Quem fica encarregado pelo controle geral de toda a alocação de memória é o *kernel* padrão do sistema *Linux*.

No caso de memória compartilhada, o dispositivo incluído no sistema operacional é o *mbuff*, em que funciona como mecanismo de alocação de blocos de memória compartilhada que conecta estes blocos e eventualmente desaloja estes também. Para utilizar tal recurso as funções devem ser descritas conforme a sintaxe:

```
1 void * mbuff_alloc(const char *nome, int tamanho)
2 void mbuff_free(const char *nome, void * mbuff)
```

A primeira vez que a função `mbuff_alloc()` é chamada, um bloco de memória compartilhada de tamanho especificado é alocado e um contador de referência para este bloco é inicializado com o valor 1. Em caso de sucesso na alocação, o ponteiro para o novo bloco alocado é retornado, sendo que caso ocorra uma falha na alocação o valor retornado é NULL, caso um bloco alocado com o mesmo nome especificado na função exista, a função retorna um ponteiro que pode ser usado para acessar este bloco já criado e modificar o contador de referência.

Adaptabilidade

Quanto a adaptabilidade, o SOTR *RTLinux*, pode ser categorizado como de adaptação reativa, porém com limitações. Isso é devido ao fato que durante os testes foram modificados alguns dispositivos de *hardware* do sistema, e podemos observar que caso o módulo para tal dispositivo não esteja pré-compilado junto com a imagem do sistema este ficará inativo. Deixar a imagem do SOTR menor possível garante que as estruturas de dados internas do *kernel* sofrerão menos alterações e modificações externas. Por outro lado, a não inclusão de uma maior variedade de módulos pode não satisfazer o sistema em determinadas situações.

Quanto ao número de plataformas de *hardware* suportadas, o sistema operacional mantém um número limitado de processadores, podendo funcionar em x86, *Power* PC e ARM (incluindo *Strong*ARM). O suporte a sistemas multiprocessados (SMP) apenas na plataforma x86.

Outra parte integrante do pacote do sistema operacional são os códigos fonte de todos os componentes, sendo que estes se encontram bem documentados e possui uma variedade muito grande de exemplos com as principais funcionalidades do sistema, como barreiras, sinais, temporizadores, controles de dispositivos de som, entre outros.

Uma função importante do SOTR e que vem desabilitada por padrão no momento da compilação da imagem, é o *debug*, que funciona como auxílio aos desenvolvedores de encontrar e soluções de erros. Acrescentando tal funcionalidade ao sistema, se observa que a imagem final sofre um aumento de tamanho já que mais um módulo no sistema operacional é incluído.

Quanto à possibilidade de modificação do escalonador a empresa responsável pelo SOTR possibilita que o escalonador padrão do sistema seja modificado. Para isso existe um arquivo fonte responsável por tal função, chamado `rtl_sched.c` e que a parte responsável pelas decisões de escalonamento é tomada pela função `rtl_schedule()`, logo modificando tal função é possível mudar a política de escalonamento do sistema.

Quanto aos dispositivos do sistema o SOTR define dois tipos de interrupções que são necessárias para controlá-los: As interrupções do tipo *hard* e as interrupções do tipo *soft*. A primeira interrupção é originada diretamente pelo *hardware*, sendo que existe apenas uma pequena interferência nos serviços oferecidos por este tipo de interrupção. Como este tipo de interrupção é executada diretamente na camada do *RTLinux* não é possível utilizá-las em serviços do sistema operacional padrão. As funções responsáveis para a manipulação de tais interrupções é a `rtl_request_irq()` – na qual o sistema aciona e a `rtl_free_irq()` – que libera uma interrupção gerada. Já o segundo tipo de interrupções, a do tipo *soft*, são normais ao sistema operacional padrão do *Linux* e que são executadas diretamente por *threads* oriundas do mesmo. Não possui interferência significativa da latência já que a mesma é gerada do sistema padrão e que não necessitam de restrições temporais e os manipuladores de tais interrupções são as funções `rtl_get_soft_irq()` – que acionam a interrupção e a `rtl_free_soft_irq()` – que libera a interrupção.

Para testar a adaptabilidade do sistema foi executado um exemplo incluído no pacote do SOTR sem que o módulo responsável para o correto funcionamento fosse adicionado a imagem do *kernel*. Após a inclusão do mesmo no sistema este se tornou instável e parou de responder, informando uma mensagem de erro ao usuário sobre o problema, porém indisponibilizando o sistema operacional como um todo.

Previsibilidade

Para análise de previsibilidade foi executado o *benchmark* criado para tal funcionalidade além de observar outros fatores como gerenciamento de processos, resolução dos temporizadores/relógios, escalonamento e comunicação entre processos (IPC, *Inter Process Communication*).

Quanto ao escalonamento o SOTR possui um escalonador dirigido a prioridades, obedecendo à função de escalonamento:

```
int pthread_setschedparam(pthread_t thread, int politica, const
struct sched_param *parametro)
```

O argumento, `int politica`, da função acima, é quem define o tipo de política utilizada pelo SOTR. Três destas encontram-se disponíveis aos desenvolvedores de sistemas de tempo real. A primeira SCHED_FIFO em que o escalonamento é baseado em prioridades fixas e em caso de alguma *thread* possuir a mesma prioridade estas serão escalonadas em ordem de chegada. A segunda é a SCHED_SPORADIC que define a execução com fundamento em servidores esporádicos que são usados para executar *threads* aperiódicas. E por última SCHED_OTHER que é uma definição aberta ao desenvolvedor, um porte a prioridades dinâmicas baseada no EDF está em fase de testes e validação e que apesar de possuir restrições de uso já encontra disponível ao desenvolvedor para uso.

A estrutura `sched_param` contém os valores de prioridade para as *threads*, quanto maior o valor, maior será a prioridade, sendo que estas são definidas pelas funções a seguir:

```
int sched_get_priority_max(int politica)
int sched_get_priority_min(int politica)
```

A primeira função, retorna o limite máximo de prioridade para a política de escalonamento e a segunda função, faz justamente o inverso, retorna o limite mínimo de prioridade. O número máximo e mínimo de prioridade do SOTR é 1000000 e 0 respectivamente.

No caso de tratamento de *threads* periódicas a API do SOTR tem disponível a seguinte chamada de sistema:

```
int pthread_make_periodic_np(pthread_t thread, const struct
itimerspec *its)
struct itimerspec {
   struct timespec it_interval /* define o tempo do período */
   struct timespec it_value}     /* define o tempo da primeira
ativação */
```

Esta chamada marca uma determinada *thread* como sendo periódica. O tempo é especificado pela estrutura `itimerspec`. Caso necessite suspender a execução de uma *thread* chamada pela função acima, até um tempo especificado deve-se usar a função `int pthread_wait_np(void).`

Quanto a relógios e temporizadores o sistema *RTLinux* possui uma interface compatível com o padrão POSIX porém com funções não portáveis, ou seja, é possível que existam *threads* que executem com *RTLinux* usando a API POSIX nativa. Para reduzir o tempo de desenvolvimento foram modificando as mesmas para que possibilitem os requisitos de tempo real sem necessidade de criação de novas funções.

Observe a chamada de função a seguir:

```
timer_create(identificacao_relogio, struct sigevent *tipo,
timer_t *identificacao)
```

A função anterior cria um temporizador fundamentado no relógio do sistema. Geralmente na API do *RTLinux* usa-se o relógio chamado CLOCK_REALTIME que existe em todos os padrões POSIX.4. Outros tipos são encontrados no SOTR em questão, entre eles CLOCK_MONOTONIC, no qual executa em uma taxa fixa e nunca é ajustado ou reiniciado, CLOCK_RTL_SCHED, que é o relógio que o escalonador usa para o escalonamento de tarefas. Embora possam existir outros tipos de relógios, estes

variam de acordo com a arquitetura da máquina, um exemplo disso é o relógio CLOCK_APIC que é usado em máquinas multiprocessadas do padrão x86.

O argumento `tipo` caso seja não nulo, aponta para a estrutura `sigevent`, na qual é alocada pela aplicação para definir notificações assíncronas da ocorrência de um sinal ou de sinalização quando um temporizador irá terminar. Caso o argumento seja nulo um sinal padrão é gerado pelo sistema operacional.

A estrutura `sigevent` contém três membros conforme definido:

```
1 struct sigevent {
2          int sigev_notificacao
3          int sigev_numsinal
4          union sigval_value}
```

Sendo que a linha 2 indica uma *flag* que especifica que tipo de notificação deve ser usada em caso de término do temporizador sendo que pode ser um sinal, aguardar ou outro evento. Apenas dois valores são definidos para `sigev_notificacao`: SIGEV_SIGNAL, que é para enviar um sinal descrito pelo restante da estrutura `sigevent` e SIGEV_NONE que envia uma notificação após o término do temporizador. A linha 3 define o número do sinal, nesse caso a lista é definida seguindo o padrão POSIX [46]. Por último a linha 4 é o parâmetro que indica aonde o sistema armazenou o identificador do temporizador criado.

A forma de manipular temporizadores seguindo esse tipo de especificação possibilita diminuir o tempo de desenvolvimento de certas estruturas por parte do desenvolvedor de tempo real, já que não necessita aprender a utilizar uma nova API de desenvolvimento para criação das aplicações de tempo real.

Dois tipos de temporizadores são utilizados para esses tipos de aplicações: o do tipo ONE-SHOT que é um temporizador que é armado com um valor de término inicial, que seja relativo ao tempo corrente em que foi criado ou absoluto, baseado em algum tempo base, podendo ser descritos em segundo ou em nanosegundos. O temporizado desse tipo finaliza quando este é desarmado.

O outro tipo de temporizador é o PERIODIC, em que desarma um temporizador em períodos de tempo relativos ou absolutos. Quando o tempo inicial definido para finalizar chega, o temporizador é recarregado funcionando como uma estrutura de *watchdog*.

Quanto a valores de tempo e suas especificações muitas das facilidades funcionais de tratamento desse tipo pelo SOTR em questão é observada conforme a Tabela 2.

Tabela 2: Definições de granularidade do tempo no RTLinux

Tipo	Nome	Descrição (tempo)
time_t	tv_sec	segundos
long	tv_nsec	nanosegundos

A variável com nome `tv_nsec` é válida apenas se maior ou igual a zero e menor que o número de nanosegundos em um segundo (1 bilhão). O intervalo de tempo descrito, em nanosegundos, por essa estrutura é (`tv_sec` * 10^9 + `tv_nsec`). Esses tipos são parte integrante da estrutura de manipulação e gerenciamento do tempo descrito em:

```
1 int clock_gettime(clockid_t clock_id, struct timespec *ts)
2 hrtime_t clock_gethrtime(clockid_t clock)
3 struct timespec {
4   time_t tv_sec
5   long tv_nsec}
```

Com a utilização da estrutura anterior é possível obter a leitura de um relógio que esteja sendo utilizado pelo sistema, onde `clock_id` é o relógio que deve ser lido e `ts` é a estrutura interna do sistema responsável pelo armazenamento do valor obtido. A expressão `hrtime_t` é um valor expresso como um número em nanosegundos com resolução de 64 *bits*, geralmente usado para identificar tempos com resoluções para aplicações do tipo *hard*.

É encontrado no sistema no sistema uma falha quanto na função específica para tratamento de operações com ponto flutuante. Essa função é descrita pela API do próprio sistema como `rt_use_fp(int permite)`, que caso seja um valor zero no parâmetro, este não é permitido o uso e se for não zero é permitido. A função funciona

mudando a execução da tarefa para a camada de tempo real, não permitindo que o sistema *Linux* controle tais operações.

Quanto a comunicação entre tarefas o *RTLinux* possui diversos mecanismos que permitem a comunicações entre tarefas de tempo real e tarefas nativas do *Linux*, podemos destacar *mutex*, memória compartilhada, semáforos e FIFO de tempo real.

A primeira, *mutex*, refere-se ao conceito de permitir apenas uma *thread* por tempo, de ler ou escrever um recurso compartilhado. Sem esse recurso a integridade dos dados encontrados no sistema, mais especificamente nesse recurso compartilhado, poderia ser comprometido. As funções disponibilizadas pela API nativa do *RTLinux* são em conformidade com o padrão POSIX, porém não portáveis:

- `pthread_mutexattr_getpshared()` : obtém o processo compartilhado definido pelo atributo de identificação do *mutex*;
- `pthread_mutexattr_setpshared()` : inicializa o processo compartilhado definido pelo atributo de identificação do *mutex*;
- `pthread_mutexattr_init()` : inicializa um objeto de atribuição de um determinado *mutex*;
- `pthread_mutexattr_destroy()` : destrói um objeto de atribuição de um determinado *mutex*;
- `pthread_mutexattr_settype()` : inicializa um tipo de atribuição para o mutex. Sendo que este pode ser PTHREAD_MUTEX_NORMAL – que é o padrão do sistema ou PTHREAD_MUTEX_SPINLOCK – usado apenas em interfaces de sincronismo em sistemas SMP;
- `pthread_mutexattr_gettype()` : retorna um tipo de objeto de atribuição de um mutex;
- `pthread_mutex_init()` : inicializa um mutex com um atributo de especificação definido em um determinado objeto;
- `pthread_mutex_destroy()` : destrói um *mutex*;
- `pthread_mutex_lock()` : bloqueia um *mutex* desbloqueado. Caso o *mutex* esteja bloqueado, a *thread* que chamou o bloqueio fica no aguardo desta ser liberada;

- `pthread_mutex_trylock()` : tenta bloquear um *mutex*. Caso o *mutex* esteja bloqueado, a *thread* que chamou o bloqueia retorna sua execução sem esperar que esta seja liberada;
- `pthread_mutex_unlock()`: desbloqueia um *mutex*.

Quanto ao compartilhamento de memória essa é fornecida com uma interface, não POSIX, chamada de *mbuff*, inicializada como módulo durante a execução do sistema operacional. Por padrão esta não está disponível no SOTR sendo que o desenvolvedor tem de acrescentar a imagem final para utilização. Esse mecanismo é usado para comunicação entre as *threads* de tempo real e os processos normais do *Linux*. Isso é possível já que ambos os sistemas tanto o nativo quando o de tempo real possuem a mesma API. A descrição da interface já foi definida na parte relacionada a modularidade do presente SOTR.

A forma mais comum de gerenciar o acesso a recursos compartilhados no *RTLinux* é através de semáforos, que segue as seguintes chamadas de funções:

- `sem_wait()` : espera o retorno de um determinado semáforo;
- `sem_timedwait()` : espera o retorno de um determinado semáforo a partir de um determinado tempo;
- `sem_post()` : postagem do semáforo;
- `sem_init()` : inicializa um semáforo;
- `sem_destroy()` : destrói um determinado semáforo.

O mais importante no uso de tais estruturas é observar se estas não irão bloquear *threads* por um tempo indeterminado, para isso o sistema fica monitorando o uso de semáforos para que estes não possam interferir no andamento da execução das *threads*.

FIFO de tempo real são filas que podem ser lidas e escritas por processos do *Linux* e/ou *threads* do *RTLinux*. Estas são unidirecionais, ou seja, a leitura obedece a uma direção por vez, ou pelo lado do *RTLinux* ou pelo lado do *Linux*, caso seja necessária uma comunicação bidirecional deve-se utilizar um par de tais estruturas de comunicação. Fazendo isso o tempo de comunicação fica com um controle melhor, já que não é necessário criar uma nova estrutura e definir períodos determinados de utilização da mesma também possibilita enxugar o tamanho de sistema evitando novas

funções. Os FIFO´s de tempo real são definidos como entradas de dispositivos do *Linux*, `/dev/rtf0` até um número Maximo definido pelo desenvolvedor do sistema durante a compilação. Antes de usar tais recursos o desenvolvedor deve inicializar seguindo as seguintes chamadas de sistema:

```
1 int rtf_create(unsigned int fifo, int tamanho)
2 int rtf_destroy(unsigned int fifo)
```

A linha 1 define a alocação do buffer de um tamanho específico para a FIFO, o argumento fifo corresponde ao menor número do dispositivo. Já a função definida pela linha 2 desaloja uma FIFO. Após a criação do FIFO. Este pode ser manipulado de acordo com as funções definidas pelo POSIX[7]. Outras estruturas de IPC seriam interessantes no SOTR tais como troca de mensagens ou caixas de mensagens, operações atômicas ou chamadas remotas para redes, sendo que podem ser incluídas as IPC do POSIX mas não tem garantias que estas estruturas possam ser úteis a aplicações de tempo real no que diz respeito a previsibilidade do sistema.

3.3. RTAI 3.3

RTAI (*Real-time Application Interface*) é um projeto que surgiu como sendo uma variante do *RTLinux*, pelo *Dipartimento di Ingeneria Aerospaziale* da *Politecnico di Milano*. Tem sua base ainda o uso de uma interface entre o *hardware* e o *kernel* do *Linux*, porém possuí algumas partes complementares para suprir algumas dificuldades encontradas no *RTLinux*.

3.3.1. Avaliação

Analisando o sistema operacional, observa-se que foram definidas unidades complementares. Com isso abrange mais o uso do *Linux* como ambiente de tempo real em computadores multiprocessados e um maior suporte a aplicações usadas em modo usuário. Esta composição é definida por cinco unidades, descritas cada uma a seguir [50]:

---

[7] As funções para manipulação de FIFO pelo POSIX são: `open()`, `read()`, `write()` e `close()`, não sendo permitido o uso de outras funções da biblioteca STDIO.H.

- A camada de abstração do *hardware* - HAL: na qual prover uma interface para o *hardware* acima do sistema operacional *Linux* e aonde o núcleo do sistema pode ser executado;
- A camada de compatibilidade do *Linux*: fornece uma interface para o sistema operacional *Linux* no qual cada processo do RTAI pode ser integrado no gerenciador de processos do sistema operacional padrão, sem que este modifique nada;
- Núcleo SOTR: oferece as funcionalidades de tempo real para o escalonador de tarefas, processa interrupções e permite bloqueio (*locking*) de processos;
- LX/RT (*Linux Real-time*): esse componente faz com que características de tempo real críticas e não-críticas estejam disponíveis na camada de usuário do *Linux*. Essa é a principal diferença entre o *RTLinux* e o RTAI;
- Pacotes de funcionalidades extras: O núcleo do RTAI é estendido para usar recursos extras como formas de comunicação entre processos, uso de rede e dispositivos de linha serial.

A arquitetura do RTAI é similar a do *RTLinux*, porém não se limita a alterar o *kernel* padrão apenas adicionando a HAL e capturando as interrupções oriundas deste, o SOTR acrescenta interrupções por *software* chamadas `hard_sti` e `hard_cli`, estas assumem as funcionalidades das interrupções originais do *Linux*, `cli` e `sti` incluindo ao sistemas as estruturas básicas para execução de aplicações de tempo real e implementa as interrupções de controle de processadores. Uma vantagem desse tipo de codificação é que possibilita o *kernel* padrão do *Linux* operar normalmente mudando os ponteiros na estrutura HAL para trabalhar com *spinlocks* para alocação de processos em diversos processadores.

A HAL possui cinco módulos principais em que podem ser adicionadas as funcionalidades de tempo real por demanda, ou seja, sem necessidade de parar o sistema para carregar os módulos ou que os mesmos estejam sempre na memória. Esses módulos são: `rtai_hal.o`, que é o núcleo básico do SOTR; `rtai_sched.o`, na qual disponibiliza os métodos de escalonamento do sistema operacional; `rtai_mups.o`, estrutura na qual possibilita que dois métodos de escalonamento funcionem simultaneamente, tendo diferentes relógios como base; `rtai_lxrt.o`, na qual permite

a execução de processos de tempo real em modo de usuário; e `rtai_smp.o`, que possibilita a utilização aprimorada de recursos de tempo real em sistemas SMP.

Como qualquer módulo do *kernel* do *Linux*, esses módulos podem ser carregados e removidos da memória usando comandos (`insmod` e `rmmod` respectivamente), e eles são independentes, o que não ocorre com todos os módulos do *RTLinux*, ou seja, caso precise comunicar com os processos do *Linux* usando FIFO apenas os módulos `rtai_hal.o` e `rtai_fifos.o` são necessários.

Tamanho

O SOTR RTAI também é hospedado em um sistema *Linux* na versão 2.4.30. Sem a aplicação da camada de tempo real ao *kernel* do *Linux* a imagem do sistema possui 1.2 MB de tamanho, após a aplicação da camada de tempo real a imagem final quase não teve modificações no tamanho aumentando 60 KB e ficando com o tamanho final de 1.26 MB.

O tamanho da imagem de tempo real (HAL) ocupa 60 KB. Isso se deve ao fato de que as funcionalidades de tempo real não são incluídas diretamente a imagem do sistema hospedeiro e sim em uma área separada. Os módulos necessários para a execução de aplicações de tempo real foram quase todos incluídos não sendo habilitados o que estavam em fase experimental.

A Tabela 3 mostra a lista de módulos necessários para o funcionamento do RTAI.

Tabela 3: Tamanho dos módulos do RTAI para suporte de aplicações de tempo real

Módulo	Tamanho
rtai_bits.o	16 KB
rtai_calibrate.o	8 KB
rtai_fifo.o	40 KB
rtai_hal.o	30 KB
rtai_leds.o	4 KB
rtai_lxrt.o	100 KB
rtai_math.o	32 KB
rtai_mbx.o	24 KB
rtai_mq.o	28 KB
rtai_msg.o	36 KB
rtai_netrpc.o	28 KB
rtai_sem.o	36 KB
rtai_shm.o	20 KB
rtai_signal.o	12 KB
rtai_sched.o	4 KB
rtai_smp.o	108 KB
rtai_tasklets.o	16 KB
rtai_tbx.o	12 KB
rtai_usi.o	4 KB
rtai_wd.o	12 KB

A soma de todos os módulos é de 570 KB que junto com a HAL do sistema operacional de 60 KB requer do sistema hospedeiro um espaço de 630 KB, tamanho relativamente pequeno para as funcionalidades de tempo real para aplicações do tipo *hard*.

Modularidade

O RTAI possui uma pequena camada anexada ao sistema hospedeiro *Linux*, LX/RT, este sempre que necessário chama um módulo específico de tempo real, além de possibilitar a execução de tarefas em modo usuário. Uma das características de modularidade do sistema operacional é este possuir estruturas próprias para identificação de unidade de tempo real, ou pela função `rt_task_init()` e esta por sua vez podem incluir *threads* de tempo real ou tarefas dependendo de que forma o desenvolvedor desejar, pelas funções `rt_make_hard_real_time()`, ou `rt_make_soft_real_time()`, ou seguindo o padrão POSIX com a própria definição de *thread* que a API do sistema disponibiliza ao desenvolvedor.

Assim como o *RTLinux* o SOTR RTAI possui compatibilidade com o padrão de desenvolvimento da API POSIX abrangendo o padrão 1003.1b que diz respeito a inclusão de filas (*pqueues*). Quanto a API nativa do sistema operacional este possui uma derivação da API do *RTLinux* na sua versão 1, sendo que mantem compatibilidade com a mesma e cerca de 156 funções incluídas por diversos programadores e que não seguem uma coerência total entre elas. Essas funções muitas vezes possuem incompatibilidade entre as mesmas, em muitos casos encontramos a mesma característica implementada de muitas maneiras distintas e com diferentes chamadas de sistema.

Quanto a serviços de tempo real é possível a execução de tarefas em dois modos distintos, ou em modo *kernel* que assim como o *RTLinux* utiliza as definições de *threads* do POSIX para sua implementação e execução como módulo do sistema operacional ou por modo usuário que é o diferencial do SOTR em comparação a outros sistemas baseados em *Linux* é que este possui uma interface bem definida de execução neste modo. Existem três diferentes implementações para execução de tal recurso no sistema: como serviços LX/RT em processos normais; LX/RT estendido; e mini RTAI LX/RT.

A primeira implementação, LX/RT em processos normais, permite o uso de processos de tempo real no RTAI, não necessariamente sendo uma *thread*. É usado para execução de processos do tipo *hard* ou *soft* e a cada chamada de função que um processo do RTAI executa, esta é direcionada para o serviço LX/RT para sua verificação. O espaço de memória no modo usuário para esses processos possibilita o uso de quaisquer chamadas de sistema do *Linux* com o SOTR, através de memória compartilhada, troca de mensagens e semáforos. Nesta implementação as funções para

identificação de tarefas de tempo real são: rt_task_init(), `rt_sem_init()`, `rt_register()`, `rt_mbx_init()`, `rt_allow_nonroot_hrt()`.

Já na segunda implementação, o LX/RT estendido, o processo que chama a função `rt_make_hard_real_time()` sempre fica com a mais alta prioridade, sendo que o processo é escalonado pelo *kernel* do *Linux*. O uso da função `rt_make_soft_real_time()` possibilita que este processo espere o processo de mais alta prioridade no sistema finalizar para poder executar. Esse tipo de processo não permite a chamada de sistema do *Linux* que possa executar troca de contexto, pois um deadlock pode ocorrer.

O último modelo mini RTAI LX/RT permite a execução de funções em um espaço similar ao do *kernel* do sistema nativo, chamado de *tasklets*, sendo que estes podem ser normais ou temporais. Esse tipo de serviço pode ser bastante útil quando o desenvolvedor tem muitas tarefas, residentes no modo *kernel* e no modo usuário, que devem ser executadas como *soft* ou em *hard*, mas que não precisam de qualquer serviço do escalonador do RTAI que possibilite a estas tarefas serem bloqueadas. Esse modelo deve ser usado sempre que uma tarefa do tipo *hard* esteja disponível no RTAI e o escalonador do mesmo esteja com recursos limitados, fazendo com que a sua execução seja de forma direta e simples. As funções usadas nessas tarefas são: `rt_insert_tasklet()`, `rt_tasklet_exec()` e `rt_remove_tasklet()`.

Assim como o *RTLinux* o RTAI tem o número limitado de tarefas em execução definido pela quantidade de memória disponível no sistema, porém não possui nenhuma configuração no *kernel* do SOTR que delimite a quantidade das mesmas. Nos testes realizados observou-se que o mesmo problema encontrado no *RTLinux* foi notado no SOTR em questão, quanto maior o número de tarefas executando maior é o tempo em questões como troca de contexto ou de latência do sistema. Um número ideal de tarefas em execução do sistema, sem um *overhead* significativo no sistema ficou em torno de 20 processos em modo usuário e de 15 em modo *kernel*.

O esquema de carregamento de módulos dinamicamente no sistema operacional também é utilizado sendo que este é feito de forma mais ajustável e possibilita que módulos que não estejam em execução não fiquem residentes na memória. No caso de algum dispositivo de aplicação de tempo real necessitar de um modulo para sua execução, este carrega diretamente, não precisando de alguma compilação específica.

Por padrão todos os dispositivos do RTAI estão disponíveis após geração da imagem final.

Quanto a identificação de uma tarefa de tempo real deve-se usar a função `rt_task_init (RT_TASK *tarefa, void *rt_thread, int arg, int tamanho_pilha, int prioridade, int uso_fpu, void *sinal)` sempre antes do uso de qualquer função de tempo real, em que `tarefa` é um ponteiro para um estrutura do tipo `RT_TASK` em que o espaço alocado na memória é fornecido pelo sistema operacional, `rt_thread` é o ponto de entrada para alguma função e o parâmetro a seguir, `arg`, é o parâmetro da função, `tamanho_pilha` é o tamanho da pilha usada pela tarefa de tempo real.

Um parâmetro interessante é a possibilidade de usar ponto flutuante sinalizando o sistema pelo parâmetro `uso_fpu` que caso deseje utilizar, o valor indicado deve ser não zero e o parâmetro `sinal` é outra *flag* que identifica se deseja usar sinal durante a operação da tarefa ou não, caso seja 0 (zero) não usa sinais. Outra função também similar a essa é utilizada para sistemas SMP, sendo explicada mais a diante quando for analisado a previsibilidade.

O gerenciamento de memória no sistema operacional tem a possibilidade de utilizar a alocação dinâmica de memória através das funções `rt_malloc()` e `rt_free()`, sendo que ambas não tem garantias de comportamento ideal para aplicações de tempo real do tipo *hard*, sendo que é necessário utilizar a função `mlockall` que possibilita ao processo ficar isolado na memória evitando paginação e falhas de memória. Aplicações do tipo *soft* não necessitam deste tipo de alocação restrita, mas é recomendada a utilização deste tipo de recurso.

Quanto a memória compartilhada a API do SOTR disponibiliza o módulo `rtai_shm.o` com quatro funções: `rtai_malloc()`, `rtai_free()`, `rtai_kmalloc()` e `rtai_kfree()`. As duas primeiras são mais utilizadas em aplicações que executem em modo de usuário e as demais em modo *kernel* ou em aplicativos que estejam acessando estruturas do SOTR.

O tamanho do bloco fica a cargo do sistema hospedeiro por esse motivo que não possui garantias de previsibilidade e de comportamento temporal, porém algumas modificações de tais funções possibilitam que tenham comportamento temporal e são iniciativas isoladas e que ainda não foram incluídas na API do sistema operacional.

Adaptabilidade

A adaptabilidade do SOTR RTAI possui algumas características peculiares. Uma delas é que o sistema dependendo das modificações sofridas pode ser incluído como adaptação reativa ou adaptação preventiva isso porque o sistema inclui uma grande variedade de modificações. Porém isso pode se tornar de certa forma uma falha, já que não mantem uma padronização ao sistema que fica inconsistente e com muitas brechas para eventuais falhas lógicas.

Durante os testes foram modificados alguns dispositivos de *hardware* do sistema, e este teve o mesmo comportamento que o *RTLinux*, que caso precise de algum componente que não esteja pré-compilado junto com a imagem do sistema este ficará inativo. Porém todos os módulos de tempo real já estavam disponíveis e não necessitando uma nova compilação por parte do desenvolvedor.

A imagem do SOTR é bem pequena o que torna possível a inclusão deste em sistemas com baixa memória física disponível. Porém pela grande quantidade de funções as estruturas internas do sistema são inconsistentes e muitas vezes redundantes. Os códigos fonte do sistema operacional e de seus componentes estão disponíveis junto com o pacote do sistema, porém não possui nenhum documento com as principais funcionalidades do sistema operacional ou de explicação das principais API´s.

Possui uma variedade muito grande de exemplos com as principais funcionalidades do sistema tanto usando abordagens no modo usuário quanto em abordagens no modo kernel com códigos bem comentados e com contato dos colaboradores.

O número de plataformas de *hardware* suportadas pelo sistema operacional inclui x86, *Power* PC, ARM (incluindo *Strong*ARM), MIPS e Motorola m68k-nommu. Suporte a SMP apenas na plataforma x86, sendo esta bem estruturada e com um suporte mais abrangente do que o do *RTLinux*.

Nativamente o sistema operacional possui um sistema de *debug* fornecido como módulo. Um módulo adicional que pode ser incluído é o *Linux Trace Toolkit*[8] (LTT) que registra todos os eventos relevantes do sistema e possui uma interface gráfica para verificação dos relatórios (Figura 12).

[8] http://www.opersys.com/LTT/index.html, Acessado no dia 29 de agosto de 2006.

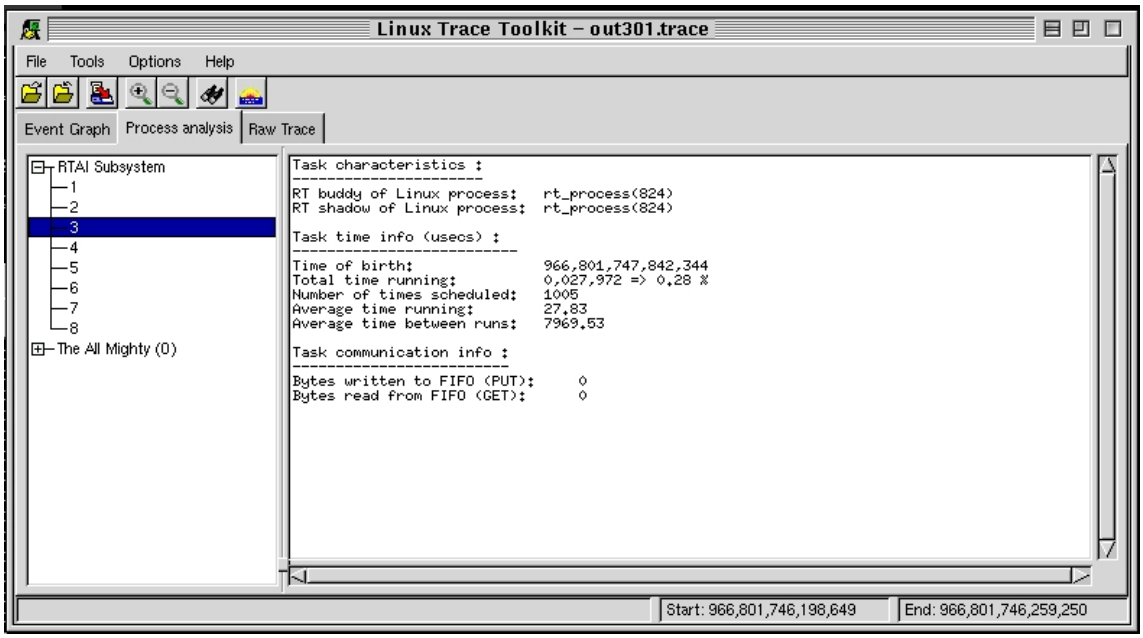


Figura 12 - Análise de tarefas do RTAI gerada pelo LTT

Podemos notar, na Figura 12, o sistema RTAI com oito tarefas sendo executadas e que cada uma com informações detalhadas. Essa ferramenta possibilita ao desenvolvedor informações para análise do comportamento de uma aplicação de tempo real sobre um certo período de tempo, além de possibilitar encontrar falhas lógicas e temporais do sistema. Porém em teste, observou-se que o relatório de análise de tarefas gerado pelo LTT possui um pequeno atraso em comparação a análise e observação direta. Uma forma que mostra com mais precisão as informações sobre o estado de elementos críticos de tempo real, tais como FIFO, interrupções, módulos e gerenciamento de memória é através da interface de sistemas de arquivos `/proc` do sistema hospedeiro. Os arquivos são o `scheduler`, `rtai`, `fifos` e `memory_manager` e estão localizados na sub-pasta `/proc/rtai`.

Quanto a modificações do escalonador, o RTAI possibilita a alteração do escalonador através do arquivo `sched.c`, que é parte integrante do pacote, porém não é tão fácil alterar o escalonador quanto é no *RTLinux*, já que as estruturas são bem aninhadas entre si e a sua modificação se torna muito difícil. As funcionalidades de escalonamento são tomadas pelos módulos ou `rtai_sched.o` ou o `rtai_lxrt.o` sendo o primeiro é mais usado para *threads* ou processos de tempo real do tipo *hard* e o segundo para *thread* ou processos de tempo real executados na mesma área que tarefas do sistema hospedeiro.

Quanto a interrupções do sistema, assim como no *RTLinux* também são definidas dois tipos: a do tipo *hard* e *soft*. As mesmas definições que foram explicadas no *RTLinux* são aplicadas aqui com a diferença que as funções responsáveis para a manipulação de tais interrupções são a `rtl_request_global_irq()` – na qual o sistema aciona uma interrupção deste tipo e a `rtl_free_global_irq()` – que libera uma interrupção gerada.

Também é possível executar os manipuladores de interrupções como processos normais de interrupções do *Linux* através da função `rt_pend_linux_irq()`. Isso é um mecanismo útil, pois este executa seu código em espaço reservado de memória do *Linux* deixando um pouco menor a carga na camada de tempo real do RTAI.

Para testar a adaptabilidade do sistema foi executado um exemplo incluído no pacote do SOTR, como por padrão todos os módulos de tempo real são gerados junto com a imagem do SOTR não corria o risco de que sem o módulo responsável para o correto funcionamento desta determinada aplicação ficasse faltando.

Previsibilidade

Na execução do *benchmark* observaram-se dois comportamentos distintos por parte do SOTR em questão. Um utilizando a ferramenta de calibração, que é padrão no SOTR, que auxilia no ajuste do sistema ao *hardware*. A idéia principal da ferramenta de calibração é mostrar o quão preciso está o sistema e dar informações sobre as freqüências do computador, temporizadores e latências. Todos os testes são feitos em *software* e não utiliza nenhum outro recurso externo que possa dar mais veracidade aos resultados. Já no outro comportamento o desenvolvedor tem a possibilidade de alterar diretamente no código do SOTR, através do arquivo `asm-i386/rtai_hal.h` as definições de latência, freqüência e temporizadores. Observe as definições padrões criadas pelo sistema sem a utilização da ferramenta de calibração:

```
1 #define RTAI_TIMER_8254_IRQ       0
2 #define RTAI_FREQ_8254            1193180
3 #define RTAI_APIC_ICOUNT    ((RTAI_FREQ_APIC + HZ/2)/HZ)
```

Na linha 2 o desenvolvedor pode mudar o tempo da freqüência do relógio, expressa em nanosegundos, diretamente, porém deve-se conhecer se a plataforma de

hardware suporta ou não essa resolução. Com os valores alterados o *benchmark* teve um comportamento não tão satisfatório quanto utilizando os valores obtidos com a ferramenta de calibração. Por isso optou-se em usar os resultados obtidos pela ferramenta para que a análise em comparação aos outros SOTR pudesse ter o mesmo nível.

Quanto ao escalonamento o SOTR possui políticas de escalonamento: SCHED_FIFO e SCHED_RR. A função para atribuir tais políticas é definida pela API como `rt_set_sched_policy(RT_TASK *tarefa, int política, int rr_quanum_ns)`, porém caso o parâmetro `rr_quantum_ns` for passado um valor zero a política SCHED_FIFO é utilizada por padrão.

O padrão SCHED_OTHER que é encontrado em sistemas *Linux* padrão sofre modificações, porém não excluído, para que possa trabalhar junto com SCHED_FIFO, que adiciona uma funcionalidade no processo de alocação da memória para um processo possa ser reservado exclusivamente nesta, livrando de influencia de paginação ou de alocações dinâmicas de memória. Além disso, processos escalonados pela política SCHED_FIFO possuem prioridades estáticas com valores variando de 1 a 99, sendo inverso que o padrão POSIX define, ou seja, a prioridade maior é 1 e a menor 99.

Como o gerenciamento de tarefas por parte é executado em modo usuário, ao encontrar alguma tarefa do *Linux* que esteja escalonado pelo padrão SCHED_OTHER, este é preemptado e passado para o escalonador SCHED_FIFO que o coloca na mais baixa prioridade do sistema. Já a apolítica SCHED_RR é um simples aperfeiçoamento para que este possa também passar seus processos para o SCHED_FIFO, no qual cada processo só é permitido executar por um determinado limite de tempo. O sistema de tempo real em questão geralmente não usa a política SCHED_RR, sendo esta deixada para uso exclusivo de processos executado no sistema *Linux*.

A atribuição da prioridade é feita na própria estrutura de inicialização da tarefa de tempo real, como mostrada a seguir tanto para sistemas uniprocessados quanto para sistemas multiprocessados:

```
1 int rt_task_init ( RT_TASK *tarefa, void *rt_thread,
2     int data, int tamanho_pilha,
3     int prio, int uso_fpu,
4     void(*signal)(void) );
```

```
5 int rt_task_init_cpuid ( RT_TASK *tarefa, void *rt_thread,
6     int data, int tamanho_pilha,
7     int prio, int uso_fpu,
8     void(*signal)(void),
9     unsigned int cpuid );
```

O parâmetro `prio` é a prioridade dada para uma tarefa sendo o número mínimo e máximo de prioridades do sistema respectivamente 67108863 e 0 (zero). Um valor limite para a menor prioridade também pode ser atribuído pela definição RT_LOWEST_PRIORITY no arquivo `rtai_sched.h`.

Quanto ao tratamento de tarefas periódicas a API do SOTR disponibiliza a seguinte chamada de sistema:

```
1 int rt_task_make_periodic ( RT_TASK *tarefa,
2     RTIME tempo_inicio,
3     RTIME periodo );
```

Na qual a tarefa de tempo real é identificada pelo parâmetro `tarefa`, previamente criada com `rt_task_init()` como periódica, por um período definido no parâmetro `periodo`. Essa função para um controle mais aprimorado também pode trabalhar em conjunto com a função `rt_task_wait_period(void)` que suspende a execução de uma tarefa.

Quanto a questões relacionadas a tempo, a estrutura de definição de relógios/temporizadores no sistema operacional é armazenada em `RTIME`, que é expressa em nanosegundos assim como no *RTLinux*, porém não sendo incluída o parâmetro de segundos. A API deste tipo de estrutura obedece a seguinte definição:

```
1 struct timespec {
2     typedef long long RTIME;};
```

Podemos notar que essa definição é mais simples, favorecendo assim o acesso mais rápido, por processos. A função principal para obter o tempo do sistema é `rt_get_time()`, que retorna o tempo mensurado em unidades internas de contagem desde `rt_start_time()` foi chamada e expressa em nanosegundos. Uma função

similar à explicada anteriormente é a `rt_get_time_ns()`. Além disso, o sistema possui o recurso de *watchdog* como módulo programável para executar determinadas ações na ocorrência de excessos de uma tarefa.

A comunicação entre processos do RTAI é mais extensa do que a do *RTLinux*, isso incluí semáforos, *mutex*, variáveis condicionais, FIFO, *mailbox*, memória compartilhada, NET_RPC (serviço de chamada remota de procedimento por rede), filas de mensagens e *pqueues*.

No entanto o RTAI como não estabelece padrões para seus códigos fonte, os desenvolvedores do RTAI adicionam novos recursos e chamadas do sistema seguindo estilo pessoal de cada um não tendo um estilo coerente na programação. Com isso muitas vezes resulta em módulos do sistema com redundância e incompatibilidade. Um exemplo disso são as implementações de semáforos do RTAI que são incompatíveis [49]. Também quanto ao padrão de compatibilidade do POSIX se limita ao estabelecido pelas regras 1003.1c e o 1003.1b.

No que diz respeito aos semáforos, o sistema operacional pode ter dois tipo e incompatíveis entre si, sendo que o desenvolvedor escolhe o que deve ser mais indicado: semáforos do RTAI e semáforos de FIFO. O primeiro possui seis funções na API para inicialização, destruição e gerenciamento de semáforos:

- `rt_typed_sem_init()` : inicializa um determinado tipo de semáforo, que podem ser CNT_SEM, que conta e registra os eventos que ocorrem depois que acionado o semáforo; RES_SEM é um tipo especial de semáforo binário usado para gerenciamento de recursos, funcionando que uma tarefa ao adquirir um semáforo deste tipo se torna dono do mesmo, e este tem um aumento na prioridade sobre qualquer outra tarefa que esteja bloqueada ou esperando por este semáforo; e por último BIN_SEM que é utilizado para semáforos binários;
- `rt_sem_init()` : inicializa um semáforo com um determinado valor;
- `rt_sem_delete()` : destrói um semáforo;
- `rt_sem_signal()` : sinaliza um determinado semáforo;
- `rt_sem_wait()` : espera por um determinado semáforo;
- `rt_sem_wait_until()` : espera por um determinado semáforo até um determinado tempo.

Quanto semáforos FIFO possuem na API as funções:

- `rtf_sem_init()` : que inicializa um semáforo;
- `rtf_sem_destroy()` : destrói um semáforo;
- `rtf_sem_post()` : sinaliza que um semáforo está em uso;
- `rtf_sem_trywait()` : espera por um semáforo;
- `rtf_sem_timed_wait()` : espera por um semáforo.

Quanto ao recurso *mutex* o RTAI possui a API compatível com o padrão POSIX, assim como o *RTLinux*, porém a quantidade de tipos é mais extensa e envolve *mutex* recursivos que podem ser bloqueados mais que uma vez, por uma determinada tarefa sem causar um *deadlock* na própria e também *mutex* que verificam erros no uso e reporta ao sistema.

Quanto a filas de mensagens, o sistema, fornece quatro diferentes facilidades para comunicação entre tarefas, a fila propriamente dita compatível, com o padrão POSIX 1003.1b, que pode ser manipulada pelas funções `mq_open()`, `mq_close()`, `mq_send()` e `mq_receive()`. O segundo tipo de filas é através de mensagens pequenas de 4 *bytes* cada, esta já usando um padrão próprio do sistema operacional, manipuladas pelas funções `rt_send()`, `rt_receive()`, `rt_close_mq()` e `rt_open_mq()`. A terceira é uma extensão do segundo tipo fornecendo a comunicação com mensagens maior que 4 *bytes* e a última é o padrão de troca de mensagens em que as tarefas que se comunicam entre elas esperam por uma confirmação.

Memória compartilhada e FIFO são iguais ao descrito anteriormente para a API do *RTLinux* sem alterações ou modificações, no entanto a API para memória compartilhada possui uma similaridade ao padrão SYSTEM V no que diz respeito a alocação de memória, em que a primeira vez que uma área na memória é requisitada para uso esta retorna um ponteiro para a memória alocada.

3.4. *Windows* CE versão *5*

O SOTR *Windows* CE foi construído especificamente para sistema embarcados e aplicações móveis, fornecendo uma plataforma escalável que pode ser utilizada em uma grande variedade de tarefas. O grande diferencial do *Windows* CE para outros SOTR é que este possui uma grande variedade de funções e serviços focados em *Internet*.

3.4.1. Avaliação

O sistema *Windows* CE é o SOTR da *Microsoft*, tipicamente usado em sistemas sem disco com limitada capacidade de memória. Este SOTR é adaptado já da conhecida plataforma *Win32* e pode ser construído para um *hardware* específico através de uma pequena camada de código. O conceito de HAL também é usado neste sistema operacional, porém não usando um sistema hospedeiro que seja adaptado para tempo real.

Uma das grandes vantagens do desenvolvimento neste sistema operacional são as ferramentas disponíveis para isso, entre elas, a mais importante em que é responsável pela criação de imagens e serviços que iram fazer parte do sistema operacional, o *Platform Builder*.

Através do *Platform Builder*, é possível customizar o sistema operacional para diversas aplicações utilizando interfaces intuitivas e de fácil entendimento, este tipo de auxilio faz com que o desenvolvimento na plataforma torne-se rápido, diminuindo consideravelmente os custos de desenvolvimento. Este tipo de recurso é conhecido como **Assistentes de Criação** ou pelo termo em inglês de *wizards* e estão disponíveis para as mais diversas aplicações.

A arquitetura do *Windows* CE 5 não é tão semelhante ao que foi visto no Capítulo 2 na arquitetura do *Windows* CE versão .NET, diferenciando da quantidade de serviços e a representação modular de cada recurso (Figura 13).

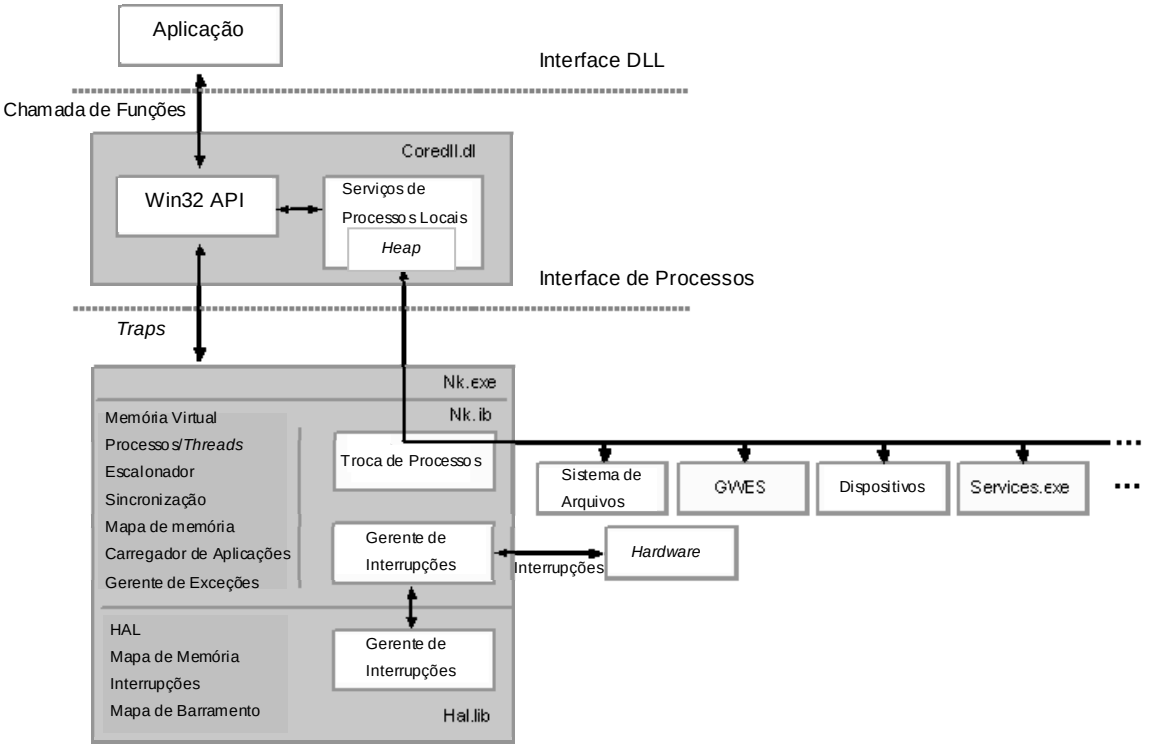


Figura 13 - Arquitetura do *Windows* CE versão 5 [37]

O *kernel* do SOTR também é enquadrado na categoria de *microkernel* já que o mesmo é fornecido pela estrutura MINKERN, que é construído sem recursos gráficos e sem interface de usuário, apenas os serviços básicos para funcionamento do sistema. Esta estrutura reside no arquivo *Nk.exe*, que possui duas camadas a *Nk.lib* e a *Hal.lib*.

A primeira camada é responsável pela comunicação com outros módulos que possam ser adicionados ao sistema como a MINSHELL, MININPUT e MAXALL que são imagens de referência para determinados tipos de configurações de plataformas, esta camada também é importante, pois possui o escalonador, gerenciador de memória, carregador de aplicações entre outros serviços, também é responsável para interligação do *hardware* com o gerenciador de interrupções, este sendo parte integrante da segunda camada, a *Hal.lib* que é a camada mais baixo nível dentro da arquitetura e responsável para controle de interrupções propriamente dita.

A interface de processos faz ligação com a imagem do sistema operacional, possibilitando o uso da API *Win32*, além de possuir uma área *heap*, que é colocada na memória uma área usada para armazenar recursos importantes tais como semáforos, pilhas, filas, entre outros. Também o *heap* otimiza o uso de memória, isolando uma aplicação de tempo real de possuir diferentes tamanhos de página para alocação.

A API do sistema é muito extensa, possuindo mais de 500 funções, e não possui compatibilidade com o padrão POSIX [51]. Demais serviços do sistema operacional são disponibilizados de acordo com a necessidade do desenvolvedor, facilitando a criação da solução de tempo real e possibilitando uma economia em espaço disponível e recursos da plataforma, este é um importante aspecto do SOTR em questão, um alto grau de modularidade do sistema operacional.

Tamanho

O *Windows* CE possui um tamanho fixo em sua unidade básica, *Nk.exe*, que possui o tamanho de 350 KB, demais módulos possuem tamanho conforme a plataforma utilizada. O tamanho total do SOTR é considerado pequeno em comparação aos outros SOTR, já que este não precisa de um sistema hospedeiro como os outros em estudo o que diminui consideravelmente. O tamanho de uma imagem básica do sistema, sem o uso de interface gráfica é de 415 KB.

Modularidade

O projeto modular do SOTR permite que vários componentes possam ser anexados ao sistema de forma prática, fazendo com que os programas sejam executados isoladamente. Cada processo de tempo real criado pelo SOTR em questão executa em um espaço de memória protegido. A modularidade do sistema permite que o núcleo de serviços seja executado de tal maneira que não interfira com outros recursos do sistema como gerenciador de dispositivos, aplicativos e outros serviços.

O número máximo de processos executando no sistema é de 32. Por causa do número mínimo de memória que o sistema operacional pode ser alocado, esse pode possuir diversos módulos discretos, cada um com uma funcionalidade específica tal como sistema de arquivos, recursos gráficos também conhecido como GWEC (*Graphic, Windowing and Event Components*) e módulos de comunicação. Muitos desses módulos são divididos em componentes ajudando o desenvolvedor que pode minimizar os recursos de *hardware*, tais como RAM e ROM.

O isolamento de processos faz com que cada tarefa do SOTR seja executada como uma PTR. Cada processo deste SOTR, nativamente, possui um mecanismo de rastreamento do mesmo, como o LTT do RTAI, e as aplicações são executadas por

padrão em modo usuário. A unidade básica do PTR também é a *thread* e a mesma obedece a estrutura seguinte:

```
1 Function CreateThread
2 HANDLE CreateThread(
3     LPSECURITY_ATTRIBUTES lpThreadAtributos,
4     DWORD dwTamanho,
5     LPTHREAD_START_ROUTINE lpEnderecoInicio,
6     LPVOID lpParametro,
7     DWORD dwFlag,
8     LPDWORD lpThreadId);
```

Sendo que `lpThreadAtributos` é geralmente usado como atributo padrão da função de inicialização da *thread*, por padrão é NULL, `dwTamanho` define o tamanho da pilha da *thread* criada, geralmente também é ignorado a não ser que seja especificada, `lpEnderecoInicio` é ponteiro para o endereço inicial da *thread*, `lpParametro` guarda o parâmetro que aponta uma variável da *thread*, `dwFlag` é a *flag* responsável pelo controle de criação da thread, por padrão é 0 (zero) e `lpThreadId` o identificador da *thread*.

Quando uma *thread* é criada pelo processo, o *kernel* reserva uma determinada porção da RAM para manter as variáveis desta, por padrão o tamanho é de 64 KB e mantida isolada das demais porções de memória. Este espaço é desalocado assim que a *thread* termina sua execução.

Um ponto questionável quanto ao gerenciamento de memória do *Windows* CE é que este possui um gerenciamento baseado em memória virtual sem garantias de previsibilidade de tempo, para suprir tal deficiência *threads* de tempo real podem ter seu nível de prioridade maior que a prioridade da paginação e as mesmas sejam mantidas na memória sem interferência deste evento, prevenindo de atrasos não determinísticos. A estrutura responsável, de acordo com a API do SOTR, pela alocação fixa da memória é descrita como :

```
1 HLOCAL LocalAlloc(
2  UINT uFlags,
3  UINT uBytes);
```

Na estrutura anterior, um número específico de *bytes* é alocado para uma estrutura *heap*. o parâmetro descrito pela linha 2 especifica como alocar a memória, ou seguido o valor LMEM_FIXED, que aloca a memória fixa, ou o valor pode ser LMEM_ZEROINIT que inicializa a memória fixa com zeros. Já na linha 3 o parâmetro especifica o tamanho, em *bytes*, que deve ser alocado.

Adaptabilidade

O *Windows* CE é um sistema de adaptação reativa, pois o mesmo é construído para que possa ser totalmente padronizado para a plataforma alvo que for projetada. Em algumas situações o sistema pode ser considerado com adaptação reativa parcial em resposta a eventos externos, porém previsíveis, que o desenvolvedor possa adaptar o sistema para tais situações ainda em tempo de projeto.

O SOTR também não consegue adaptar o sistema a um eventual acréscimo de dispositivo. A imagem gerada durantes os testes só funcionou na plataforma que foi projetada. A imagem do SOTR é bem pequena o que torna possível a inclusão deste em sistemas com baixa memória física disponível.

A compatibilidade do sistema com a API já muito bem difundida do *Win32* faz com que uma grande variedade de funções possa ser utilizada no sistema, poupando o desenvolvedor de aprender uma nova metodologia de desenvolvimento para criar aplicações. Porém a maior parte das questões de adaptabilidade do sistema se limita ao *Platform Builder* que fornece ao desenvolvedor o suporte necessário para que o desenvolvimento de aplicações de tempo real e embarcado seja feito, pois organiza e apresenta ao desenvolvedor todos os componentes que podem ser adicionados a uma imagem do sistema operacional e permite que possa ser escolhido apenas componentes necessários ao projeto básico de funcionamento do mesmo.

O fabricante do SOTR disponibiliza junto com o pacote acesso aos códigos fonte do sistema operacional. Os códigos fontes estão disponíveis tanto para estruturas internas do sistema como escalonadores, filas e interrupções. Também nesse SOTR não encontramos nenhuma documentação consistente que seja incluída com o pacote para demonstrar as principais funcionalidades e explicações dos códigos disponíveis com o sistema. O comportamento das aplicações que executem no SOTR pode ser analisado utilizando a ferramenta *Kernel Tracker* através desta ferramenta é possível visualizar

certas condições do sistema em uma determinada configuração como perda de *deadlines*, *watchdog*, entre outras.

O número de plataformas de *hardware* suportadas pelo sistema operacional inclui x86, ARM, SH e MIPS e não possui suporte a SMP. A possibilidade de alteração do escalonador do sistema pode ser feita diretamente através da modificação do arquivo responsável pelo escalonamento do sistema, o arquivo `schedule.c`. De certa forma isso é muito trabalhoso já que o arquivo é muito extenso e com as funções muito aninhadas uma com as outras. Uma outra forma de escalonar os processos é criar a própria política de escalonamento colocando a mesma para executar como se fosse uma tarefa com a mais alta prioridade entre todas e possa dessa forma escalonar as *threads* que façam parte da mesma.

No tratamento de interrupções do sistema operacional, o mesmo utiliza interrupções compartilhadas que executam como um dispositivo do sistema e funciona como gerenciador destas interrupções, através da função `InterruptInitialize()` que é usada diretamente pelo *kernel* do sistema para acionar as interrupções e mapeia os eventos que são criados usando a função da API *Win32* `CreateEvent()`.

Essa chamada faz com que o sistema espere até que o serviço de interrupções informe ao *kernel* quais eventos são permitidos executar. Quando a interrupção ocorre, o *Nk.exe* salva o estado da tarefa que esteja em execução e cria um identificador para a mesma. Quando isso ocorre o manipulador de interrupções desabilita todas as interrupções de igual ou menor prioridade e então chama a interrupção requisitada e retorna um valor lógico para esta interrupção, na forma de um identificador.

Ao final todas as interrupções do sistema são acionadas novamente e o manipulador chama a função `InterruptDone()` que retorna o estado anterior de determinada tarefa que estava sendo executada com o identificador salvo no *Nk.exe*.

Previsibilidade

Quanto a previsibilidade o SOTR em questão possui um algoritmo de escalonamento baseado em prioridades e preemptivo. Uma *thread* que for criada no sistema pode ter uma das 256 prioridades possíveis (faixa de 0 a 255), divididas como de 0 a 96 reservada para dispositivos de tempo real, 97 a 152 usado por padrão de dispositivos do SOTR, 153 a 247 reservado ao sistema e de 248 a 255 para tarefas não prioritárias.

Quanto maior o valor menor a prioridade. *Threads* que possuírem a mesma prioridade executam em *round-robin* e quando uma destas é bloqueada retorna a execução quando todas as outras de mesma prioridade estiverem finalizado ou, utilizado seu tempo definido pelo escalonamento que em geral é de 100 ms (milisegundo), sendo que este valor pode ser modificado pela função `CeSetThreadQuantum()`. *Thread* que possuírem prioridades menores só irão executar, quando todas as outras de maior prioridade finalizarem ou estiverem bloqueadas.

Para manipular as *threads* que estejam no sistema algumas funções foram criadas especificamente para tratamento das mesmas: `CeGetThreadPriority()` possui a função de retorna a prioridade de uma determinada *thread*; `CeSetThreadPriority()`, função atribui prioridade; `Sleep` que suspende a execução da *thread* corrente pelo intervalo de tempo especificado; `SuspendThread()` que suspende a execução e `ResumeThread()` que é a função que reinicia a execução de uma *thread*.

Quanto a questões relacionadas a tempo, o SOTR usa um intervalo de tempo definido de acordo com a plataforma que esteja em execução. O temporizador tem uma precisão de 1 milisegundo na chamada da função `Sleep()`. O escalonador do sistema utiliza os valores de tempo determinados por software da API *Win32* e que podem ter uma resolução de até 10 ms.

Caso não tenha nenhuma *thread* para ser escalonada pelo sistema, a função `OEMIdle()` é chamada pelo escalonador, em que reprograma o intervalo de tempo do sistema baseado na quantidade de tempo ocioso do mesmo, e coloca o sistema em um estado de economia de energia. Isso de certa forma retira um pouco da previsibilidade do sistema operacional, já que inclui um tempo de retorno a condições a receber aplicações de tempo real que não era previsível pelo desenvolvedor.

Quanto a sincronismo o SOTR em questão possui uma série de objetos que permitem que uma *thread* se comunique com outra. Esses objetos incluem seções críticas, mutex, semáforos, watchdog e filas de mensagens. Uma descrição das mesmas é muito extensiva por conta das mais diversas funções que o sistema possui, sendo que uma leitura mais aprofundada do sistema pode ser obtida em [37], porém para o trabalho apenas as funções de semáforo e seções críticas vão ser abordadas, pois as mesmas foram utilizadas.

Os semáforos podem ser utilizados e manipulados pela seguinte estrutura:

```
1 HANDLE CreateSemaphore(
2     LPSECURITY_ATTRIBUTES AtributoSemaforo,
3     LONG ContInicial,
4     LONG ContMaxima,
5     LPCTSTR nome);
```

Qualquer *thread* pode manipular um semáforo e chamar qualquer uma das funções de manipulação do semáforo. Os estados do semáforo podem possuir uma contagem definida pela linha 3 e atingir o valor máximo definido pela linha 4, sendo que esta contagem não pode ter valores negativos. Múltiplas *threads* podem acessar o mesmo semáforo desde que estes na hora da criação do semáforo possuam o mesmo nome. A finalização do semáforo é feita de forma automática quando a *thread* que a chamou for finalizada.

Quanto a seções críticas o *Windows* CE possui uma estrutura específica para a criação deste objeto. A função `EnterCriticalSection()` é indicada para que uma *thread* possa acessa uma região crítica do código fazendo com que a mesma execute sem que tenha interferência de outra *thread*. Caso outra *thread* entre em seção crítica esta é bloqueada até que a função `LeaveCriticalSection()` seja executada.

Para usar uma seção crítica uma estrutura chamada CRITICAL_SECTION deve ser declarada, visto que a função requer um ponteiro para esta estrutura, e esta declaração deve ser feita no escopo de todas as funções que a forem utilizar, para finalizar o uso de uma seção crítica a função `DeleteCriticalSection()` deve ser chamada.

Quanto ao gerenciamento de memória o SOTR implementa um sistema de memória virtual com paginação similar a outros produtos da fabricante do SOTR. Esse padrão é criar páginas que possuam um tamanho de 4 KB, e que cada processo criado no sistema operacional possua apenas 32 MB de espaço de endereçamento virtual. Outras estruturas como pilhas, *heap*, bibliotecas e alocações de memória virtual usam também o mesmo tamanho. Por padrão a memória é alocada pela função `VirtualAlloc()` que reserva uma região de páginas para o processo que a chama a estrutura a seguir:

```
1 LPVOID VirtualAlloc(
2     LPVOID endereco,
3     DWORD dwTamanho,
4     DWORD TipoAlocacao,
5     DWORD flProtect);
```

O tipo de alocação indicado pelo parâmetro da linha 4 pode ser MEM_COMMIT que aloca um espaço na memória ou em um arquivo de paginação em disco para uma específica região de páginas; MEM_RESERVE que reserva uma faixa de endereço virtual sem alocar um espaço físico e MEM_TOP_DOWN, que aloca a memória no espaço de endereçamento maior possível. Sabe-se que usar memória virtual não é considerada uma forma confiável para a alocação de processos de tempo real principalmente em aplicações críticas. Uma solução encontrada é utilizar as definições de *heap* descrita anteriormente.

3.5.Quadro comparativo dos sistemas operacionais de tempo real

A seguir Quadros comparativos com as principais funcionalidades analisadas são mostrados:

Pelo exposto no Quadro 2 a primeira coluna mostra os SOTR estudados no presente trabalho, a segunda coluna mostra que tipos de processadores são suportados por cada sistema, a terceira coluna mostra se os sistemas têm suporte a múltiplos processadores, na quarta coluna mostra as políticas de escalonamento usados, na quinta e sexta colunas mostram respectivamente que unidade são usadas para usar tarefas de tempo real e a prioridade das tarefas no escalonador.

Quadro 2: Avaliação dos resultados levando em consideração *hardware*, SMP, Políticas de escalonamento, unidades de tempo real e prioridades.

SOTR	***Hardware* suportado**	**Suporte a SMP**	**Políticas de Escalonamento**	**Unidade de tempo real**	**Prioridades (mín. – max.)**
RTLinux	x86, *Power* PC, ARM.	Sim, apenas para x86.	Escalonamento tipo FIFO, escalonamento customizável, escalonamento baseado em servidores.	*Threads* (padrão POSIX)	0 – 1000000
RTAI	x86, MIPS, *Power* PC, ARM, Motorola m68k-nommu.	Sim, apenas para x86.	Prioridade fixa, escalonamento FIFO e escalonamento round-robin (time sharring).	*Threads* (padrão POSIX), tarefas (rt_task_init)	67108863 – 0
Windows CE	x86, ARM, MIPS	Não	Prioridade fixa, *round-robin* (time sharring)	*Threads* (não POSIX usando função CreateThread)	0 – 255

Já no Quadro 3 mostra um comparativo entre os sistemas operacionais analisando os pontos de compatibilidade da API, métodos de comunicação entre os processos, gerenciamento para controle de inversão de prioridades e de estrutura para proteção de memória.

Quadro 3: Avaliação dos resultados levando em consideração tamanho, API, IPC, controle de inversão de prioridades e proteção de memória.

SOTR	**Tamanho**	**Compatibilidade da API**	**IPC**	**Controle de Inversão de Prioridade**	**Proteção de memória**
RTLinux	3.3 MB, sem considerar sistema *Linux.*	POSIX 1003.1c, 1003.13/PSE51, *RTLinux* V. 1.0, implementada em linguagem C (pode usar linguagem C++ por módulo complementar instalado no SOTR).	semáforo, *mutex*, memória compartilhada, sinais, FIFO.	Usando mutex.	Não
RTAI	630 KB, sem considerar sistema *Linux*.	sem padrão nativo, *RTLinux* V. 1.0, POSIX 1003.1c e 1003.1b implementada em linguagem C/C++	semáforo, *mutex*, variáveis condicionais, FIFO, memória compartilhada, filas POSIX (*pqueues*), NET_RPC, *mailbox.*	Usando *mutex.*	Sim, quando usado aplicações em modo usuário no *Linux.*
Windows CE	Varia de acordo com a solução. Tamanho básico de 415 KB	*Win32*, implementada em C++, não compatível com POSIX	Seções Críticas, *mutex*, semáforos, *watchdog*, filas de mensagens.	usando *mutex* ou seções críticas.	Sim, uso de paginação.

Capítulo 4 – Avaliação Comparativa a partir dos resultados obtidos com os *benchmarks*

Mostra-se nesse Capítulo a idéia do *benchmark* desenvolvido e das técnicas de avaliação de latência e troca de contexto também são avaliados os resultados obtidos durante a execução do *benchmark*. Além de como foi criado o *benchmark* pelo autor seguindo esquemas do *MiBench* e *Hartstone*.

É importante para um SOTR fornecer ao desenvolvedor uma forma eficiente e segura para execução e criação de aplicações de tempo real. Possuir um grupo consistente e robusto de estruturas e ferramentas torna o processo de desenvolvimento de sistemas mais previsíveis e com baixa probabilidade de erros.

Isso é fundamental já que diferentes filosofias de projetos de SOTR existem e cada uma com suas vantagens e desvantagens. Porém não podemos deixar de escolher um determinado sistema operacional por falta de uma ou outra estrutura.

De certa forma um sistema operacional enxuto e mínimo é o ideal para aplicações que utilizam conceitos de tempo real, pois diminuem conflitos nas suas estruturas internas e deixa o sistema menos imperativo nas soluções de tempo real. A previsibilidade estabelecida em tempo de projeto, para uma aplicação de tempo real, não pode em hipótese alguma ter sido alterada drasticamente na inclusão de um determinado SOTR.

Também levamos em consideração que a previsibilidade do sistema é apenas uma entre várias características que definem um sistema operacional de tempo real. Baseado nos resultados encontrado na execução do *benchmark* é mostrado nesse trabalho uma forma gráfica beneficiando o leitor para o entendimento dos mesmos.

Os resultados dos testes de latência e interrupção foram comparados a outros feitos pela *Microsoft* [38] e Ripoll *et al* [49] a fim de observar se os valores dos resultados efetuados em *software* se aproximam a testes feitos em *hardware*. Muitas das complexas funcionalidades e descrições algorítmicas foram omitidas dos resultados e nas explicações.

Foi criado um *benchmark* que foi fundamentado em outras duas soluções para testes de sistemas de tempo real o *MiBench* [23] e o *Hartstone* [65]. Verificou também a resposta do sistema na capacidade de executar tarefas do tipo *hard* e a manipulação de interrupções com a execução de tarefas em dois momentos, o primeiro que agem de forma periódica e não-harmônica e o segundo momento que executa de maneira periódica com uma carga de tarefa aperiódica.

4.1.Métodos de Análise

Análises e medidas, confiáveis e corretas requer definições claras do que realmente deseja pesquisar. Atributos de objetos são medidos preferencialmente do que os próprios objetos. Atributos de um determinado sistema operacional podem ser **Medidas diretas** ou **Medidas indiretas** dependendo da forma de estudo.

Medidas diretas podem ser entendidas quando nenhum outro atributo ou entidade tenham de ser medidos, por exemplo, quando se deseja medir a produção do sistema (*throughput*) de um determinado sistema operacional apenas envolve a atividade de contar a quantidade de transações executadas por determinada unidade de tempo. Enquanto que medidas indiretas requerem um ou mais atributos para obter resultados consistentes, por exemplo, qualidade de *software* pode ser vista como uma combinação complexa de atributos, quando se deseja medir um atributo simples, como linhas de código, raramente se obtém um resultado útil.

Quando estamos analisando de forma científica SOTR temos de envolver técnicas específicas de estudos e observar o comportamento interno destes sistemas. Fazer estudos superficiais dos sistemas operacionais não é a melhor opção para que possamos entender realmente para que serve um determinado sistema, é necessária uma investigação mais profunda dos pontos que foram relacionados anteriormente.

Deve-se observar o que realmente importa quando se deve medir um SOTR. Alguns pontos devem ser conhecidos quando trabalhamos com análise de sistemas operacionais como seus atrasos, latências e eficiências em determinadas situações.

Uma série de testes de SOTR foram feitos pela empresa *Dedicated Systems Encyclopedia* [14], em casos específicos de tais sistemas como o *QNX Neutrino*, *Windows* CE versão .NET e *VxWorks* utiliza técnicas que envolve a observação direta dos pontos abordados anteriormente, além de que outras formas de análise com uso de aplicações específicas foram utilizadas. O presente trabalho não tem como objetivo principal utilizar aplicações específicas de tempo real e sim gerar uma visão global de análise de dados que são pertinentes a qualquer que seja o SOTR.

A maioria dos SOTR possuem temporizadores, em *softwares*, que podem ser utilizados como uma ferramenta de medida de tempo. Mas temporizadores em diferentes sistemas operacionais não tem a mesma resolução. Além do mais, estes nem sempre oferecem a precisão necessária para testes. O uso de temporizadores nativos de

um SOTR específico adiciona um *overhead* não-previsível aos resultados, visto que as medições são executadas pelo próprio sistema operacional enquanto este executa o teste.

A solução ideal para o problema é utilização de algum dispositivo (geralmente algum *hardware* externo) que não esteja fazendo parte do sistema operacional. Um exemplo disso é o uso de osciloscópios ou analisadores de barramentos para executar tais medições [14]. Por outro lado, essas formas de análise muitas vezes são muito dispendiosas e exige um conhecimento muito abrangente do *hardware* em questão, sendo que para diferentes plataformas uma solução específica deve ser criada. A forma mais viável ainda é determinada em *software* com o uso de *benchmarks*, porém para sanar a deficiência da não-precisão dos temporizadores utiliza-se de estipulações de tempos máximos para execução das tarefas.

O presente trabalho utiliza tais técnicas de desenvolvimento, além de que propõe uma solução genérica de *benchmark* em que possa ser utilizada por outros SOTR que não estejam citados no escopo deste trabalho. A idéia inicial é definir o que deve ser medido e suas equações, logo após utilizar as unidades básicas das tarefas de cada sistema operacional – geralmente *threads* – acrescentando as definições propostas, observe a Figura 14.

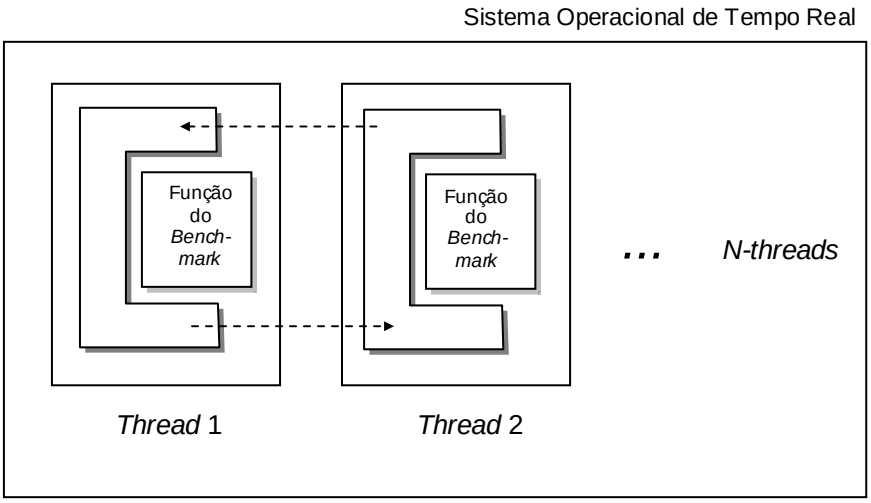


Figura 14 - Modelo de tarefas do *benchmark*

Vemos na Figura 14 que cada *thread* possui uma função específica do *benchmark*, sendo que essa é desenvolvida não como componente específico do SOTR e sim de uma forma genérica para que possa ser portável para outros ambientes operacionais. As setas pontilhadas indicam que ocorre comunicação entre as tarefas, quando necessário, usando os próprios recursos de comunicação do SOTR.

4.1.1. Latências e atrasos em Sistemas Operacionais de Tempo Real

Todos os parâmetros relacionados à execução dos testes de tempo real devem ser conhecidos e determinados, assim os testes poderão ser validados. No caso do presente trabalho os testes medem os tempo de cada execução de um determinado sistema operacional. Os parâmetros devem ser bem conhecidos fazendo dessa forma uma comparação sensata entre os resultados obtidos em diferentes ambientes. Contudo deve-se ter em mente que a carga do sistema operacional também influencia no resultado. A carga do sistema depende do número de tarefas que estejam no sistema, uma forma de demonstrar isso é através das filas de tarefas que são criadas em sistemas operacionais toda vez que uma tarefa não consegue um determinado recurso.

A carga do sistema operacional também é influenciada pela quantidade de dados manipulados durante a troca de contexto entre as tarefas. Para reduzir esse tempo na troca de tarefas, muitos SOTR implementam como unidade básica de tempo real o recurso de *threads* que são um ou mais fluxos de trabalhos executados dentro de uma determinada tarefa. Como estas, geralmente, compartilham o mesmo endereçamento de memória e carregam muito menos informações, o *overhead* causado pelo armazenamento do estado de uma *thread* preemptada por outra é reduzido. A quantidade de dados contidos em um bloco de controle de uma determinada *thread* depende da configuração de memória que o ambiente esteja usando.

Como citado anteriormente no Capítulo 2, as tarefas podem ser executadas em modo usuário ou modo *kernel*, porém outras formas de configuração para gerenciamento de memória podem existir, como o **modo sem proteção** em que não existe proteção de memória entre as *threads* do sistema e não requer nenhuma Unidade de Gerenciamento de Memória (MMU, de *Memory Management Unit*) e o **modo de proteção privada**, em que cada processo de usuário é atribuído em seu próprio espaço de endereçamento virtual, nesse caso exige o uso de uma MMU. A seguir os principais fatores de atrasos e latências e a análise de cada uma deste será demonstrada para um melhor entendimento de como o *benchmark* foi modelado. Existem diversos atrasos gerados em um sistema operacional de tempo real como latência do escalonador, duração do tempo de escalonamento, latência de gerenciamento de recursos, etc. sendo que a seguir serão analisados os dois atrasos que mais são utilizados nos testes de previsibilidade do sistema que é a troca de contexto e a latência de interrupção.

Troca de Contexto entre *threads*

A troca de contexto entre *threads* é o tempo em que o sistema operacional usa para interromper e salvar o contexto de uma *thread* que esteja em uma execução e preparar uma outra *thread* para ser executada. Observe a Figura 15, ela decompõe os principais componentes que são encontrados numa troca de contexto entre três *threads*, sendo C_n o tempo de execução, TC_n o tempo de troca de contexto e T_T o tempo total do conjunto dado.

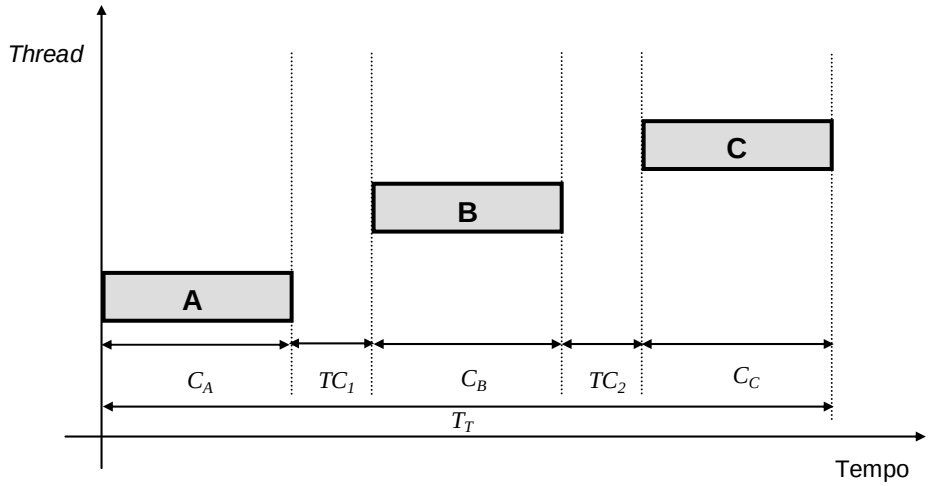


Figura 15 - Troca de contexto entre *threads* e componentes envolvidos na mesma [56]

Pelo exposto, o tempo que um determinado conjunto de *threads* executa (T_T) é a soma do tempo de todas as trocas de contexto e todos os tempos de execução. Deriva-se dessa afirmação o valor do tempo da troca de contexto de um determinado conjunto de *threads*:

$$T_T = \sum C_n + \sum TC_n \quad \therefore$$

$$\sum TC_n = T_T - \sum C_n \textbf{ (1)}$$

Podemos também afirmar que o tempo da troca de contexto em um determinado tempo é definido por:

$$\sum TC \approx (n-1)TC \quad \therefore$$

$$TC = \sum TC \Big/ (n-1) \textbf{ (2)} \therefore$$

$$\textbf{(1) e (2)} \rightarrow TC = (T_T - \sum C) \Big/ (n-1) \textbf{ (3)}$$

A fórmula **(3)** é o valor aproximado para uma *thread* sofrer a troca de contexto em um determinado momento e considerando *n* sendo o número de tarefas.

Essa equação é apenas uma suposição, pois é muito difícil determinar o valor do tempo de execução de uma determinada *thread* com precisão, no caso descrito é considerando que os tempos de computação das mesmas não sofrem mudanças significativas. É importante salientar que variando o número de *threads* no conjunto, mostrará se o gerenciamento de filas das mesmas pelo sistema operacional é eficiente.

Latência de interrupção

Entender como calcular o tempo de latência de interrupção em tarefas de um SOTR é fundamental para que possamos determinar com precisão se ou não o gerenciador de interrupções do sistema operacional funciona no contexto de um *thread* que esteja executando ou um diferente ambiente. Para calcular a latência de interrupção deve-se primeiro entender como é o processo de geração deste evento, observe a Figura 16.

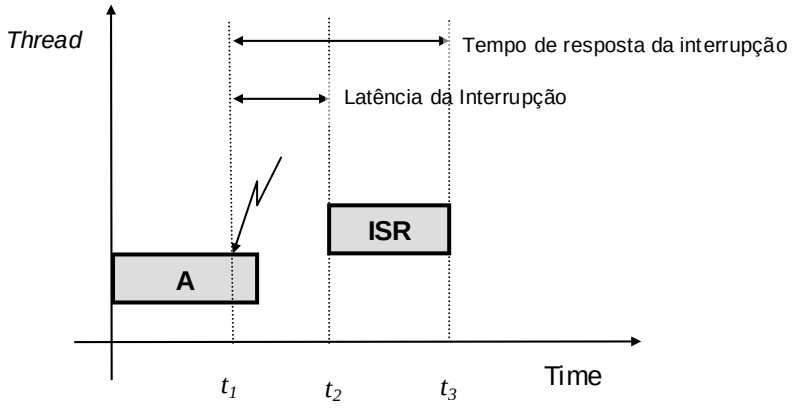


Figura 16 - Latência de interrupção e Tempo de resposta da interrupção

Vemos na Figura 16, que em um determinado instante t_1 um interrupção é gerada, essa leva um determinado tempo (t_2) para que comece a ISR, no tempo t_3 é finalizado o processo de interrupção. Para determinar a latência de interrupção de forma precisa pode ser obtido com o auxílio de um equipamento osciloscópio, porém no presente trabalho o uso do cálculo desse tempo não tem uma precisão muito grande já que é determinado em *software*. Podemos expressar matematicamente através da expressão:

$$T_{li} = T_{latencia\_CPU} + MAX(T_{SOTR\_irq}, T_{app\_irq}) \quad \textbf{(4)}$$

$$T_{ri} = T_{li} + WCET\_ISR \quad \textbf{(5)}$$

Sendo que T_{li} é o tempo da latência da interrupção, T_{ri} é o tempo de resposta da interrupção, $T_{latencia\_CPU}$ é o atraso gerado pela CPU, WCET_ISR é o tempo de pior caso de execução da ISR, T_{SOTR_irq} é o tempo que o sistema operacional desabilita as interrupções e T_{app_irq} é o tempo que a aplicação desabilita as interrupções. Estas ultimas unidades são apenas desabilitadas por um curto período de tempo. Devemos novamente considerar que cálculos de medidas de latência de interrupção vão incluir em seus resultados qualquer atraso gerado por periféricos do computador.

4.1.2. Uso de *Benchmark* para avaliação da previsibilidade

Benchmark é uma aplicação ou conjunto de aplicações que exercitam um determinado sistema baseado em diferentes parâmetros de entrada e retornando valores que possam ser analisados com finalidade de trazer melhorias [67]. São usados largamente para analise de performance em sistemas computacionais. Os *benchmarks* específicos para cada aplicação são encontrados e utilizados nas mais diversas aplicações, como dispositivos de rede, telecomunicações, entretenimento digital entre outras.

Nos testes realizados optou-se em utilizar *benchmarks* de conceitos comuns em sistemas de tempo real: troca de contexto, tratamento de interrupções, chamadas de sistemas (*system calls*), tempo de execução de uma *thread* em um determinado tipo de tarefa e outras intervenções geradas pelo sistema operacional. Tipicamente esse tipo de *benchmark* tenta isolar uma característica específica de tempo real e medi-la. Para

minimizar o teste realizado por *software*, repete as operações um número considerável de vezes de modo que consiga uma média com valores significativos.

Definir um *benchmark* envolve um longo e árduo processo de estudo e pesquisa. Um outro desafio é determinar como criar *benchmarks* portáveis suficientemente para que possam ser executados em uma grande variedade de plataformas, processadores e configurações. Muitos destes sistemas apresentam uma carga fixa de trabalho, o que pode ser demonstrado em aplicações que seguem algum tipo de padrão. O consórcio *Embedded and Microprocessor Benchmark Consortium* (EEMBC [63]), formado por grandes empresas de microeletrônica como IBM, ARM, NEC, Motorola entre outras, é uma instituição sem fins lucrativos que desenvolve padrões para *benchmarks* de aplicações embarcadas e de tempo real, tanto em *software* quando em *hardware*, que tentam refletir ao máximo aplicações reais. O alvo de tais sistemas são aplicações automotivos, industriais, redes, automação e telecomunicações.

A EEMBC derivou 37 algoritmos para os domínios citados que constituem a base de sua suíte de aplicativos, que além de fornecer e desenvolver códigos de *benchmark* também procura certificar aplicativos para setores públicos tais como comunidade de desenvolvimento de sistemas embarcados. Para criação do *benchmark* optou-se em utilizar as definições e conceitos operacionais definidos em dois sistemas o *MiBench* e o *Hartstone*. A escolha do *Mibench* deu-se pelo fato de que o mesmo constitui de um sistema acessível à comunidade acadêmica e segue o modelo proposto pelo EEMBC. Este é dividido em seis domínios: Controle automotivo e industrial, dispositivos para consumidores, automação de escritório, redes, segurança e telecomunicações. Cada domínio possui um conjunto de códigos abertos.

Apesar das similaridades com a suíte da EEMBC, o sistema *MiBench* é composto por algoritmos próprios e de algumas peculiaridades, como o uso de aplicações que exigem bibliotecas específicas para teste, o que limita a abrangência de ambientes que podem utilizar tais aplicações. No trabalho não foi feito uso de seus algoritmos ou nenhuma cópia, apenas das definições de uma categoria específica que é a do domínio automotivo. Nesse domínio o *benchmark* é usado para demonstrar como sistemas de controle críticos são testados e exigem dos sistemas que saibam como manipular *bits* de forma confiável, habilidade em tratar com problemas matemáticos simples e organização de dados. Os testes são usados para caracterizar situações típicas de aplicações de tempo real como sistema de *airbag*, controle de combustível e sensores do motor.

As funções das aplicações do *benchmark* são:

- Efetuar cálculos matemáticos: Efetua a transformação de um valor em graus para radianos e vice-versa; resolve um cálculo polinomial cúbico; cálculo de raiz quadrada de valores inteiros e de ponto flutuante;
- Ordenar através do algoritmo *quicksort* 20 elementos inteiros;
- Multiplicação de matriz;
- Transformada Rápida de Fourier;
- Manipulação de *bits*.

O Apêndice A mostra o fluxograma dos testes criados e a explicação geral de cada um, seguindo o padrão definido pelo EEMBC. Nenhuma chamada de biblioteca específica no código é feita, todas as funções são implementadas no próprio código fonte. Mantendo dessa forma a portabilidade aos mais diversos ambientes de sistemas operacionais.

Definido o conjunto de testes e suas aplicações, faz necessário saber como estas serão utilizadas no *benchmark* e para isso deve-se gerar uma série de testes de execução. Os requisitos para a série de testes de execução e como estes devem ser executados para a verificação das funcionalidades e eficiência do SOTR foram descritas pelo *benchmark Hartstone* [65] na qual representa uma forma de como os requisitos de tempo real (algoritmo, implementação de compilador, sistema operacional e componente de *hardware*) devem ser representados.

Características de tempo real como atividades periódicas, processamento aperiódico gerado por interrupções ou por intervenção de usuário, sincronização entre as mais diversas atividades, acesso a dados compartilhados, mudança de modo e distribuição de tarefas, devem ser levadas em consideração pela execução do *benchmark hartstone* já que este precisa ser o mais confiável possível, principalmente para que possa atuar em aplicações reais.

As definições utilizadas do *hartstone* são a **Série-PH** – tarefas periódicas e harmônicas e a **Série-AH** – tarefas do tipo Série-PH com processamento aperiódico. A primeira tem como objetivo fornecer requisitos de testes simples com uma carga de tarefas que são puramente periódicas e harmônicas em sua forma de execução, esta série pode representar um programa que monitore vários sensores de controle com diferentes

taxas e mostra o resultado sem intervenção do usuário ou interrupção. A base do sistema consiste em cinco *threads* periódicas em que a freqüência de cada *thread* é múltipla de toda freqüência de *thread* que seja maior. No presente trabalho foi descrita da seguinte forma (ver Figura 17):

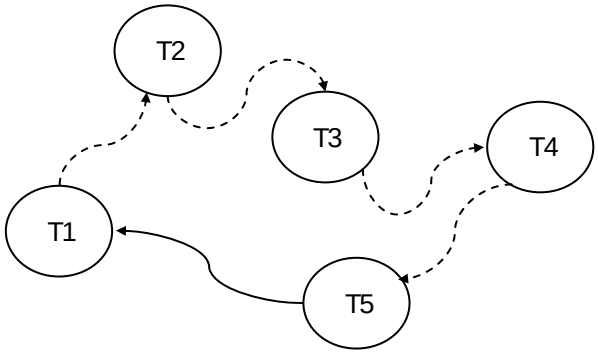


Figura 17 - Ordem de execução das tarefas de acordo com a Série-PH [65]

Considerando a Figura 17, as freqüências das *threads* T1, T2, T3, T4 e T5 são respectivamente, 16Hz, 8Hz, 4Hz, 2Hz e 1Hz. Todas as tarefas que possuem as *threads* devem ser escalonadas para iniciar ao mesmo tempo e o ciclo total deve ser de no máximo 2 segundos. Apesar de que na Figura 17 exista uma ordem de precedência entre as tarefas, quem irá determinar qual irá executar primeiro é o escalonador do sistema operacional. Cada *thread* irá conter uma das funções citadas anteriormente.

Já na Série-AH, o objetivo dos testes é determinar um conjunto de regras que representam domínios de aplicações em que o sistema responde a eventos externos. Por exemplo, um sistema que possui uma interface com usuário que é dirigida a eventos. As tarefas aperiódicas serão caracterizadas por intervalos de tempo de ativação não conhecidos, geradas aleatoriamente, e que tenham que cumprir com seus *deadlines*. O mesmo esquema de tarefas serão criados com base no esquema da série-PH porém terá agora duas tarefas aperiódicas para serem executadas em algum momento dos testes (ver Figura 18).

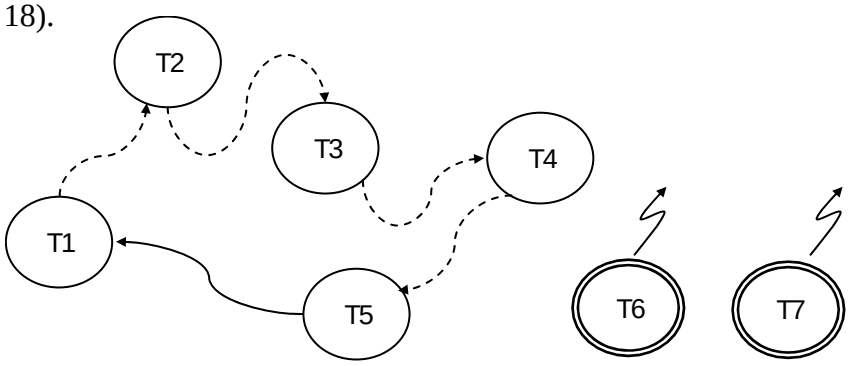


Figura 18 - Ordem de execução das tarefas de acordo com a Série-AH [65]

Considere na Figura 18 que as tarefas T6 e T7 começam a sua inicialização em um tempo gerado aleatoriamente pela aplicação. E irão executar com uma prioridade maior que as outras tarefas e seus *deadlines* devem ser de 20 ms após a sua ativação. A função que estas devem executar é da *thread* enviar um sinal para si mesma. Um manipulador de sinais foi criado e imprime na tela uma mensagem de alerta.

Definido como foi criado o *benchmark* de teste dos SOTR será mostrado a seguir a avaliação de cada sistema em estudo, o *RTlinux* 3.2 RC1, RTAI 3.3 e *Windows* CE 5.

4.2. Avaliação dos Resultados

Testes no *RTLinux*

Durante a execução do *benchmark* nos SOTR estudados podemos observar que os mesmos tiveram comportamentos distintos, porém todos atingiram as restrições impostas pelo programa. No sistema *RTLinux*, quando executado o *benchmark* não ocorreu nenhuma anomalia ao sistema operacional e todos os testes foram bem sucedidos.

Porém para não gerar mais latência aos resultados, utilizando funções de impressão na tela, estes foram dirigidos diretamente para o relatório (*log*) do sistema. Na execução da seqüência da série-PH o sistema manteve a média de execução das cinco tarefas, dentro do ciclo total ou seja em torno de 2 segundos, não teve resposta negativa nenhuma dos testes e o mesmo foi repetido durante 2 horas. Não sofrendo influência das tarefas nativas do *Linux* que se encontravam acionadas no sistema operacional.

O tempo de execução de cada tarefa também não sofreu alterações bruscas. Na série-AH também manteve a mesma ordem de grandeza na execução das tarefas. Observando as Figuras 19 e 20 podemos observar de como o sistema se comportou em ambas as séries.

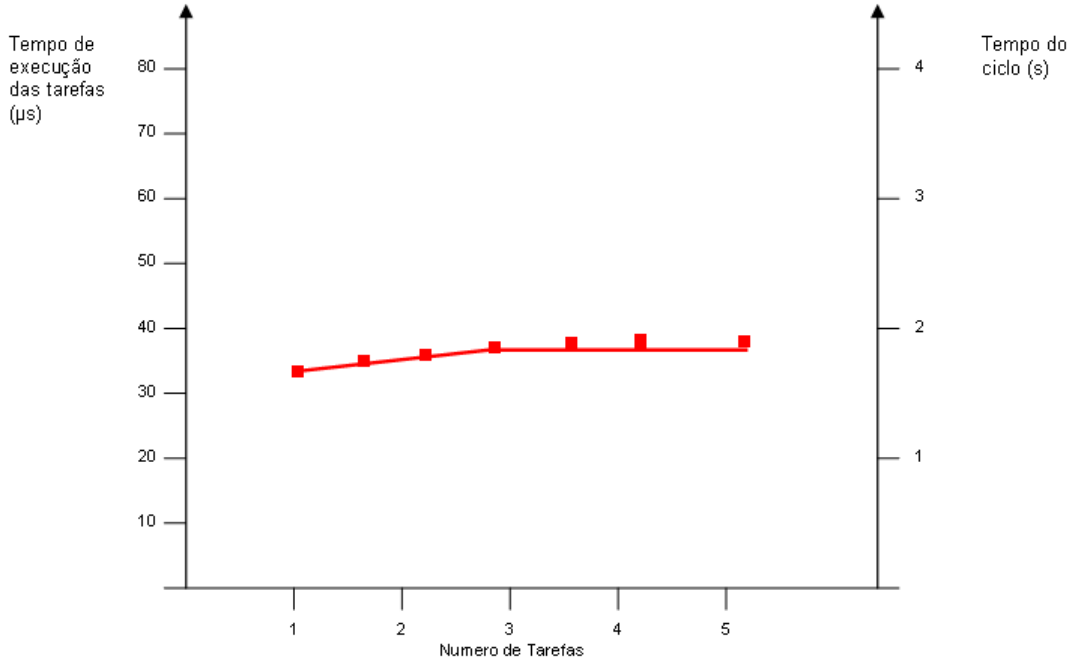


Figura 19 - Gráfico de tempo médio de execução das tarefas em microssegundo, considerando um ciclo de 2 segundos para 5 tarefas, obedecendo a Série-PH (*RTLinux*)

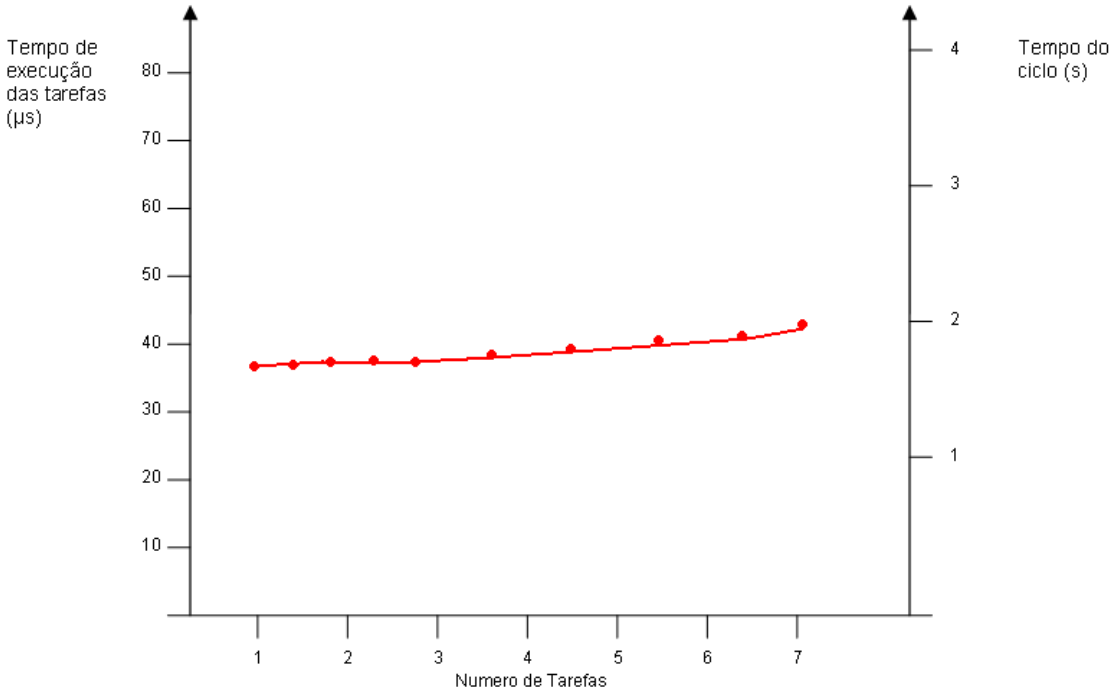


Figura 20 - Gráfico de tempo médio de execução das tarefas em microssegundo, considerando um ciclo de 2 segundos para 7 tarefas, obedecendo a Série-AH (*RTLinux*)

Os pontos nas Figuras indicam mudanças de tempo significativas ao sistema, podemos observar na Figura 19 que o tempo de execução do conjunto de tarefas fica oscilando entre os tempos de 30 µs a 40 µs, sendo que este tempo aumenta

gradualmente com o número de tarefas que estejam em execução, já na Figura 20 para um conjunto de 7 tarefas sendo 2 aperiódicas o tempo de execução foi maior que 40 µs.

Nos testes realizados, a faixa de tempo da latência do sistema operacional fica por volta de 4 µs, considerando um computador que utiliza a plataforma x86, resultado obtido através da análise por *software*.

O resultado do teste de latência está próximo ao obtido por [49], em que a faixa de tempo da latência do sistema operacional tem um valor aproximado de 2 µs, considerando um computador da mesma plataforma, resultado obtido medindo as transições dos pinos da porta paralela e de transições de relógio do *hardware*.

Testes no RTAI

No SOTR RTAI, tivemos algumas particularidades. Como forma de execução o *benchmark* foi executado em modo *kernel* assim como nos outros SOTR. Também não foram utilizadas funções de impressão na tela, estes foram dirigidos diretamente para o *log* do sistema.

Na execução da seqüência da série-PH o sistema manteve a média de execução das cinco tarefas, dentro do ciclo total ou seja em torno de 2 segundos, não teve resposta negativa nenhuma dos testes e o mesmo foi repetido durante 2 horas e também não sofrendo influencia das tarefas nativas do *Linux* que se encontravam em execução no sistema operacional.

O tempo de execução de cada tarefa também não sofreu alterações bruscas. Na série-AH também manteve a mesma ordem de grandeza na execução das tarefas. Observando as Figuras 21 e 22 podemos observar os resultados obtidos

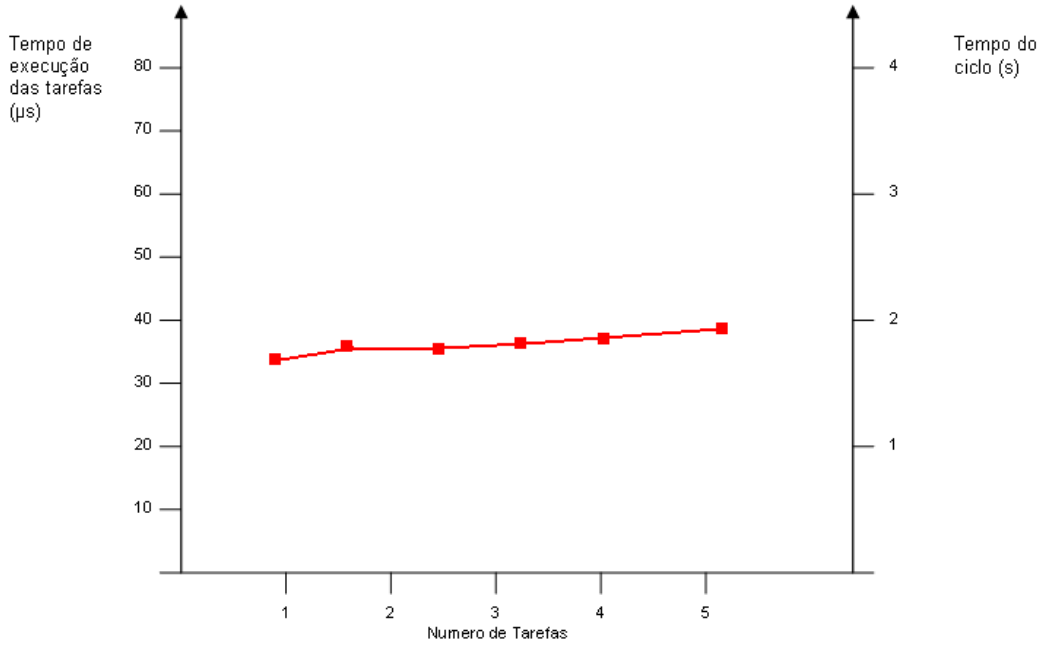


Figura 21 - Gráfico de tempo médio de execução das tarefas em microssegundo, considerando um ciclo de 2 segundos para 5 tarefas, obedecendo a Série-PH (RTAI)

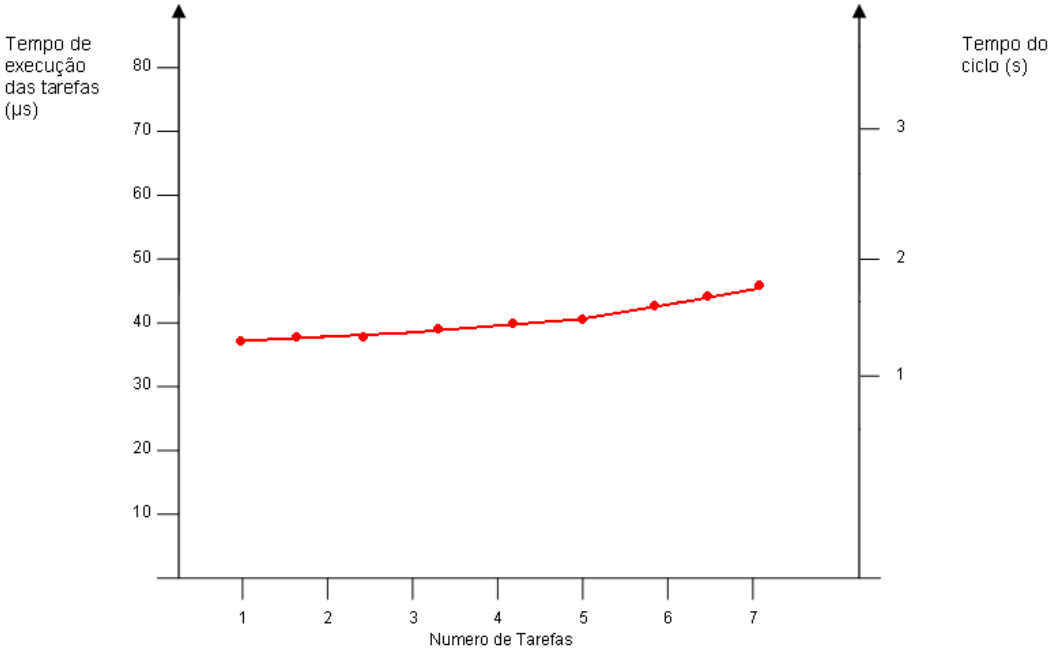


Figura 22 - Gráfico de tempo médio de execução das tarefas em microssegundo, considerando um ciclo de 2 segundos para 7 tarefas, obedecendo a Série-AH (RTAI)

Os pontos nas Figuras indicam mudanças de tempo significativas ao sistema. Na Figura 21 o tempo de execução do conjunto de tarefas fica oscilando entre os tempos de 30 µs a 40 µs, porém com resultados melhores se comparados ao do *RTLinux*, sendo que este tempo aumenta gradualmente com o número de tarefas que estejam em

execução, e na Figura 22 o tempo de execução foi maior que 40 µs, sendo que as vezes, durante a execução dos testes se aproximava dos 50 µs. Nos testes realizados, a faixa de tempo da latência do sistema operacional fica por volta de 3 µs, também considerando um computador que utiliza a plataforma x86, resultado obtido através da análise por *software*.

Podemos observar que um resultado maior do que o *RTLinux* na série-AH nos tempos de execução deve-se ao fato que o sistema tem uma definição para os tempos de latência e freqüência muito maior devido a ferramenta de calibração explicado no Capítulo 3. O resultado do teste de latência está próximo ao obtido por [49], em que a faixa de tempo da latência do sistema operacional tem um valor aproximado de 2,5 µs, considerando um computador da mesma plataforma, resultado obtido medindo as transições dos pinos da porta paralela e de transições de relógio do *hardware*.

Testes no *Windows* CE

Executando os testes no *Windows* CE a forma de execução e obtenção dos resultados foi feita de forma mais simples, já que o SOTR gera um *log* com os resultados de execução das aplicações. Utilizando o *benchmark,* na imagem gerada para a plataforma alvo, o sistema apresentou um tempo de execução das tarefas muito maior entre os sistemas operacionais estudados, sendo que o tempo do ciclo de execução se manteve dentro do tempo estabelecido, porém o sistema não se manteve estável durante todo o tempo de execução, oscilando várias vezes durante o período estimado para execução do *benchmark*.

Testes de execução nas medidas de latência por *software* realizados por [38], observou que o sistema operacional variava muito em cada execução, por isso mensurou essas variações de acordo com uma faixa de tempo entre 93 e 273 µs entre o melhor e pior caso respectivamente. As Figuras 23 e 24 mostram os tempos das séries do *benchmark*.

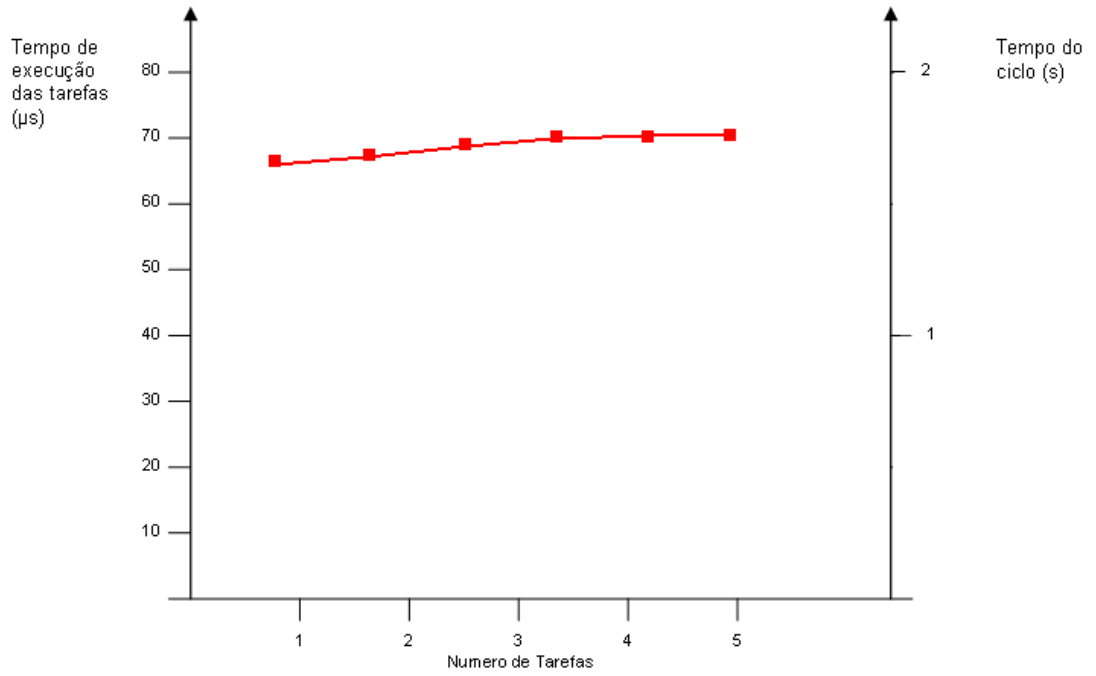


Figura 23 - Gráfico de tempo médio de execução das tarefas em microssegundo, considerando um ciclo de 2 segundos para 5 tarefas, obedecendo a Série-PH (*Windows* CE)

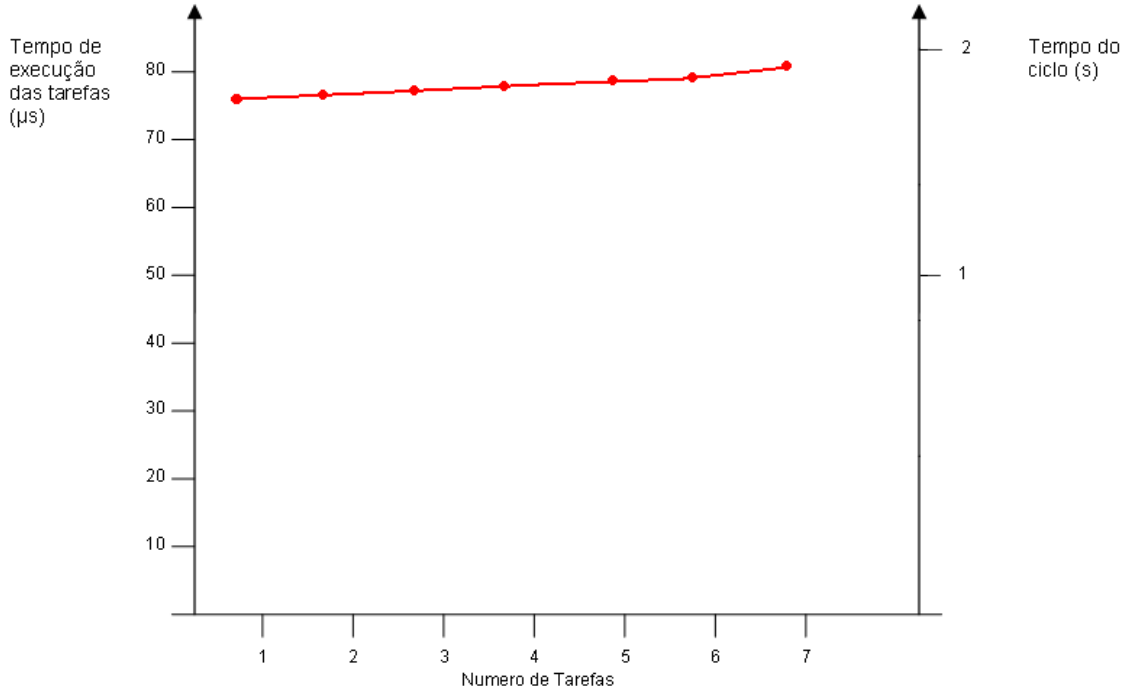


Figura 24 - Gráfico de tempo médio de execução das tarefas em microssegundo, considerando um ciclo de 2 segundos para 7 tarefas, obedecendo a Série-AH (*Windows* CE)

Podemos observar que durante a execução dos testes que a série-PH os valores encontrados ficaram entre 60 e 70 µs, e a série-AH fica próximo a 80 µs ao se aproximar do limite de tarefas estipulados no testes, esta variação de certa forma está maior que os outros SOTR em estudo no presente trabalho, mas que ainda estão satisfatórios para um sistema de tempo real. Quanto a testes de latência os valores encontrados na faixa de 10 µs e 15 µs.

4.3. Quadro comparativo do uso do *benchmark*

A seguir um Quadro comparativo com os resultados obtidos durante a execução do *benchmark* é mostrado:

Quadro 4: Resultados obtidos durante a execução do *benchmark*.

SOTR	**Tempo de Latência**	**Tempo para Série-PH**	**Tempo para Série-AH**
RTLinux	4 µs	~ 35 µs	~ 42 µs
RTAI	3 µs	~ 34 µs	~ 47 µs
Windows CE	10 µs	~ 65 µs	~ 80 µs

Pelo Quadro 4 o SOTR que apresenta os melhores resultados de execução do conjunto de tarefas é o *RTLinux* e o que apresenta os piores resultados é o *Windows* CE. Porém se considerado o RTAI com a configuração manual das resoluções de freqüência, muito provavelmente obteria melhores resultados que o *RTLinux*.

Capítulo 5 - Estudos de Caso

Neste Capítulo, serão descritas as implementações de três exemplos de diferentes situações de aplicações de tempo real, críticas ou não, em que envolvam o uso de SOTR. O fator de escolha de tais situações é que os mesmos já foram explorados por outros autores [18], [33], [54] e [56]. A representação do funcionamento de tais situações são necessárias já que mostra ao leitor do trabalho as restrições de tempo impostas e o ambiente em que é desenvolvido. A principal intenção deste Capítulo é mostrar ao leitor que SOTR estão ganhando espaço não somente em universidades como também em outras áreas como na indústria e em aplicações governamentais.

5.1. Sistema de Tráfego Aéreo

CTA fornece um bom exemplo por inúmeras razões. Em primeiro lugar, esses sistemas são certamente aplicações importantes, conhecidas e difíceis, além disso, há inúmeros deles por todo o mundo. Também encontramos instancias de todas as características distintas de sistemas de tempo real.

A aplicação em questão é simplificada e consiste em rastrear, em um espaço aéreo monitorado, todas as aeronaves e assegurar que cada uma destas mantenha sempre uma distancia mínima de separação das outras, além de ter sua rota prevista. Um radar rastreia a posição de cada aeronave no espaço. Quando uma aeronave entra em um espaço, sua presença é anunciada através de um subsistema de comunicação digital. Ao deixar o espaço, o sistema de computação irradia a notícia de sua partida para as aeronaves vizinhas.

O sistema de comunicação pode também usado para comunicar-se diretamente com a aeronave. Um terminal com teclado e monitor faz a interface com um operador humano, nele é exibido o rastro de cada aeronave e ele responde a comandos do operador, que incluem opções para interrogar o estado e os dados descritivos de uma aeronave e para transmitir mensagens, por exemplo, para que aeronaves alterem sua direção, a fim de evitar colisões.

O radar pode varrer qualquer porção do espaço aéreo, dadas as coordenadas do espaço de busca. Se o radar detecta um objeto na posição informada, ele retorna um acerto (*hit*). Uma aeronave é considerada perdida se o radar falha em produzir um acerto. Nesse caso, o operador deve ser notificado e uma ação corretiva deve ser tomada. O radar é também usado para varrer o espaço em busca de objetos desconhecidos e aeronaves perdidas.

As restrições temporais razoáveis para manipular os vários dispositivos e o ambiente são:

- O radar deve rastrear cada avião no espaço a razão de, no mínimo, uma observação a cada 1 s por avião;
- A posição e o trajeto de cada avião devem ser atualizados e exibidos, no mínimo, uma vez a cada 300 ms;

- Uma resposta a um comando de operador deve ser fornecida em até 500 ms.

A Figura 25 mostra parte de uma possível implementação do exemplo de CTA, em termos de processos periódicos e esporádicos. A estrutura de dados mais importante é um *Arquivo de rota* que é compartilhado por vários processos, tempo, velocidade e identificação de cada aeronave no espaço. As setas na Figura representam fluxo de dados ou comunicação.

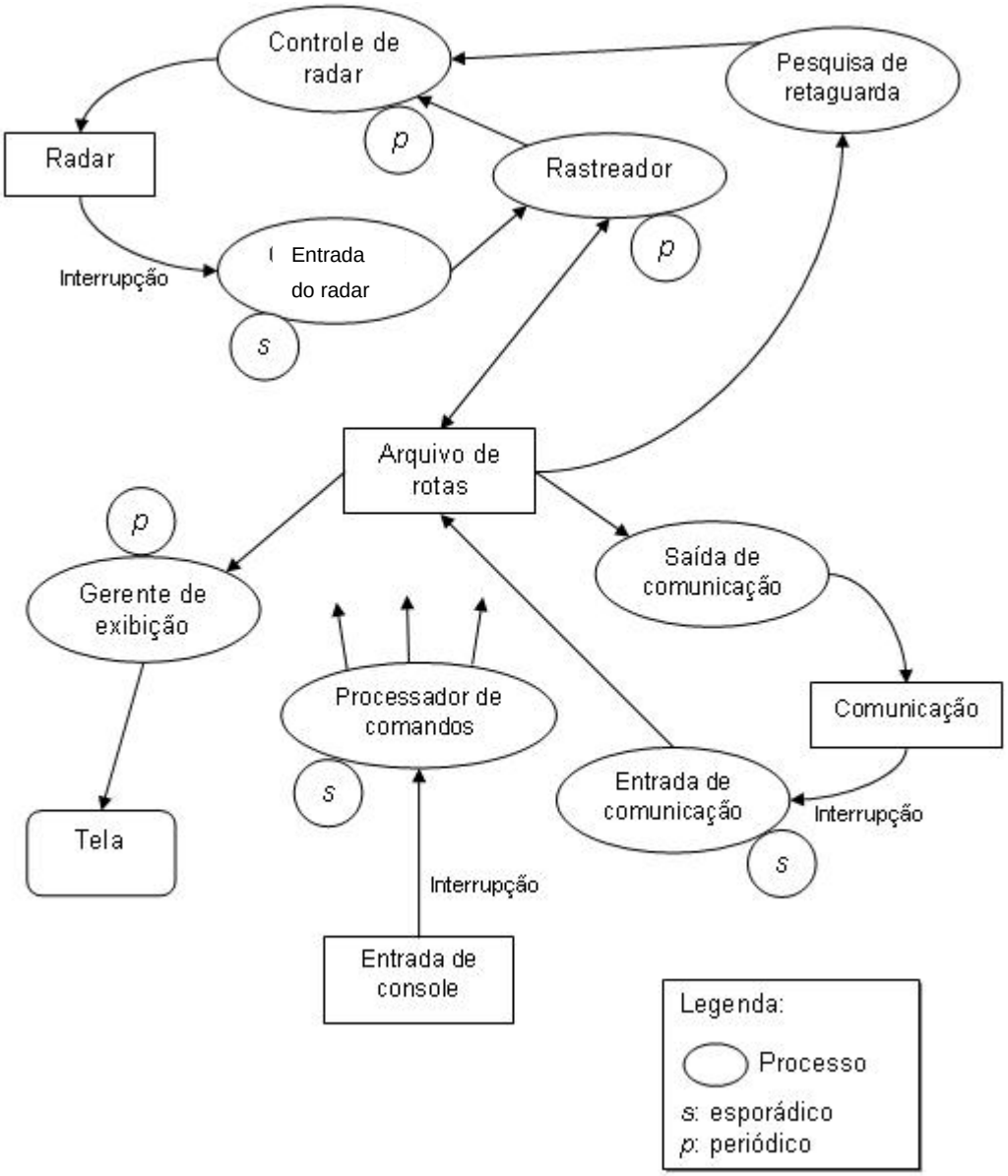


Figura 25 - Projeto parcial de um sistema de CTA [56]

Os principais objetivos desse sistema são a segurança, eficiência e desempenho. Devem-se prevenir colisões e a ocorrência de outros desastres. Aplicando as regras descritas anteriormente e comparando com os SOTR escolhidos no estudo, nenhum se torna adequado totalmente para este tipo de função.

A limitação do sistema *RTLinux* está no fato de que vários processos estarão concorrendo ao mesmo tempo, e como todos executam em modo *kernel* uma falha em um desses processos compromete o sistema todo, também podemos citar que como o sistema necessita de cálculos aprimorados de rotas e localização, o uso de operações com ponto flutuante é necessária.

Mesmo o sistema podendo se adequar a essa situação, a carga gerada pelo módulo para tratar esse tipo de operação pode aumentar a probabilidade de falhas do SOTR com outras estruturas. Um ponto forte do *RTLinux* é possuir um sistema de escalonamento para situações que envolvam tarefas esporádicas.

O RTAI poderia ser indicado para desempenhar esta função, porém a grande redundância de funções na API e operações no SOTR, acarretariam muito tempo por parte do projetista em contornar determinadas situações e fazer o sistema funcionar corretamente. Muito tempo seria perdido durante a fase de projeto do sistema.

Outro fator limitante do RTAI é que poderia falhar em situações em que as interrupções fossem usadas. Como a camada HAL possui muito mais funções de controle além de interceptar as interrupções do *Linux*, as interrupções de *software* geradas poderiam aumentar o *overhead* no sistema e conseqüentemente mais tempo não previsível ao projeto.

A vantagem de tal sistema é que os processos podem ser executados em modo usuário, garantido uma proteção ao sistema operacional e o recurso de rastreamento de ações das tarefas de tempo real criadas pelo LTT é uma excelente ferramenta para análise de comportamento.

No que diz respeito ao *Windows* CE fatores como controle de memória usando paginação não é o ideal, porém o mesmo pode ser contornado usando o recurso de *heap*. No entanto como o sistema de CTA exige que um número considerável de tarefas estejam executando. No SOTR em questão isso não é adequado, já que se limita a executar apenas 32 processos simultaneamente, como foi explicado no Capítulo 3. No entanto o *Windows* CE é o sistema mais leve no presente estudo e que pode trazer

benefícios quando colocado na plataforma, diminuindo os acessos a estruturas desnecessárias no *kernel*.

5.2. Sistema de Controle Industrial

Sistemas de automação industrial e de manufaturas usam uma vasta gama de sistemas de tempo real, indo de peças robóticas até controles de fabricação e sistemas de monitoramento por sensores. O número de dispositivos e subsistemas envolvidos é bastante grande e esses são diferentes em sua natureza.

O presente caso é um exemplo típico usado para demonstrar como o uso de sensores podem ser utilizados, controlados por um sistema operacional de tempo real. Esse exemplo utiliza um tanque de armazenamento de combustíveis com sistemas críticos para monitorar a concentração e temperatura do líquido contido no mesmo. A descrição do problema pode ser observada na Figura 26.

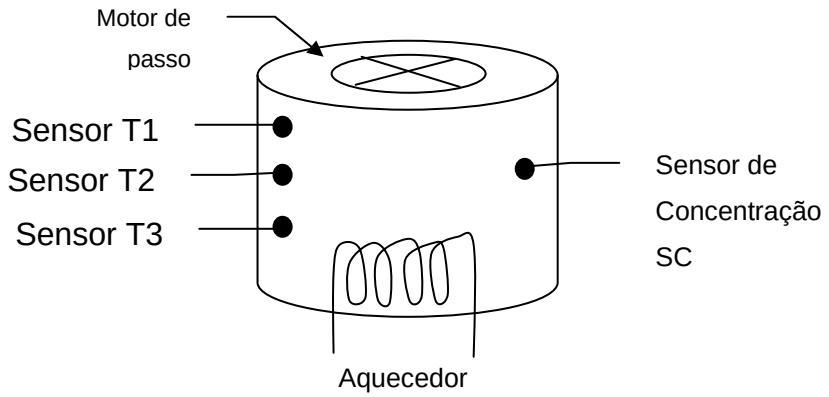


Figura 26 – Tanque de armazenamento de combustível controlado por sensores [54]

O tanque tem os seguintes elementos:

- Três sensores de temperatura (T1, T2 e T3) localizados em diferentes pontos de altura do tanque, sendo que cada sensor tem uma escala de valores entre -10°e 40°. A operação de leitura do sensor leva 100 ms;
- Um sensor de concentração (SC), provendo valores entre 1 e 10 e também leva 100 ms a operação de leitura efetuada pelo sistema;
- Um aquecedor que pode ser ligado ou desligado de acordo com o comportamento do sistema;

- O depósito também possui um motor de passo que pode ser ligado ou desligado também de acordo com o comportamento.

Em qualquer situação do problema, manter a temperatura do tanque é a tarefa mais prioritária para auxiliar nessa tarefa deve-se ligar ou desligar o motor. O comportamento do sistema obedece aos seguintes critérios: Caso o aquecedor estiver ligado, a temperatura no tanque aumenta 1° a cada 3s para o sensor T1, 1° a cada 2s para o sensor T2 e 1° a cada 1s para o sensor T3. Por outro lado caso o aquecedor esteja desligado a temperatura no tanque diminui 1° a cada 4s para o sensor T1, 1° a cada 8s para o sensor T2 e 1° a cada 10s para o sensor T3. O valor de concentração encontrado no líquido do tanque aumenta em 1 unidade a cada 2s caso o motor esteja ligado e se o motor estiver desligado a concentração diminui em 1 a cada 5s.

Por padrão o motor de passo é desligado e ficará em funcionamento caso o operador do sistema de controle ligá-lo, e este recebe uma requisição a cada 200 ms para verificar se está funcionando o motor. Se caso o motor esteja funcionando por um tempo muito grande este será desligado automaticamente toda vez que a concentração no tanque estiver muito grande.

O aquecedor também por padrão é desligado e irá ficar em funcionamento toda vez que a temperatura no interior do tanque atingir um valor elevado acima de 30°. O sistema de controle do tanque deverá manter o comportamento da temperatura e concentração dentro de certos limites. A temperatura deverá ser entre 3° e 10° a cada momento. A diferença entre um sensor e outro mais abaixo não deverá ser diferente que 3°. Se isso ocorrer o motor de passo deve ser iniciado para misturar o combustível. Já a concentração deverá está entre 3 e 7, caso não esteja o motor também entrará em funcionamento controlando a concentração.

O sistema de controle deve ler a temperatura dos sensores com um período de 2s e o sensor de concentração a cada período de 4s. Sendo que o sistema de controle deve atuar no motor de passo a cada 2s e no aquecedor a cada 1s. O tempo de processamento dos sensores de temperatura e concentração é de 400 ms (leitura mais computação) e o tempo de atuação para o aquecedor é de 500 ms e do motor de 600 ms.

Precisão na execução das tarefas é a chave para o bom funcionamento do sistema. O sistema RTAI é o que mais se adequou ao ambiente descrito já que se as tarefas de tempo real venham a sofrer alguma falha o sistema como um todo não irá parar. Por outro lado caso usássemos o sistema *Windows* CE pode-se fazer uso do

recurso de alocação fixa na memória e desabilitar as alocações dinâmicas fazendo com que o sistema operacional ganhe mais previsibilidade. O sistema *Windows* CE ainda fornece uma imagem pequena e leve para controlar o ambiente, além de ser fácil de configurar utilizando o *Platform Builder* para obedecer a descrição da aplicação de tempo real.

5.3.Sistema de Controle Robótico

Aplicações de robótica fornecem uma forte ferramenta de validação dos componentes de SOTR, já que consiste de vários dispositivos comunicantes entre si e alguns com controles independentes que fazem uso de gerenciamento dinâmico de memória até processo de comunicação entre processos como semáforos, barreiras e rede. Este exemplo não pretende fornecer um comportamento realístico e não faz uso de todos os recursos de comunicação dos SOTR que possa a vir utilizar, sendo que ele utiliza a possibilidade de verificação da modificação do escalonador, barreiras e se o mesmo pode ser utilizado entre sistemas diversos.

A aplicação consiste de um sistema computacional entre um braço robótico e um motor ambos controlados por SOTR e a comunicação entre os sistemas é feito através de uma rede *Ethernet*, assim como observado na Figura 27.

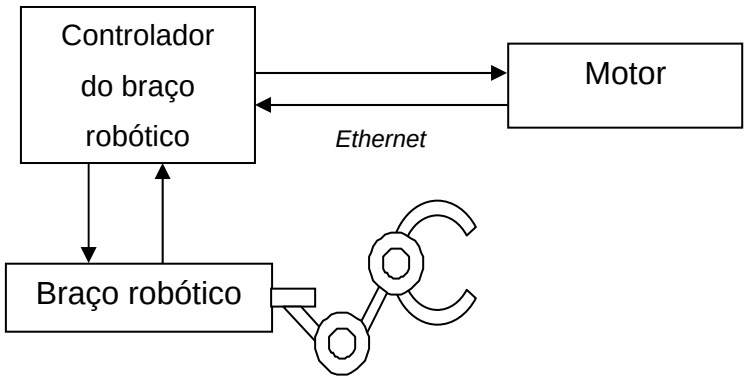


Figura 27 - Descrição do sistema robótico [33]

O usuário do sistema pode interagir com o mesmo usando um ambiente de controle localizado no motor, sendo que para inicializar o braço robótico deve-se usar um sistema interativo que verifique a correta inicialização através de mensagens ao usuário e o motor é inicializado verificando se existe a sincronização entre o mesmo e o controle do braço. Caso falhe na verificação mensagens de erro devem ser mostradas ao usuário.

No ambiente do motor uma tela deve desenhar o atual estado do braço e se este se encontra próximo a algum obstáculo. As tarefas no controle do braço são ignoradas e leva-se em conta apenas a forma de comunicação e se o ambiente pode utilizar um SOTR para controle. A verificação deve ser a cada 20 ms e a sincronização entre os componentes envolvidos não deve ser maior que 30 ms.

Poderia ser utilizado o SOTR *Windows* CE para controlar o ambiente de motor já que fornece uma API gráfica bastante rica e de fácil desenvolvimento, também podendo contar com uma interface de rede bem desenvolvida. Operações com precisão também é mais um ponto a favor para utilização o *Windows* CE. Outros sistemas como o *RTLinux* ou o RTAI não se mostram tão eficientes para controle do motor já que fornecem uma API com poucos recursos gráficos e no sistema *RTLinux* limitação quanto a precisão.

Para o controlador do braço robótico pode ser utilizado o *RTLinux* já que possui uma API concisa e estável tanto de recursos de comunicação quanto de recursos de alocação de memória, também a forma de modificar o escalonador do sistema ser bem simples. Também o SOTR fornece baixa latência e rápida troca de contexto entre as tarefas, a resolução em nanosegundos dos temporizadores pode ser bastante úteis para este cenário.

5.4. Quadro comparativo do Estudo de Caso envolvendo os SOTR analisados

Quadro 5: Comparação entre os SOTR de acordo com vantagens e desvantagens na implementação do Estudo de Caso.

Estudo de Caso	***RTLinux***	**RTAI**	***Windows* CE**
Sistema de Tráfego Aéreo.	**Desvantagens** – Processos executando em modo *kernel* as vezes interromper o sistema; falta de precisão em comparação a outros SOTR em estudo. **Vantagens** – Possuir sistema de escalonamento para tarefas aperiódicas.	**Desvantagens** – Redundância de determinadas funções da API dificultando desenvolvimento; interrupções do sistema hospedeiro (*Linux*) geram tempo não previsível; **Vantagens** – Processos executando em modo usuário não interrompem o sistema; facilidade de análise de	**Desvantagens** – Uso de paginação no SOTR pode gerar tempo não previsível; limitado número de tarefas em execução. **Vantagens** – SOTR compacto e específico para a plataforma alvo; grande facilidade de implementação (uso da API *win32*) e criação da imagem (*Platform*

		comportamento (uso da ferramenta LTT).	*Builder*).
Sistema de Controle Industrial.	**Desvantagens** – Processos executando em modo *kernel* as vezes interromper o sistema. **Vantagens** – Desenvolvimento rápido usando *threads* no padrão POSIX.	**Desvantagens** – Redundância de determinadas funções da API dificultando desenvolvimento; interrupções do sistema hospedeiro (*Linux*) geram tempo não previsível; **Vantagens** – Processos executando em modo usuário não interrompem o sistema; uso de recursos de rede.	**Desvantagens** – Uso de paginação no SOTR pode gerar tempo não previsível; **Vantagens** – SOTR compacto e específico para a plataforma alvo; grande facilidade de implementação (uso da API *win32*) e criação da imagem (*Platform Builder*); alocação dinâmica de memória (proteção das tarefas); uso de recursos de rede.
Sistema de Controle Robótico.	**Desvantagens** – Motor: API gráfica de difícil desenvolvimento; falta recursos de comunicação com garantias de previsibilidade; Braço: falta de precisão (operações com ponto flutuante). **Vantagens** – uso de *pthreads* (POSIX) facilita o desenvolvimento do Braço; baixa latência e rápida troca de contexto.	**Desvantagens** – Motor: API gráfica de difícil desenvolvimento; Redundância de determinadas funções da API; interrupções do sistema hospedeiro (*Linux*) geram tempo não previsível; **Vantagens** – Processos executando em modo usuário; uso de recursos de rede; Braço: fácil conexão com outros SOTR que utilizam recursos de rede.	**Desvantagens** – Motor: sistema com alta latência; uso de paginação. **Vantagens** – Motor: API gráfica favorece o desenvolvimento de uma interface amigável ao usuário; uso de recursos de rede e conectividade; sistema compacto.

Capítulo 6 - Conclusões

Este capítulo apresenta as considerações finais com as principais contribuições resultantes do desenvolvimento deste trabalho. Além das dificuldades encontradas durante a avaliação dos SOTR. Por fim, são expostas as principais sugestões para trabalhos futuros que estendam o trabalho apresentado.

6.1. Observações sobre os Sistemas Operacionais de Tempo Real

Nos SOTR que foram estudados podemos notar que tanto o ambiente do *RTLinux* quanto o RTAI são limitados quanto a serviços do padrão POSIX e que são primordialmente ligados a ambientes de gerenciamento de E/S, já que a camada de tempo real faz uso do controle de interrupções para utilização de aplicações de tempo real. O mesmo não ocorre no sistema *Windows* CE, que para utilização de recursos de tempo real aplica soluções ligadas a própria plataforma alvo, mas de certa forma limitante já que o sistema não possui nenhuma garantia que os serviços irão obedecer a limitações temporais de tempo real.

Sincronização e operações no ambiente entre as *threads* são disponíveis em todos os SOTR em estudo e na maioria dos SOTR comerciais também. Os ambientes baseados em *Linux* podem enviar e receber dados através de serviços do sistema hospedeiro, mas acesso a disco, arquivos, sistemas de rede e acessos de E/S de dispositivos não são acessíveis totalmente na camada de tempo real. O mesmo já não ocorre com o *Windows* CE, que, de certa forma, por causa das facilidades encontradas na API do *Win32*, possui acessos a estes objetos, o que nos outros sistemas operacionais não é possível.

Também podemos observar que os SOTR não são apropriados para hospedar aplicações de tempo real que sejam muito complexas, como por exemplo, um sistema de simulador de vôo completo entre outras. Por isso os estudos em cima de sistemas de *benchmark* são importantes, pois verifica se os SOTR em questão podem ser apropriados para um determinado tipo de aplicação ou não. Uma forma de verificar esse tipo de aplicações em ambientes de tempo real é usar aplicações críticas em conjunto com aplicações normais e verificar se as críticas sofreram interferência das não críticas, como apresentado no Capítulo 4 na execução da Série-AH.

Um fator importante observado durante a execução dos testes é que os sistemas *RTLinux* e RTAI reduzem o tempo na ocorrência de interrupções, e estes sistemas disponibilizam um rico ambiente de gerenciamento de E/S muito maior do que o sistema operacional *Windows* CE. Porém a habilidade de resolver problemas de requisitos de tempo real não se limita apenas em adaptar um sistema *Linux* para funcionar com um melhor gerenciamento de interrupções. Outras características como controle de disco, dispositivos e recursos de rede se fazem necessárias, e ter um controle temporal também. Isso é encontrado nativamente no sistema *Windows* CE, porém com

limitações que nem sempre obedecem aos requisitos temporais impostos pelo desenvolvedor.

Observamos também que quanto à compatibilidade com outros padrões o que observamos é que a interface POSIX que é encontrada nos sistemas baseados em *Linux* permite as aplicações serem portadas para outros sistemas sem a necessidade de ficarem atreladas a um determinado SOTR. Mesmo que a compatibilização, em muitos casos, como no *RTLinux* e RTAI não sejam totalmente compatível com o padrão POSIX, a maioria das funções de tempo real podem ser utilizadas sem muitos problemas de conversão em outros ambiente. Exemplo disso é a transcrição de aplicações do *RTLinux* para o SOTR *VxWorks*.

Outro fator que é importante citar e que foi observado é o uso de estruturas com ponto flutuante. Em aplicações de controle, usando um sistema computadorizado, existem muitos casos em que podem ser úteis o uso de operações com ponto flutuante, principalmente em gerenciamento de interrupções. Casos desse tipo são encontrados em controles digitais, dos mais simples aos mais complexos, que requerem um comportamento periódico com alguma forma de comunicação com as interfaces de controle.

Nessa situação pode existir um *overhead* por parte do escalonador, gerado principalmente por causa dos temporizadores/relógios e sua granularidade. Em SOTR baseados em *Linux*, operações de pontos flutuantes ainda são feitas pelo sistema hospedeiro que usa *traps* toda vez que um processo necessita, gerando um tempo não previsível. Notamos esta situação no sistema *RTLinux* que, por padrão, não permite operações deste tipo mas, caso necessite, o sistema operacional oferece uma função específica para isso.

Isso de alguma forma traz malefícios ao desenvolvedor já que tem de preocupar-se com o uso de operações com ponto flutuante em suas aplicações. No caso dos outros dois sistemas operacionais, RTAI e *Windows* CE, isto não ocorre, já que estes usam estruturas e operações de ponto flutuante de forma nativa.

Outro fator de importância no sistema *Windows* CE é que todas as primitivas de proteção do SOTR utilizam herança de prioridade. Isso ajuda a prevenção de falhas de sincronismo entre as tarefas de tempo real. Esse comportamento seria desejável em todos os outros sistemas operacionais estudados. Nota-se que o sistema *Windows* CE se torna limitado já que este permite um número pequeno de processos, comparado aos outros sistemas operacionais, e uso de memória virtual com paginação sem controle

efetivo da previsibilidade, que pode ocasionar limitações no uso de aplicações de tempo real.

Também se nota o alto tempo de latência e interrupção por parte do *Windows* CE em comparação aos outros dois SOTR, porque durante a alocação dos dispositivos na memória, o *kernel* manipula a tabela de paginação, da memória virtual, aumentando dessa forma o tempo para carregar as aplicações de tempo real.

Observa-se também no *Windows* CE uma restrição a um conjunto específico de dispositivos. Enquanto que gradativamente o número de dispositivos vem crescendo mundialmente, o SOTR exige que o projetista selecione de uma lista restrita qual dispositivo utilizar. Qualquer dispositivo ou periférico não suportado pelo SOTR deve ser desenvolvido, aumentando assim o tempo de projeto.

6.2.Considerações e trabalhos futuros

A dissertação apresentada compara algumas características pouco analisadas em trabalhos que envolvam SOTR utilizando a observação do comportamento dos mesmos e o uso de um programa de *benchmark* para avaliar a resolução e execução das tarefas envolvidas. O *benchmark* desenvolvido é restrito apenas em algumas características mas, como mencionado anteriormente, tais características avaliadas são de extrema importância para um sistema de tempo real já que trata da previsibilidade do mesmo.

O *benchmark* é apenas um processo sistemático e contínuo de avaliação não podendo ser considerado um método aleatório de apenas recolher informações, mas sim um processo sistemático e estruturado com o objetivo de verificar um determinado SOTR. Então o presente trabalho comparou três dos principais SOTR utilizados em projetos acadêmicos e comerciais examinando características que podem dar valiosos resultados aos desenvolvedores de aplicações de tempo real.

Tempos envolvidos durante a latência e troca de contexto de tais sistemas operacionais são importantes de serem conhecidos visto que estes são fatores muito utilizados por projetistas ainda na fase de análise em tempo de projeto. É fato que outros tipos de latência não explorados pelo atual trabalho devam ser levados em consideração e não descartados pelos projetistas para criação de sistemas mais previsíveis, já que estes causam um grande impacto no comportamento de sistemas de tempo real dependendo da aplicação.

A previsibilidade, como foi observado, deve ser mantida independente da quantidade de recursos que o SOTR venha a ter e isto deve claro ao projetista da aplicação. Isso significa, de outra forma, que o sistema deve ter o mesmo comportamento e que as tarefas hospedadas mantenham as restrições temporal impostas, independentes da carga do sistema, o tamanho das filas de tarefas do sistema e o número de eventos simultâneos detectados.

Algumas tecnologias de tempo real, por mais testadas e estáveis que se encontrem, ainda não são empregadas de forma total ou parcial nos SOTR, já que o mercado de sistema operacional deste tipo é bem conservador e novas teorias são apenas aceitas quando ocorre a quebra de paradigmas. Nota-se mais evidente isso no *RTLinux* que utiliza uma política de escalonamento baseada no EDF e tratamento de tarefas aperiódicas. O mesmo não é encontrado nos SOTR RTAI e *Windows* CE.

O acréscimo de recursos de tempo real aos sistemas *Linux* é uma alternativa de baixo custo para muitas aplicações que utilizem tais restrições temporais. Tanto o *RTLinux* quanto o RTAI estão comprometidos para serem pequenos e utilizarem a alta funcionalidade dos recursos da comunidade de desenvolvimento *Linux*. Tais sistemas operacionais permitem multitarefa/*multithread* além dos recursos de tempo real.

Já no sistema *Windows* CE mostra-se um ambiente muito modularizável permitindo aos desenvolvedores e projetistas um ambiente realmente integrado de desenvolvimento, possibilitando a execução e o acompanhamento das tarefas de forma dinâmica antes que o mesmo possa ser utilizado na plataforma. Isso se deve ao grande volume de recursos investido neste SOTR que vem ganhando espaço em diferentes mercados de sistemas embarcados.

Uma questão interessante no presente trabalho foi observar que o *Windows* CE pode ser utilizado em aplicações críticas, já que pontos como latência se mostram bem reduzidos e este fato possibilita que o mesmo seja utilizado em uma série de aplicações, porém não tão complexas.

O presente estudo mostra como diferentes SOTR se comportam, cada um com sua particularidade que destaca ou inibe o uso dos mesmos em aplicações de tempo real. Podemos inferir também que alguns SOTR se mostram mais adequados a alguns tipos de aplicações que outros e que a portabilidade entre os diferentes ambientes é quase utópica, já que o padrão POSIX por si só não é uma condição suficientemente necessária para que a portabilidade possa ser utilizada.

É muito difícil para o projetista de tempo real detalhar a aplicação sem conhecer minuciosamente o SOTR que irá conter essa aplicação e sem conhecer o real comportamento do sistema operacional. Para auxiliar nesse tipo de análise é necessário que ele tenha em mente o que levar em consideração na avaliação do sistema operacional, para isso que é importante a definição de regras, já que por mais diferentes que os SOTR venham a serem, as características podem ser agrupadas seguindo certas regras.

Também foi demonstrado que é possível alterar o escalonador dos sistemas operacionais estudados e incorporar de novas políticas de escalonamento. Alguns trabalhos acadêmicos já desenvolvem tais alterações e possibilitam novos recursos ao escalonador, como é o caso da iniciativa européia OCERA, que trabalha modificando de forma a incorporar características até então não abordadas ao sistema *RTLinux*.

Uma extensão do presente trabalho é a criação de um sistema de *benchmark* que envolva outras características de latência e que possa ser utilizado como uma aplicação simulada de tempo real, fazendo assim uma melhor observação por parte dos projetistas, e possibilitar levar em consideração outros aspectos na utilização de tais sistemas operacionais. Uma avaliação maior também poderia ser feita no presente trabalho, envolvendo outros sistemas comerciais como o QNX ou o *VxWorks*, comparando com as iniciativas abertas encontradas.

Um outro trabalho futuro é realizar a mesma pesquisa em diferentes plataformas como SH, MIPS ou *Power* PC, assim um comparativo mais abrangente pode ser definido e a utilização de tais plataformas mostrará uma analise mais ampla, favorecendo a criação de aplicações de tempo real mais precisas e robustas. É bom citar que a previsibilidade está diretamente ligada ao tipo de processador que a aplicação irá executar. Outras técnicas de avaliação do tempo através de recursos de *hardware* podem ser utilizadas, aprimorando os resultados encontrados. Os testes também poderiam ser feitos mediante a modificação dos escalonadores seguindo diferente políticas.

O presente trabalho pode ser uma referência na análise de SOTR tanto envolvendo *Linux* de tempo real quanto iniciativas comerciais, auxiliando de forma direta os projetistas e desenvolvedores de aplicações de tempo real.

Referências Bibliográficas

[1]ANDERSSON, B., ABDELZAHER, T., JONSSON, J., ***Global Priority-Driven Aperiodic Scheduling on Multiprocessors***, *Proceedings of the 17th International Symposium on Parallel and Distributed Processing, Ed. IEEE Computer Society*, pp. 8, 22-26 de Abril de 2003.

[2] BAKER, T. P., ***An analysis of EDF schedulability on a multiprocessor***, *Parallel and Distributed Systems, IEEE Transactions,* vol. 16, no. 8, pp. 760-768, Agosto de 2005.

[3] BARABANOV, M. ***A Linux-based Real-Time Operating System***, *Thesis submitted for the degree of Master in science in Computer Science, New Mexico Institute of Mining and Technology*, Socorro, Novo México, EUA, 43 p., Junho de 1997.

[4] BARABANOV, M., ***Getting Started with RTLinux***. FSM Labs, Inc., 2001, Disponível em http://www.fsmlabs.com. Acessado em Outubro de 2005 .

[5] BARNES, J. ***Rationale - Tasking and real-time***. Disponível em: http://www.adacore.com/home/ada_answers/ada_2005. Acessado em 18 de novembro de 2005.

[6] BARUAH, S. K., COHEN, N. K., PLAXTON, C. G., VARVEL, D. A. ***Proportionate progress: a notion of fairness in resource allocation***. *San Diego, California, USA*: *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, vol. 15, pp 600-625, 1993.

[7] BERNAT, G., COLIN, A., PETTERS, S.M. ***WCET Analysis of Probabilistic Hard Real-Time System***. *IEEE Real-Time Systems Symposium, 23rd*, pp. 279-288, 3-5 de Dezembro de 2002.

[8] BIHARI, T.; SCHWAN, K. ***Dynamic adaptation of real-time software. ACM Transactions on Computer Systems***, vol. 9, no. 2, *College of Computing, Georgia Institute of Technology*, Atlanta, Georgia, EUA, pp. 143-174, Maio de 1991.

[9] BURNS, A. ***Real-Time Systems Scheduling***. *Technical Report YCS 134, Department of Computer Science, University of York, UK*, pp. 61-93, 1992.

[10] BURNS, A., WELLINGS, A., ***Real-Time Systems and Programming***

***Languages (Third Edition) Ada 95, Real-Time Java and Real-Time POSIX*.** Ed. Addison Wesley Longman, 611 p, Março de 2001.

[11] CAO, Q., STANKOVIC, J. A. ***Real time properties: Dual face phased array radar scheduling with multiple constraints***. *Proceedings of the 5th ACM international conference on embedded software* EMSOFT '05. ACM Press. *Jersey City, NJ, USA*, pp. 361-370, Setembro de 2005.

[12] CHART, T., RACE, N., METAXAS, G., SCOTT, A., ***Experiences with Windows CE and Linux***. *Computing Department, Lancaster University, Lancaster, UK, Technical Report*, 10 p, 2003.

[13] DANKWARDT, K. ***Comparing real-time Linux alternatives***. Artigo publicado na Linuxdevices.com em 11 de outubro de 2000. Disponível em: http://www.linuxdevices.com/articles/AT4503827066.html. Acessado em: 20 de agosto de 2005.

[14] ***Dedicated Systems Encyclopedia***. Acessado em 15 de novembro de 2005. Disponível em: http://www.omimo.be/encyc

[15] DING, H.; ZHENG, C.; AGHA, G.; SHA, L.; ***Automated verification of the dependability of object-oriented real-time systems***, *Object-Oriented Real-Time Dependable Systems, Ninth IEEE International Workshop on Real-Time Systems*, pp. 171–178, Outubro de 2003.

[16] DUDA, K. J., CHERITON, D. R. ***Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose system***. SOSP, *Charleston, South Carolina, USA*, vol. 33, no. 5, pp. 261-276, Dezembro de 1999.

[17] FARINES, J., FRAGA, J.S. e OLIVEIRA, R.D. **Sistemas de tempo real, Escola de Computação**, Florianópolis, 2000.

[18] FENG, X., MOK, A. K., ***Real-Time Virtual Resource: A Timely Abstraction for Embedded Systems***, *Lecture Notes in Computer Science, vol* 2491, *Genoble, France*, pp. 182-196, Outubro de 2002.

[19] FSMLABS. ***Literature – Archive***. Disponível em: http://www.fsmlabs.com/literature.html. Acessado em 05 de setembro de 2005.

[20] FSMLABS. ***RTLinux Free – A overview***. Disponível em http://www.fsmlabs.com/rtlinuxfree.html. Acessado em 05 de setembro de 2005.

[21] ***General Purpose Platform, VxWorks Edition, Product Overview***. Acessado

em: 15 de novembro de 2005. Disponível em: http://www.windriver.com/products/product-overviews/General-Purpose-Platform-ve-overview.pdf

[22] GHOSH, K.; MUKHERJEE, M. SCHWAN, K., ***A Survey of Real-Time Operating Systems***, *College of Computing – Georgia Institute of Technology*, Atlanta, Georgia, EUA, pp. 1-62, Fevereiro de 1995.

[23] GUTHAUS, M.; RINGENBERG, J.; ERNST, D.; AUSTIN T.; MUDGE, T.; BROWN, R., ***MiBench: A free, commercially representative embedded benchmark suite***, *Proceedings of the IEEE International Workshop on Workload Characterization*, vol. 00, ed. IEEE Computer Society, Washington, EUA, pp. 3-14, 2001.

[24] ***How Windows CE .NET is Designed for Quality of Service***. *Technical Articles, Microsoft Corporation*, Fevereiro de 2003. Acessado em: 8 de novembro de 2005. Disponível em: http://msdn2.microsoft.com/en-us/library/ms836770.aspx

[25] JEJURIKAR, R., GUPTA, R. ***Procrastination scheduling in fixed priority real-time systems***. *Proceedings of Language Compilers and Tools for Embedded Systems, vol*. 39, *no*. 7, ACM *Press*, pp. 57-66, Junho de 2004.

[26] KIM, J., KIM, S., KIM, D., CHOI, W., ***Implementing Real-Time Scheduling Daemon in General Purpose Operating System Unix***, *Real-Time Computing Systems and Applications, 2000, IEEE – Proceedings of Seventh International Conference on Real-Time*, pp. 177-182, Dezembro de 2000.

[27] KOLANO, P. Z., DEMMERER, R. A. ***Classification schemes to aid in the analysis of real-time systems***. *Software Engineering Notes, Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis* ISSTA '00, *vol* 25, *no*. 5. Ed. ACM *Press*, pp. 86-95, 2000.

[28] KOPETZ, H. ***Real-Time Systems: design principles for distributed embedded applications***. *Kluwer Academic Publishers*, 338 p, 1997.

[29] KRISHNA, C.; SHIN, K., ***Real-Time Systems***, ed. McGraw-Hill , New York, 448 p, 1997.

[30] LASZLO, Z. ***Memory Allocation in VxWorks 6.0 - White Paper***. *Wind River Systems, Inc.*, 7 p, 2005.

[31] LEE, Y., REDDY, K.P., KRISHNA, C.M. ***Scheduling techniques for reducing leakage power in hard real-time systems***. *EuroMicro Conference on Real Time Systems*, pp. 105-112, Julho de 2003.

[32] LIU, C. L., LAYLAND, J. W. ***Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment***. *Journal of the Association for Computer Machinery*, v. 20, n. 1, pp. 46-61, 1973.

[33] MANTEGAZZA, P., BIANCH, E., DOZIO, L., PAPACHARALAMBOUS, S., HUGHES, S., BEAL, D. ***RTAI: Real Time Application Interface.*** Artigo escrito em 6 de abril de 2000 e atualizado em setembro de 2003. Disponível em: http://www.linuxdevices.com/articles/AT6605918741.html. Acessado em 20 de Agosto de 2005.

[34] MANTEGAZZA, P., BIANCHI, E., DOZIO, L., ANGELO, M., BEAL, D. DIAPM. ***RTAI Programming Guide 1.0***, Lineo, Inc, 2000, Disponível em: http://www.aero.polimi.it/~rtai/documentation/. Acessado em 20 de Agosto de 2005.

[35] MASMANO, M.; RIPOLL, I.; CRESPO, A. ***Dynamic storage allocation for real-time embedded systems***. *Universidad Politécnica de Valencia*, Espanha, 4 p. 2001.

[36] MASTRIANI, S. J., ***Windows CE - the best choice in handheld systems for the corporate mobile***, *Thesis Presented to Department of Computer Science, Kennedy-Western University, Unionville-Connecticut*, EUA, 176 p, 2005.

[37] MICROSOFT. ***Platform Builder for Windows CE 5.0***, Disponível em: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcecoreos5/html/wce50conkernel.asp. Acesso em: 12 de outubro de 2005.

[38] MICROSOFT. ***Real-Time Systems with MS Windows CE***. White Paper, Acessado em 21 de Julho de 2006, Disponível em: http://www.microsoft.com/technet/prodtechnol/wce/plan/realtime.asp.

[39] MOHAMMADI, A., AKL, S. G., ***Scheduling Algorithms for Real-Time Systems***, *Technical Report no. 2005-499, School of Computing – Queen's University, Kingston, Ontario, Canada*, 49 p, Julho de 2005.

[40] O'BROIN, J. ***Real-Time Processes (RTPs) for VxWorks 6.0 - White Paper***. *Wind River Systems, Inc.*, 9 p, 2005.

[41] PAPADIMITRIOU, C; LEWIS, H. **Elementos de Teoria da Computação**, Ed. *Bookman*, 336 p, 2000.

[42] PEDRO, A., SANTOS, M., RENÓ, D. **Análise de escalonabilidade de tarefas no kernel de tempo real S.Ha.R.K.**, Anais do III Congresso Brasileiro de Computação (CBCOMP), Universidade do Vale do Itajaí, Itajaí, SC, 13 p, Agosto de 2003.

[43] PILLAI P.; SHIN K., ***Real-Time Dynamic voltage scaling for Low-Power Embedded Operating Systems***, *ACM SIGOPS Operating Systems Review, Proceedings of the eighteenth ACM symposium on Operating Systems principles SOS´01*, vol. 35, no. 5, ed. ACM Press, pp. 89-201, 2001.

[44] POSIX 2003.1b 2000. ***IEEE Standard for Information Technology - Test Methods Specifications for Measuring Conformance to POSIX - Part 1: System Application Program Interface (API) - Amendment 1: Real-time Extension [C Language]***, Nova York, NY, EUA, 384 p, 2000.

[45] POSIX ***Real-Time Application Support (AEP)***, IEEE std 1003.13-1998, IEEE, Piscataway, NJ, 138 p, Março de 1998.

[46] POSIX. 2001, ***The Open Group Base Specifications*** Issue 6. IEEE Std 1003.1-2001-2004. Disponível em: http://www.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap01.html, Acessado em 12 de julho de 2005.

[47] ***QNX Neutrino RTOS – Features and Benefits.*** Acessado em: 14 de agosto de 2006. Disponível em: http://www.qnx.com/products/rtos/

[48] ***Real-time operating system From Wikipedia, the free encyclopedia.*** Acessado em 28 de novembro de 2005. Disponível em: http://en.wikipedia.org/wiki/RTOS

[49] RIPOLL, I., PISA, P., ABENI, L., GAI, P., LANUSSE, A., SAEZ, S. e PRIVAT, B. ***WP1 - RTOS State of the Art Analysis: Deliverable D1.1*** - RTOS Analysis, 2002, Disponível em http://www.ocera.org, Acessado em 10 de junho de 2005.

[50] RTAI, ***RTAI – Official Website***, Disponível em https://www.rtai.org/. Acessado em: 23 de agosto de 2005.

[51] RUSSINOVICH, M; SOLOMON, D. ***Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000***. Ed.: *Microsoft Press*, 976 p, Dezembro de 2004.

[52] **S.Ha.R.K.** - ***Soft Hard Real-time Kernel.*** Disponível em: http://shark.sssup.it/kernel.shtml. Acessado em 10 de outubro de 2005.

[53] SADJADI, S.; MCKINLEY, P.; CHENG, B. ***Transparent shaping of existing software to support pervasive and autonomic computing***, *ACM SIGSOFT Software Engineering Notes, Proceedings of the 2005 Workshop on Design and Evolution of Autonomic Application Software* DEAS '05, vol. 30, no. 4, pp. 1-7, Maio de 2005.

[54] SCALLON, G.; NAST, D., ***A sample problem in real-time control systems***, *Real-Time Systems Newsletter*, vol. 3, no. 3, 1998, pp. 6-12.

[55] SHA, L.; RAJKUMAR, R.; LEHOCZKY, J. ***Priority Inheritance Protocols: Na approach to real-time syncronization***. IEEE *Transactions on Computers*, pp. 1175-1185, Setembro de 1991.

[56] SHAW, A. C. **Sistemas e Software de Tempo Real**. São Paulo. Ed.: Bookman, 240 p, 2003.

[57] SHIN, D.; KIM, J., ***Intra-task Voltage Scheduling on DVS-Enabled Hard Real-Time Systems***, *IEEE Design and Test of Computers*, Março de 2001.

[58] SPUN, M., BUTTAZZO, G.C., ANNA, S.S.S. ***Efficient Aperiodic Service under Earliest Deadline Scheduling***. *IEEE Real-Time Systems Symposium, December*, pp. 2-11, 1994.

[59] STANKOVIC, J. ***Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems***. IEEE *Computer Society Press*, vol. 21, no. 10, pp. 10-20, Outubro de 1988.

[60] STANKOVIC, J.; RAJKUMAR, R., ***Real-time Operating Systems***, *Kluwer Real-Time Journal*, vol. 28, no. 2-3, ed. *Springer Netherlands*, pp. 237-253, Novembro de 2004.

[61] STANKOVIC, J; SPURI, M.; RAMAMRITHAM, K; BUTTAZZO, G. ***Deadline scheduling for real-time systems: EDF and related algorithms***. Kluwer Academic Publishers, 272 p, 1998.

[62] ***The Concise Handbook Of Real-Time Systems.*** *TimeSys Corporation*, Versão 1.3, 2002. Acessado em 20 de novembro de 2005. Disponível em: http://www.timesys.com

[63] ***The Embedded and Microprocessor Benchmark Consortium*** – Acessado em 18 de fevereiro de 2006, Disponível em: http://www.eembc.org/about/

[64] ***The OCERA Project, What is OCERA: Objectives and Description of the Wor***k. Acessado em 14 de outubro de 2005. Disponível em: http://www.ocera.org/info/whatis.html

[65] WEIDERMAN, N.; KAMENOFF, N., ***Hartstone Uniprocessor Benchmark: Definitions and Experiments for Real-Time Systems***, *The Journal of Real-Time Systems*, pp. 353-382, 1992

[66] WEINBERG, B., CESATI, M. ***Moving from a Proprietary RTOS To Embedded Linux.*** *MontaVista Software, Inc*., pp. 55-58, 2001.

[67] ***Wikipedia, the free encyclopedia*** - Acessado em 19 de fevereiro de 2006, Disponível em: http://en.wikipedia.org/wiki/Benchmark_computing.

[68] YAO, P. ***Windows CE: Enhanced Real-Time Feature Provide Sophisticated Thread Handling***. Acessado em 11 de Agosto de 2005, Disponível em: http://msdn.microsoft.com/msdnmag/issues/1100/RealCE/default.asp.

[69] ZHANG, F., CHANSON, S.T., BAY, C.W., KOWLOON, H.K. ***Blocking-Aware Processor Voltage Scheduling for Real-Time Tasks***. *ACM Transactions on Embedded Computing Systems*, ACM *Press*, *vol.* 3, *no.* 2, pp. 307-335, Maio de 2004.

[70] ZHAO, W., RAMAMRITHAM, K., STANKOVIC, J., ***Scheduling Tasks with Resources Requirements in Hard Real-Time Systems***, *IEEE Trans. Software Eng.*, vol SE-13, no. 5, 10 p, Maio de 1987.

[71] ZHU, D.; MELHEM, R.; CHILDERS, B., ***Scheduling with Dynamic Voltage/Speed Adjustment Using Slack Reclamation in Multi-Processor Real-Time Systems***, *IEEE Trans. on Parallel & Distributed Systems*, vol. 14, no. 7, pp. 686 - 700, 2003.

Apêndice A

Efetuar Cálculos Matemáticos

Descrição do *benchmark*

O *benchmark* efetua um cálculo polinomial cúbico em ponto flutuante, calcula a raiz quadrada de valores inteiros e de ponto flutuante e calcula a transformação de um valor em graus para radianos e vice-versa. Usa-se as seguintes equações:

Cálculo polinomial:

```
a1 = b/a, a2 = c/a, a3 = d/a
Q = (a1*a1 - 3.0*a2)/9.0
R = (2.0*a1*a1*a1 - 9.0*a1*a2 + 27.0*a3)/54.0
R2_Q3 = R*R - Q*Q*Q
theta = acos(R/sqrt(Q*Q*Q))
x[0] = -2.0*sqrt(Q)*cos(theta/3.0) - a1/3.0
x[1] = -2.0*sqrt(Q)*cos((theta+2.0*PI)/3.0) - a1/3.0
x[2] = -2.0*sqrt(Q)*cos((theta+4.0*PI)/3.0) - a1/3.0
```

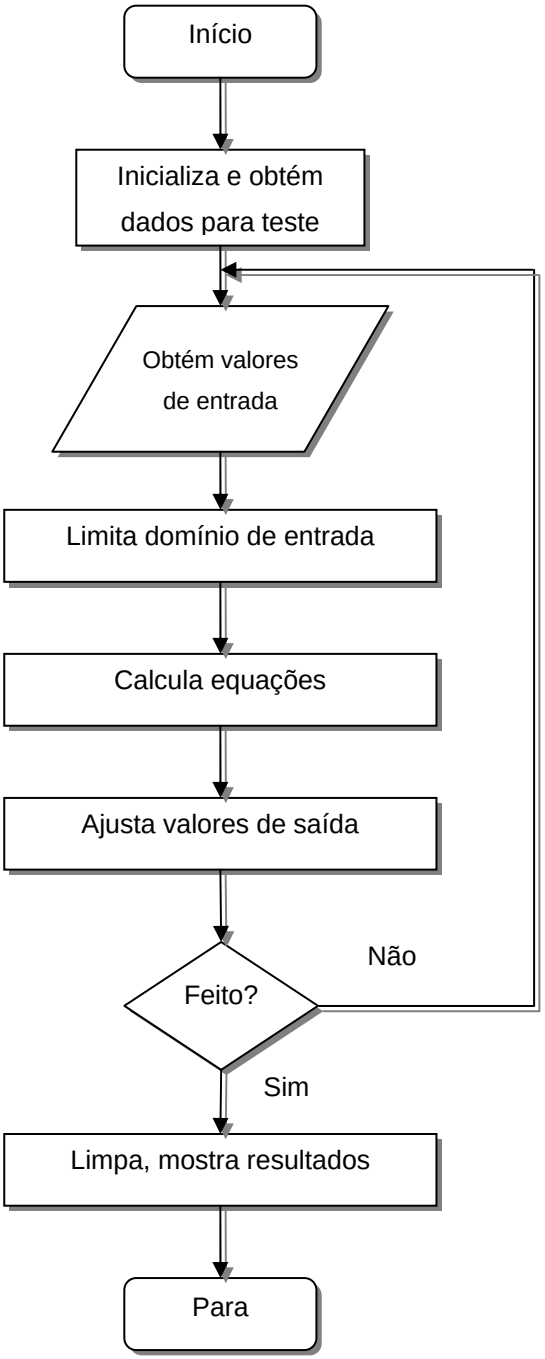
Cálculo da raiz quadrada (Usando série de Taylor):

```
x = val/10
dx = (val - (x*x)) / (2.0 * x)
x = x + dx
diff = val - (x*x)
```

Transformação de Graus para Radianos e vice-versa:

```
x1 = (180.0 * rad / (PI));
x2 = (PI * deg / 180.0);
```

Fluxograma do algoritmo



Ordenação de Inteiros Usando *Quicksort*

Descrição do *benchmark*

O *benchmark* efetua ordenação de 20 elementos inteiros usando o algoritmo *quicksort*, não recursivo, que em média possui um desempenho muito bom para seqüências grandes, porém não é muito eficiente para seqüências pequenas. Isso ocorre porque para seqüências pequenas o tempo do *quicksort* será quadrático e ocupará mais espaço em memória devido aos particionamentos.

Descrição do algoritmo

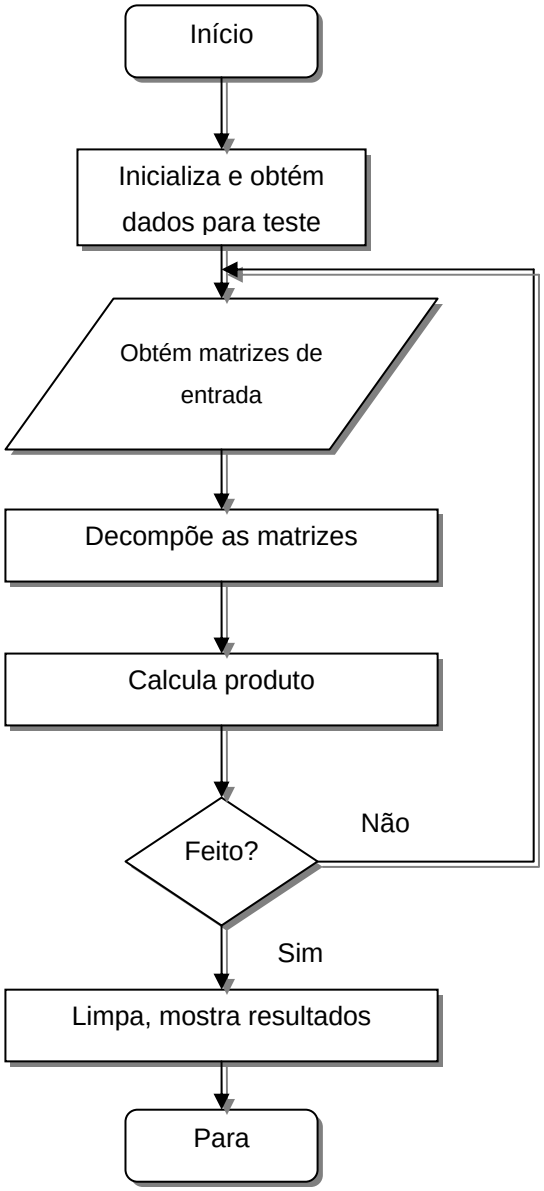
```
empilha(1, n)

repita até que a pilha esteja vazia
 desempilha(esq, dir)

 enquanto esq < dir faça
      partição(esq, dir, i, j)

      se j - esq > dir - i então
           empilha(esq, j)
           esq = i
      fim se
      senão
           empilha(i, dir)
           dir = j
      fim senão
 fim enquanto
fim repita
```

Multiplicação de Matriz

Descrição do *benchmark*

Este *benchmark* simula uma aplicação automotiva/industrial em que efetua um cálculo aritmético com matriz. A aplicação executa a decomposição de duas matrizes 5x5 usadas como entrada e efetua o cálculo de sua multiplicação obtendo a saída.

Fluxograma do algoritmo

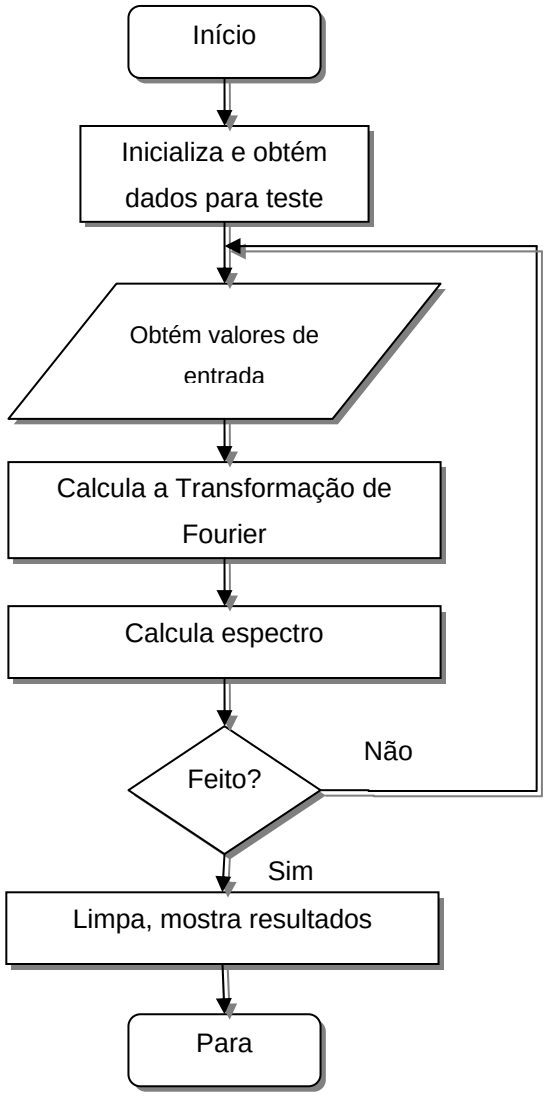


Transformada Rápida de Fourier

Descrição do *benchmark*

Este *benchmark*, de acordo com as especificações do EEMBC, simula uma aplicação automotiva/industrial executando uma análise de tempo variando seus dados de entrada. Existem duas entradas, INPUT_1 e INPUT_2, sendo que esta segunda entrada é a parte do número real do *array* de entrada e a primeira entrada é a parte do número imaginário. Após os valores do domínio de tempo serem convertidos para a freqüência equivalente, os valores do espectro é calculado.

Fluxograma do algoritmo



Manipulação de *Bits*

Descrição do *benchmark*

O *benchmark*, de acordo com as especificações do EEMBC, simula uma aplicação automotiva/industrial onde grandes números têm de ser manipulados, muitas decisões tenham de ser tomadas fundamentadas em valores e aritmética de *bits*. A idéia do *benchmark* é utilizar esta aplicação para simular parte de um sistema de *display* onde caracteres tenham de se mover em uma determinada ordem.

Fluxograma do algoritmo