Suporte a Aplicações de Tempo Real em um Ambiente baseado em Multicomputador

Edgard de Faria Corrêa
Núcleo de Computação Científica
Universidade Federal do Rio Grande do Norte
CP 1530 Natal RN Brasil - CEP 59078-970
efc@ncc.ufrn.br

Luis Fernando Friedrich
Departamento de Informática e de Estatística
Universidade Federal de Santa Catarina
CP 476 Florianópolis SC Brasil - CEP 88040-900
lff@inf.ufsc.br

Resumo

Sistemas de Tempo Real são aqueles sistemas onde a execução correta não depende apenas dos resultados lógicos da computação, mas também do tempo no qual os resultados são produzidos, ou seja, se ocorre dentro do tempo previsto. Aplicações de tempo real estão cada vez mais presentes no dia-a-dia e em áreas das mais variadas, tais como: multimídia, robótica, sistemas médico-hospitalares, telecomunicações, controle de manufatura e de processos, controladores de vôo, dentre outros.

Este artigo apresenta um conjunto básico de primitivas de suporte para aplicações de tempo real no ambiente do multicomputador CRUX baseados no padrão POSIX da IEEE. O Multicomputador CRUX servirá como base para o projeto e implementação de um ambiente completo para programação paralela em desenvolvimento no Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina. Com o fornecimento desse conjunto mínimo de primitivas objetiva-se adequar o ambiente de processamento paralelo para a execução de aplicações de tempo real. Além da definição do conjunto básico de primitivas, foi realizada a avaliação de desempenho, através de simulação, para analisar o desempenho e a previsibilidade dos mecanismos de comunicação.

Abstract

Real Time Systems are those systems where the correct execution doesn't just depend on the logical results of the computation, but also on the time in which the results are produced, meaning that, the results happened on foreseen time. Nowadays, applications of real time are more and more present in various areas, such as: multimedia, robotics, support-life systems, telecommunications, manufacture and processes control, flight controllers, among others.

This article presents a basic group of primitives of support for real time applications on the environment of the multicomputador CRUX based on the standard POSIX of IEEE. The multicomputador CRUX will be the base for the project and implementation of a complete environment for parallel programming on development in the Computer Science Department of the Federal University of Santa Catarina. The main goal of this group of primitives is to be able to support real time application on CRUX environment. In addition, a performance evaluation, by simulation, in order to analyse the performance and the predictability of the communication mechanism.

Palavras Chaves

Sistemas Operacionais de Tempo Real, Sistemas Distribuídos, Multicomputador, POSIX, Multithreads, Avaliação de Desempenho, Simulação.

1. Introdução

Sistemas de Tempo Real (STR) podem ser definidos como a classe de sistemas de computação em que a execução correta destes sistemas não depende apenas dos resultados lógicos da computação, mas também do tempo no qual os resultados são produzidos. Tais sistemas interagem com o ambiente externo em intervalos de tempo definidos por este ambiente. Esta definição abrange uma grande variedade de sistemas, tais como: sistemas multimídia, filtros digitais, sistemas de telecomunicações, sistemas de controle de manufatura e sistemas de controle de processos.

O tempo é o recurso principal a ser gerenciado em STR. Outros dois componentes que caracterizam esses sistemas são [1]: a confiabilidade no atendimento às restrições temporais, que é crucial, e o ambiente com o qual o computador interage, que é um componente ativo nas aplicações de tempo real. A primeira característica poderá ser relaxada ou não, dependendo do tipo de STR em questão: *soft* ou *hard*. As possíveis conseqüências de uma falha determinam o nível de confiabilidade necessária. Nos sistemas de tempo real *soft*, uma falha temporal é da mesma ordem de grandeza que os benefícios do sistema em operação normal, enquanto que nos sistemas de tempo real *hard* as conseqüências de uma falha temporal excedem em muito os benefícios normais, sendo, em geral, fatal para o sistema.

Dentre os diversos conceitos errôneos sobre STR citados em [2], o mais freqüente é a concepção de que o principal requisito é a velocidade (da computação, da comunicação, de acesso ao sistema de arquivos, dentre outros). O desempenho é um requisito importante em muitos sistemas de tempo real, mas deve-se levar em consideração apenas o desempenho no pior caso, ao contrário de sistemas de propósito geral que costumam considerar o desempenho no caso médio.

As pesquisas em STR abordam vários aspectos: linguagens de programação, sistemas operacionais, sistemas distribuídos, teoria de escalonamento, dentre outros. Sistemas Operacionais de Tempo Real (SOTR) são parte fundamental de sistemas de tempo real, devendo ser capazes de realizar de forma integrada o escalonamento da CPU e a alocação de recursos, de maneira que o conjunto de tarefas cooperantes possam obter os recursos que elas necessitam dentro das suas restrições temporais. Esta previsibilidade requer, entre outras coisas, primitivas de sistemas operacionais que tenham seus tempos máximos de execução definidos. Além disso, a utilização dos paradigmas de sistemas operacionais convencionais, que permitem esperas arbitrárias por recursos ou eventos, ou tratem uma tarefa como um processo randômico, não pode ser considerada em se tratando de sistemas críticos, por exemplo, em sistemas de tempo real *hard*. Da mesma forma que em sistemas operacionais convencionais, nos SOTR as quatro áreas funcionais mais importantes são: gerenciamento de processos, sincronização e comunicação entre processos, gerenciamento de memória e mecanismos de entrada/saída. Entretanto, a forma como estas áreas são suportadas difere dos sistemas convencionais.

É possível programar e executar algumas aplicações de tempo real *soft* (multimídia, por exemplo) sobre sistemas operacionais convencionais (como UNIX). Entretanto, essa execução ocorre de forma pouco satisfatória devido a falta de suporte à previsibilidade necessária em tais aplicações. Recentemente, a importância de SOTR veio a tona com a experiência da sonda *Pathfinder* enviada pela NASA ao planeta Marte. A utilização de primitivas existentes no SOTR (no caso *VxWorks*) do robô *Sojouner*, enviado com a sonda, foi essencial para resolver seguidos *resets* que ocorreram durante a exploração [3].

Este artigo apresenta, na seção 2, uma descrição do ambiente de aplicação CRUX. Na seção 3 é abordado o suporte a tempo real, fazendo uma breve descrição do padrão POSIX da IEEE e apresentando o conjunto básico de primitivas para o ambiente CRUX. A seção 4 traz uma análise de desempenho dos mecanismos de comunicação utilizados, através da simulação de seus parâmetros e de sua previsibilidade. As considerações finais deste trabalho são apresentadas na seção 5.

2. Ambiente de Aplicação: Multicomputador CRUX

O ambiente CRUX representa um completo ambiente para programação paralela, sendo composto pelo Multicomputador CRUX e pelo Sistema Operacional CRUX.

2.1 Arquitetura CRUX

O multicomputador CRUX (Figura 1) é composto de um conjunto de nós de trabalho (NT) ligados por meio de um comutador de conexões e um barramento compartilhado. O comutador de conexões (*crossbar*) é manipulado durante o funcionamento normal da máquina pelo nó de controle (NC), que por sua vez, utiliza um barramento de serviço (BS) e um *link* de configuração para determinar as necessidades do sistema e definir dinamicamente a estrutura da rede comunicação.

O nó de controle utiliza o barramento de serviço para realizar uma pesquisa seqüencial de modo a determinar os pedidos dos nós de trabalho. Esta pesquisa é feita através do questionamento, pelo envio de um comando ao nó inquirido. Caso este nó não deseje nenhum serviço, o nó de controle questiona o próximo nó de trabalho. No entanto, se o nó questionado desejar algum serviço, ele então envia uma requisição ao nó de controle. Após atender cada pedido, o nó de controle volta ao questionamento seqüencial dos nós de trabalho. No caso do pedido de serviço, retornado por um nó de trabalho, ser um pedido de conexão com outro nó, o nó de controle envia ao comutador de conexões a configuração para a conexão pedida. Contudo, se o nó solicitado já estiver conectado, o pedido é colocado em uma fila de espera e o nó de controle prossegue com o *polling*.

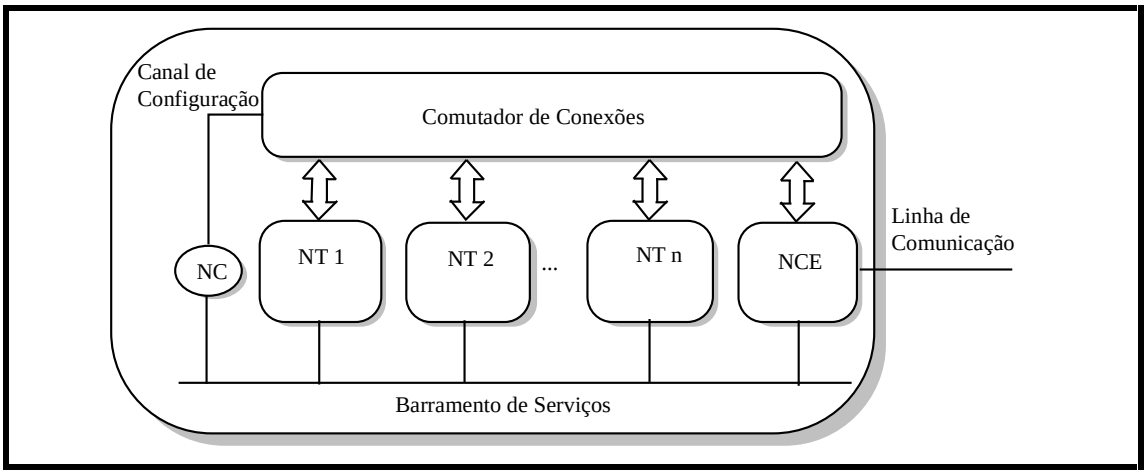

Figura 1 - A arquitetura do multicomputador CRUX

2.2 Sistema Operacional CRUX

O sistema operacional CRUX. combina o modelo de processos que se comunicam exclusivamente por trocas de mensagens com a arquitetura dinâmica do multicomputador, sendo compatível com o sistema UNIX. Ele é constituído de um micronúcleo distribuído, de biblioteca de funções de acesso às primitivas do sistema e de um servidor [4] [5].

Técnicas de gerenciamento de memória, como paginação ou segmentação, não são aplicadas, pois considera-se a memória como um recurso abundante. Na gerência de processos não há o escalonamento, pois existe apenas um processo por nó e este cabe em sua memória privativa. O sistema operacional CRUX foi implementado para executar na arquitetura baseada em barramento comum e é formado por duas peças principais: um micronúcleo e um conjunto de processos servidores (Figura 2).

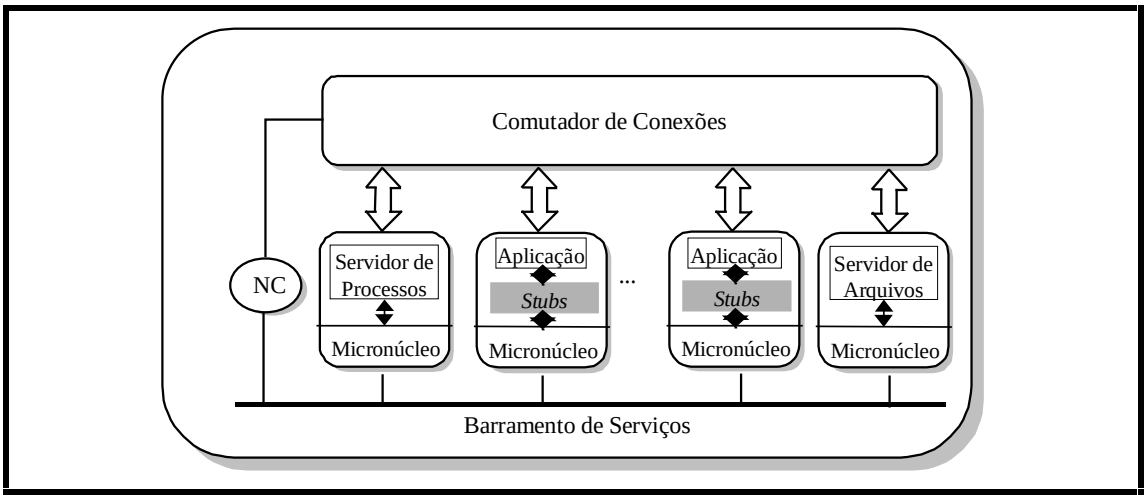

Figura 2 - A arquitetura do sistema CRUX

A Comunicação entre os processos no CRUX é efetuada através de canais bipontuais, com endereçamento direto, de maneira síncrona. Os serviços de sistema no CRUX são implementados através de dois processos servidores: o servidor de processos e o servidor de arquivos. Os processos de aplicação acessam estes serviços através de uma camada de *stubs*, que provê uma interface de programação compatível com a do sistema UNIX.

3. Suporte a Tempo Real

De uma forma geral, aplicações de tempo real apresentam requisitos diferentes das aplicações convencionais "não tempo real". Dentro do ambiente de aplicação da máquina CRUX propõe-se um conjunto mínimo de primitivas de tempo real, baseadas no padrão POSIX da IEEE.

3.1 Padronização

O objetivo de uma padronização é, fundamentalmente, permitir a portabilidade do código-fonte de aplicações, de modo que na mudança, de uma aplicação de um sistema operacional para outro, seja necessário apenas a recompilação dessa. O padrão POSIX é o conjunto de documentos produzido pela IEEE, e adotado pela ANSI e ISO, composto basicamente dos seguintes documentos:

- POSIX.1: núcleo básico de interface entre sistemas operacionais. É necessário em qualquer sistema de tempo real, mas não é suficiente. Ele trata das operações comuns de funções de sistema que tornam portáveis as aplicações na mudança de um sistema operacional para o outro, necessitando apenas a recompilação das mesmas. Algumas dessas operações são: mecanismos para criação/término de processos (*fork*, *exec*, *wait*, *exit*), sinais, entrada/saída básica e manipulação de terminal.
- POSIX.1b [6]: acrescenta extensões para tempo real, tratando assuntos como: entrada/saída síncrona e assíncrona, bloqueio de memória, memória compartilhada, passagem de mensagem, semáforos, escalonamento (prioritário e round-robin) e relógios de tempo real (com granularidade da ordem de nanosegundos).
- POSIX.1c [7]: acrescenta extensões adicionais ao POSIX.1 e POSIX.1b, provendo a habilidade de executar múltiplos *threads* concorrentes de um único processo. Ele trata de gerenciamento e escalonamento de *threads*, variáveis condicionais, sinais e escalonamento de processos.
- POSIX.1d [8]: acrescenta outras extensões adicionais de tempo real: criação de processos, funções de bloqueio e *timeout*, monitoração de tempo de execução, escalonamento de servidores esporádicos e controle de dispositivos e de interrupções.

- POSIX.13: proposta para prover quatro perfis de sistemas correspondentes aos vários níveis de funcionalidade, desde de sistemas embutidos até sistemas com todas as funcionalidades de tempo real.

Essa portabilidade, entretanto, envolve generalização e flexibilidade, que custam tempo e espaço. Um programa não-portável é pequeno e rápido. Na construção da aplicação é necessário balancear portabilidade com uma eficiência razoável, o que geralmente implica em componentes não-portáveis. Assim o POSIX não resolve totalmente o problema de portabilidade, mas procura facilitar.

Outro fator importante, em se tratando de sistemas tempo real, é a necessidade de garantia temporal. A padronização POSIX permite uma portabilidade de plataformas, mas não garante uma "portabilidade temporal" [9].

O conjunto de primitivas de suporte a tempo real para o ambiente CRUX foi definido dentro das áreas funcionais de gerência/escalonamento de processos e *threads*, gerência de memória e comunicação entre processos/*threads*.

3.2 Criação, Gerenciamento e Escalonamento de Threads

Tendo em vista que na máquina CRUX tem apenas um processo sendo executado por processador, não se faz necessário um controle no escalonamento dos processos, pois estes são alocados em um dado processador no momento de sua criação. No entanto, cada processo pode ser *multithread*, ou seja, pode ter mais de uma *thread* de execução. Isto permite uma maior flexibilidade por parte das aplicações de tempo real. Entretanto, é necessário realizar o gerenciamento dessas *threads*. A opção foi pela utilização de um número mínimo de primitivas de suporte, de modo a facilitar a sua utilização a nível do núcleo ou a nível de usuário.

O modelo de *multithreads* divide o processo em duas partes. Uma parte refere-se ao processo e, contém recursos usados ao longo de todo o programa, tais como: instruções de programa, dados globais, entre outros. A outra parte refere-se as *threads* e, contém informações relacionadas ao estado de execução, tais como: contador de programa e pilha (Figura 3). Assim, na mudança de contexto entre as *threads*, não é necessário salvar as informações das pilhas, mas apenas o conteúdo dos registradores, simplificando as mudanças de contexto e reduzindo o tempo de comutação.

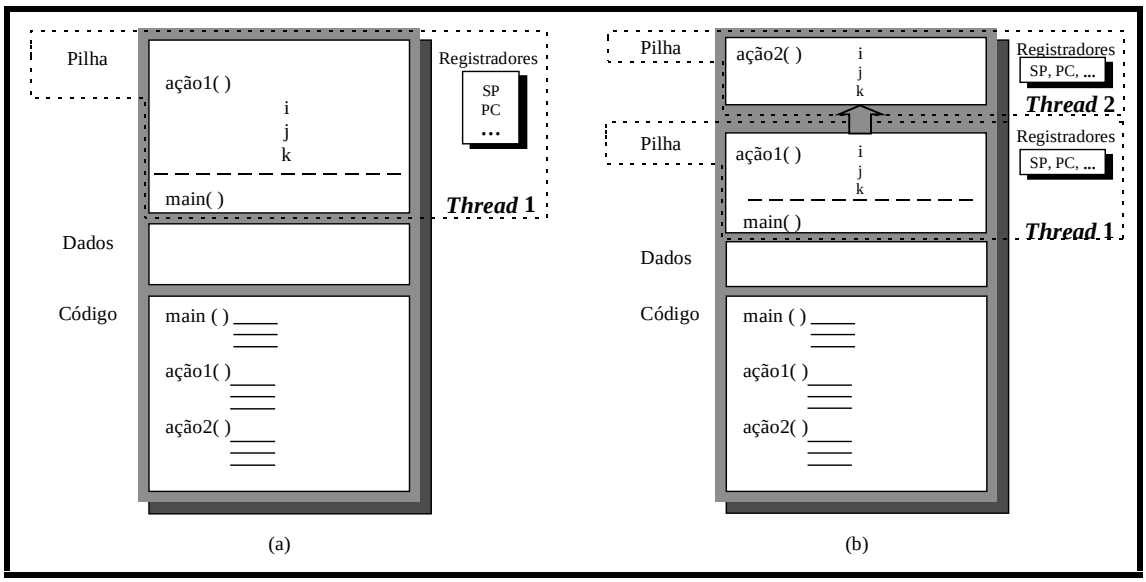

Figura 3 - Processo com uma *thread* (a) e com *multithreads* (b)

No gerenciamento das *threads*, principalmente quando se trata de mudança de contexto, é necessário que o sistema mantenha as informações necessárias para uma posterior retomada na execução da *thread* que está sendo suspensa. Essas informações são armazenadas em uma estrutura de dados, a tabela de *threads* (Tabela 1), onde, para cada *thread*, existe uma entrada na

tabela. Além disso, é acrescentado à tabela de processos, a identificação das *threads* pertencentes a cada processo.

Cada entrada da tabela de *threads* é referenciada por um item de identificação da *thread* (*thread_id*). Esse identificador é incluído na tabela de processos, no item que possui o vetor com as identificações das *threads* do processo.

Tabela 1 - Tabela de *threads*

ITEM	**DESCRIÇÃO**
thread_status	estado da *thread* (executando, pronta, suspensa, livre, *sleep*, mens, *mutex*).
thread_reg	registradores.
thread_stack_base	base da pilha da *thread*.
thread_stack_tam	tamanho da pilha da *thread*.
thread_prio	prioridade da *thread*.
thread_deadline	prazo máximo para execução (*deadline*) da *thread*.
thread_start_time	tempo estabelecido para o início da *thread*.
thread_arrival_time	tempo de chegada da *thread*.
thread_politica	política de escalonamento.

O estado da *thread* é identificado no item *thread_status*, dentre um dos valores do vetor de estados da *thread*: executando, quando a *thread* está ativa; pronta, quando a *thread* se encontra na fila de prontas, aguardando para executar; suspensa, quando está no estado suspenso; livre, quando não há nenhuma *thread* executando; *sleep*, quando a *thread* está "dormindo"; mens, quando a *thread* está aguardando uma mensagem e *mutex*, quando está aguardando um *mutex*.

O item *thread_reg* é o vetor que guarda os valores dos registradores da *thread*. Os itens *thread_stack_base* e *thread_stack_tam* armazenam, respectivamente, o endereço de base da pilha da *thread* e o seu tamanho. O limite de tamanho da área total de pilha é definido na tabela de processos.

Os demais itens (*thread_prio*, *thread_deadline*, *thread_start_time*, *thread_arrival_time*) indicam, respectivamente, os seguintes atributos da *thread*: sua prioridade, o prazo máximo para ser executada (*deadline*), o tempo estabelecido para o seu início e o seu tempo de chegada. Os três primeiros atributos podem ser estabelecidos no momento da criação da *thread*, ou alterados, posteriormente, através da primitiva de escalonamento.

Threads não têm um relacionamento hierárquico ("pai/filho") semelhante aos processos. Assim, uma *thread* pode cancelar qualquer outra *thread*, mesmo que não tenha sido responsável pela sua criação. Além de primitivas para criação e cancelamento de *threads*, outras primitivas foram previstas para permitir a associação entre *threads*, ou seja, que uma *thread* possa aguardar ou deixar de aguardar o término de uma outra, e também que a *thread* que está sendo aguardada possa indicar o seu término.

Muitos sistemas operacionais utilizam mecanismos de escalonamento baseados em prioridade fixas, onde a aplicação pode alterar a prioridade de uma determinada *thread*, através de chamadas de sistema. Isso funciona bem quando as prioridades das tarefas são fixas. Mas, na maioria dos casos, essa alternativa é ineficiente, pois o sistema irá requerer prioridades dinâmicas e o mapeamento de um esquema dinâmico sobre prioridades fixas pode ser bastante ineficiente. O escalonamento com prioridades fixas é também incompleto, pois trata apenas de recursos da CPU, sem considerar outros recursos envolvidos com as tarefas. Além disso, um simples número, associado pela aplicação, como parâmetro de prioridade, não é suficiente para representar todo o conjunto de características, que devem ser levados em conta, em um escalonamento de tempo real.

Com o objetivo de manter a flexibilidade, a utilização de um escalonador mínimo é uma alternativa atraente, sobre o qual podem ser desenvolvidos mecanismos de escalonamento mais complexos, do tipo: Taxa Monotônica (*Rate Monotonic*), Próximo *Deadline* Primeiro (*Earliest Deadline First*) [10], *Deadline* Monotônico (*Deadline Monotonic*) [11], dentre outros. Essa flexibilidade, permite, entre outras coisas, uma maior portabilidade e adaptabilidade.

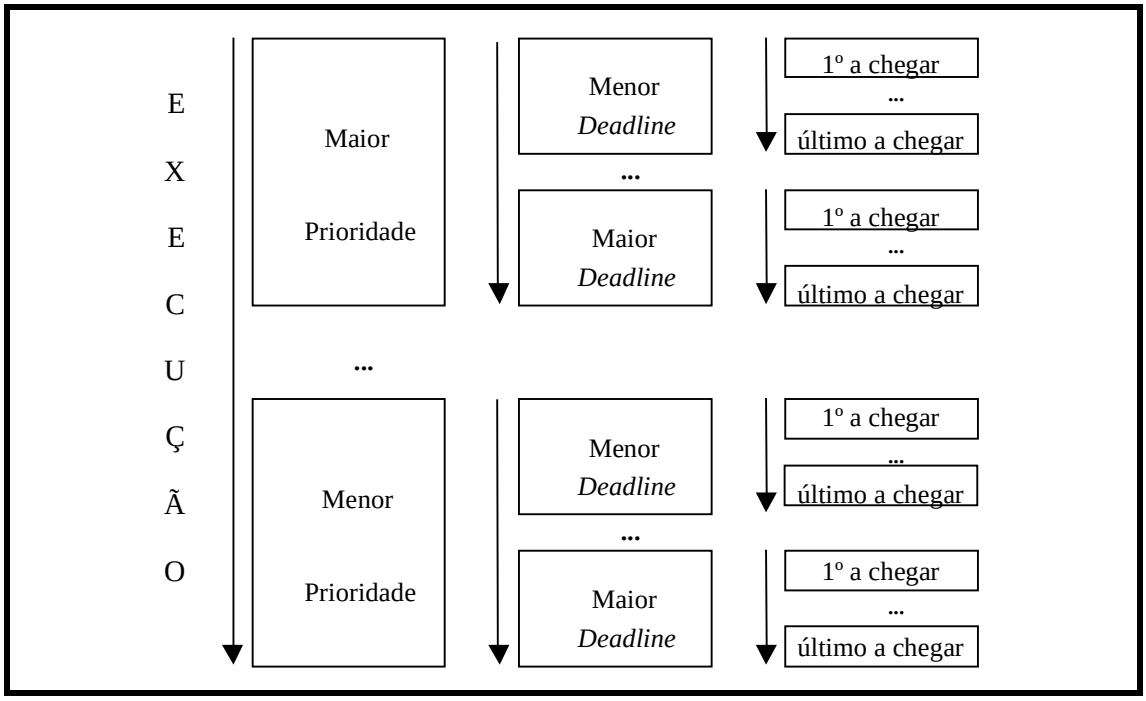

Figura 4- Esquema de escalonamento hierárquico

Os atributos da *thread* utilizados para o escalonamento (prioridade, *deadline*, tempo estabelecido para o início e o tempo de chegada) com exceção deste último, podem ser estabelecidos no momento da criação da *thread*, ou em um momento posterior com a primitiva de escalonamento. Se o parâmetro de tempo mínimo para inicialização (*thread_start_time*) é especificado a *thread* é suspensa até o valor de tempo indicado no parâmetro. As threads são escalonadas de acordo com a prioridade. *Threads* são preemptadas por outras de maior prioridades e aquelas com mesma prioridade serão executadas de acordo com seus *deadlines*. Quando não for indicado nenhum *deadline* no atributo *thread_deadline*, executa primeiro a que possuir menor tempo de chegada (*thread_arrival_time*). Esse esquema de escalonamento hierárquico (Figura 4) segue modelo semelhante ao apresentado por [12], que provê modularidade e flexibilidade, além de permitir o tratamento dos diversos parâmetros envolvidos no escalonamento, em lugar de transformá-los em apenas um número.

3.3 Sincronização de Threads

Threads, além de cooperar, podem também concorrer por recursos comuns. Assim, é preciso existir garantias de exclusão mútua quando essas forem acessar regiões críticas do código que trata desses recursos compartilhados. Para tanto, é necessário estabelecer uma sincronização entre as diversas *threads* que podem estar concorrendo por determinado recurso, ou dependendo de resultados de outras *threads* (cooperação). Essa sincronização é feita utilizando *mutex* ou variáveis condicionais, onde as primitivas criam, deletam, bloqueiam ou desbloqueiam os *mutexes* ou variáveis condicionais.

A variável condicional difere do *mutex* na maneira de tratar a liberação da variável. Quando um *mutex* é liberado, seu efeito continua, ou seja, mesmo que não haja nenhuma *thread* bloqueada aguardando a liberação daquele *mutex*, quando, mais tarde, uma *thread* tentar

bloquear o *mutex*, obterá êxito. No caso das variáveis condicionais, nada ocorrerá se nenhuma *thread* estiver esperando uma determinada variável condicional no momento de sua liberação.

No entanto, um cuidado que se deve ter na questão de sincronização e escalonamento de *threads* é o problema da inversão de prioridades. Este problema ocorre quando temos uma *thread* com maior prioridade que está aguardando um recurso alocado para uma *thread* de menor prioridade, e nesse ínterim uma *thread* com prioridade intermediária faz a preempção da *thread* com menor prioridade. Devido a isso, a *thread* de maior prioridade também fica suspensa, ocasionando uma quebra na hierarquia de prioridades. Uma maneira de tratar isto é utilizar o mecanismo de herança de prioridade. Nesse mecanismo, quando uma *thread* de maior prioridade ficar bloqueada devido a uma outra *thread* de menor prioridade, que está na região crítica, a prioridade desta última é "elevada", temporariamente, ao mesmo valor da primeira. Esse artifício evita que uma *thread* com prioridade intermediária, que não necessita daquele recurso, consiga assumir o processador. É importante ressaltar que o mecanismo de herança de prioridade não evita totalmente a inversão de prioridade, apenas permite uma delimitação do pior caso em que uma tarefa pode ficar bloqueada por uma de menor prioridade, garantindo a existência de previsibilidade.

3.4 Gerência de Memória

Uma vez que o CRUX associa a cada processador um único processo, torna-se desnecessário uma gerência de memória virtual. Isso é muito desejável, já que o gerenciamento de memória virtual, como paginação ou *swapping*, traz consigo uma imprevisibilidade associada, pois o benefício de tornar menor o tempo de acesso não pode ser considerado uma vez que em tempo real temos que considerar o pior caso (que, no caso, seria a informação não estar em *cache*).

Outro fator que também contribui para a não preocupação com o gerenciamento de memória é a inexistência de memória compartilhada, uma vez que em um mesmo processador apenas um processo pode existir. Desta forma, não existe problemas de compartilhamento de memória por vários processos e a tarefa de alocação é trivial. A comunicação entre esses processos é feita através de troca de mensagens, sem haver utilização de áreas de memória comum.

Deve ser considerado, entretanto, a necessidade de gerenciamento de memória para as diversas *threads* que compõem um processo. Uma vez que as *threads* compartilham da área do processo, é necessário a garantia de exclusividade no acesso a memória. Isso pode ser realizado através dos *mutexes* ou das variáveis condicionais.

3.5 Comunicação entre Processos/Threads

A comunicação entre processos e *threads*, no ambiente CRUX, é realizada, exclusivamente, por passagem de mensagens. Essa passagem de mensagens pode ser feita utilizando canais de comunicação.

A comunicação em um sistema de tempo real necessita ser, antes de mais nada, previsível, ou seja, é necessário a garantia de que uma determinada mensagem seja recebida/atendida em um tempo máximo (*deadline*), ou ainda, seja enviada em um determinado instante. As primitivas utilizadas permitem que seja estabelecida a comunicação por passagem de mensagens, implementando filas de mensagens. Desta forma, é possível a utilização, tanto de mecanismos de canais quanto de mecanismos de caixas postais(se desejável), associando esses mecanismos a uma determinada fila de mensagem.

Através das primitivas de comunicação pode-se criar ou acessar uma determinada fila de mensagens. Para tanto, é necessário indicar, como parâmetros, o identificador da fila e o valor dos parâmetros necessários, como o modo de acesso (somente leitura, somente escrita ou escrita/leitura), se será não-bloqueante (*nonblock*), bem como se aquela fila está sendo criada ou acessada.

Quando se deseja fechar uma fila, passa-se o identificador da fila. Filas de mensagens são persistentes, isto é, mesmo quando fechadas, as mensagens enviadas permanecem na fila de

mensagem, e continuarão lá quando forem abertas novamente, a menos que sejam abertas antes por outro processo ou *thread*.

No envio das mensagens para uma fila deve-se passar, como parâmetros, o identificador da fila, a mensagem e o valor da prioridade associada a esta mensagem. Ela será bloqueante ou não, dependendo do que foi definido na criação/acesso da fila. Para o recebimento de mensagens que estejam armazenadas em uma fila é necessário apenas passar o nome da fila como parâmetro, retornando a mensagem e o valor da prioridade associada a esta. Assim como no envio, o recebimento das mensagens pode ser bloqueante ou não.

4. Avaliação de Desempenho do Mecanismo de Comunicação

Para avaliar o desempenho do mecanismo de comunicação utilizou-se a técnica de simulação, no intuito de verificar a sua previsibilidade. A escolha da simulação ocorreu, principalmente, pelo fato dela permitir tratar sistemas com uma maior complexidade e não necessitar de medições do sistema já concluído. Além disso, possibilita a construção de teorias e hipóteses quanto a viabilidade do funcionamento do sistema, sem necessitar o comprometimento de recursos.

4.1 Simulação do Mecanismo de Comunicação CRUX

A simulação do mecanismo de comunicação seguiu os procedimentos que são propostos em [13]:

- estabelecimento dos objetivos e definição do sistema: o desempenho do sistema será medido através do número de mensagens atendidas no tempo adequado (antes de seus *deadlines*).
- lista dos serviços e resultados: cada mensagem gerada em um nó de trabalho tem um *deadline* associado, que uma vez não atendido provoca a perda da mensagem.
- seleção das medidas: as medidas verificadas são o número de mensagens atendidas e o número de mensagens perdidas, dentro de suas restrições temporais.
- definição dos parâmetros: os experimentos são definidos através do número de canais utilizados, do tamanho das mensagens e da taxa de geração dessas mensagens.
- escolha dos fatores: os fatores que definem os experimentos são os seguintes:
  - ⇨ número de NT: 8
  - ⇨ tempo de processamento: 50
  - ⇨ tempo de ocupação do NC: 1
  - ⇨ freqüência de *pooling* do BS: 2
  - ⇨ tamanho da mensagem: 32, 64, 128, 256 e 512
  - ⇨ número de canais[1]: 1, 2 e 4
  - ⇨ taxa de chegada de mensagens: taxa[2]
- seleção da técnica de avaliação: a técnica utilizada foi a simulação, e o *software* utilizado foi o Arena, versão 3, da empresa *System Modeling Corporation* (número de série 9510001). O Arena executa a linguagem de simulação SIMAN V, contém vários recursos para simulação e inclui facilidades como um ambiente de programação gráfica, recursos de animação e relatórios de resultados.

[1] No modelo estático não há uma variação no número de canais dentro de um mesmo experimento, enquanto no modelo dinâmico, o número de canais será definido de acordo com as restrições temporais, ou seja, será aquele necessário para completar a comunicação antes do *deadline* de cada mensagem.

[2] Varia de um valor mínimo a um valor máximo. O valor mínimo corresponde ao tempo mínimo estimado para que a mensagem percorra o sistema. O valor máximo equivale ao tempo estimado com todos os atrasos conhecidos.

- escolha da carga de trabalho: a taxa de geração de mensagens nos NT utilizada varia do valor de menor tempo estimado no sistema ao valor de maior tempo estimado no sistema.
- execução dos experimentos: os experimentos foram realizados em um microcomputador *Pentium* 100.
- análise e interpretações dos dados: os resultados dos experimentos foram colhidos e estão apresentados em 0.

4.2 O Modelo do Mecanismo de Comunicação

O mecanismo de comunicação do CRUX foi modelado de forma estática e de forma dinâmica. O diferencial, entre estes modelos, é a forma como os canais são utilizados em cada experimento. No modelo estático o número de canais não é alterado em um mesmo experimento, enquanto no dinâmico o número de canais será escolhido de acordo com as restrições temporais, ou seja, cada mensagem utilizará um número de canais que possibilite o atendimento do seu *deadline*, dentro do limite de canais existentes e disponíveis. Os elementos básicos (Figura 5) de ambos os modelos são idênticos, diferenciando apenas a estrutura do Nó de Controle (NC) onde é verificado a restrição temporal da mensagem para determinação do número de canais necessários (modelo dinâmico).

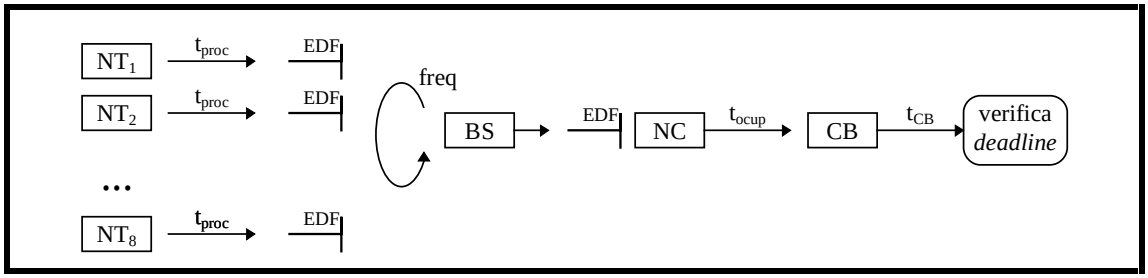

Figura 5- Modelo de comunicação CRUX

O NT é responsável pela geração das mensagens, que são enviadas após um tempo de processamento t_{proc} para um outro NT destino. As mensagens são ordenadas de acordo com o *deadline* associado na geração de cada uma delas. A ordenação é feita de acordo com o *deadline* mais próximo (*earliest deadline first*- EDF). Enquanto ocorre o "enfileiramento" das mensagens de cada NT, o NC verifica, através do BS, se os NT tem mensagens para enviar. Essa verificação ocorre em um intervalo de freqüência *freq* entre um NT e o seguinte. O NC é um recurso de capacidade unitária, ou seja, ele somente atende uma requisição de conexão por vez. O NC verifica se o NT destino e o CB tem disponibilidade de canais para efetivar a comunicação. A comunicação é efetivada quando existe a disponibilidade de recursos, sendo o tempo gasto na comunicação proporcional ao tamanho da mensagem e ao número de canais utilizados. No modelo dinâmico é feito também a verificação do número de canais necessários para cumprir o *deadline* de cada mensagem, com base no tempo estimado para a comunicação. O NC verifica se com a utilização de um canal o *deadline* será atendido, senão testa com dois ou mais canais, até o limite da capacidade de cada NT. Se, mesmo com a utilização do número máximo de canais, não for possível cumprir o *deadline*, a mensagem é descartada. Obtido o número de canais necessários, é verificado a disponibilidade de recursos e feita a alocação destes. O Crossbar possui 32 canais, o que permite 16 conexões simultâneas entre pares de canais dos NT. O tempo gasto no CB (t_{CB}) será o tempo de comunicação (equivalente ao tamanho da mensagem: 1 *byte* / unidade de tempo) dividido pelo número de canais utilizado para o envio da mensagem.

Decorrido o tempo de comunicação é contabilizado o número de mensagens que atendeu os seus respectivos *deadlines*. A partir do levantamento do tempo da mensagem no sistema, pode-se estabelecer o valor de *deadline* que pode ser atendido pelo sistema. Assim, as mensagens recebem valores de *deadline*, dentro de um intervalo que varia do tempo mínimo ao tempo

máximo. Essa atribuição do valor de *deadline* para cada mensagem ocorre de forma dinâmica e segue uma distribuição do tipo uniforme.

Com base nos parâmetros e fatores utilizados é possível se estimar o tempo mínimo e o tempo máximo que uma mensagem permanece no sistema. O tempo despendido no sistema é o somatório do tempo de processamento (t_{proc}), do tempo de *pooling* no BS (*freq*), do tempo de ocupação do NC (t_{ocup}) e do tempo no CB (t_{CB}), conforme mostrado em (1).

$$t_{sistema} = t_{proc} + (x_1 * freq) + (x_2 * t_{ocup}) + t_{CB} \qquad (1)$$

O tempo no CB (t_{CB}) é dependente do tamanho da mensagem (t_{comm}) e do número de canais utilizados, conforme mostra a equação (2).

$$t_{CB} = t_{comm} / canais \qquad (2)$$

O tempo mínimo (3) é aquele onde a mensagem é transmitida sem nenhum atraso (x_1=1, x_2=1), ou seja, assim que a mensagem é gerada no NT é enviada ao NC, e depois do tempo de comunicação é retirada do sistema.

$$t_{minimo} = t_{proc} + freq + t_{ocup} + t_{CB} \qquad (3)$$

O tempo máximo (4) é aquele onde ocorrem os atrasos máximos (x_1=número de NT, x_2=número de NT * número de canais por NT), ou seja, a mensagem tem de aguardar que todas as outras mensagens que possam existir tenham sido atendidas.

$$t_{maximo} = t_{proc} + (NT * freq) + (NT * canais * t_{ocup}) + t_{CB} \qquad (4)$$

4.3 Análise dos Resultados

Considerando o modelo estático e pela análise da variância pode-se confirmar a expectativa de redução no percentual de mensagens descartadas (em virtude do não atendimento ao *deadline*), quando se aumentou o número de canais utilizados. Em relação ao aumento no tamanho das mensagens verificou-se um aumento no número de descartes por não atendimento aos *deadlines*. No entanto, a variação no valor das taxas de geração de mensagens não representou alteração significativa no percentual de mensagens efetivadas. O mesmo pode ser verificado para o modelo dinâmico.

O aumento no número de canais ocasiona uma maior efetivação das mensagens no modelo estático e no modelo dinâmico, conforme observado na Figura 6, em virtude da redução no tempo de permanência da mensagem no canal. A diferença apresentada (para quatro canais), entre o modelo estático e o modelo dinâmico, é explicada pelo fato de no modelo dinâmico só é utilizado o número necessário de canais para o cumprimento do *deadline*, ou seja, somente aquelas mensagens, que necessitam, ocupam os quatro canais. No modelo estático o número de canais é atribuído de forma fixa para cada experimento, o que ocasiona a ocupação, algumas vezes, de um número maior de canais do que o necessário. Essa utilização de um maior número de canais traz embutida o tempo gasto para a conexão, que pode provocar uma demora maior do que ocorreria na utilização de um número menor de canais. Analisando a variação do percentual de comunicações efetivadas em relação ao tamanho das mensagens (Figura 7), observa-se que uma discreta redução do percentual de efetivação ocorre quando se aumenta o tamanho das mensagens. Essa variação é observada em ambos os modelos.

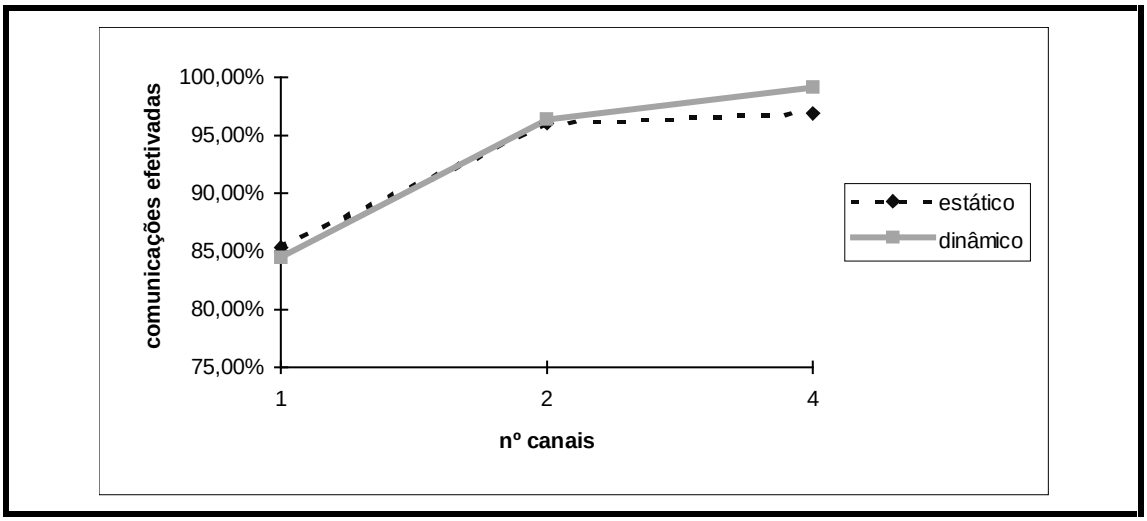

Figura 6 - Percentual de mensagens efetivadas de acordo com o número de canais

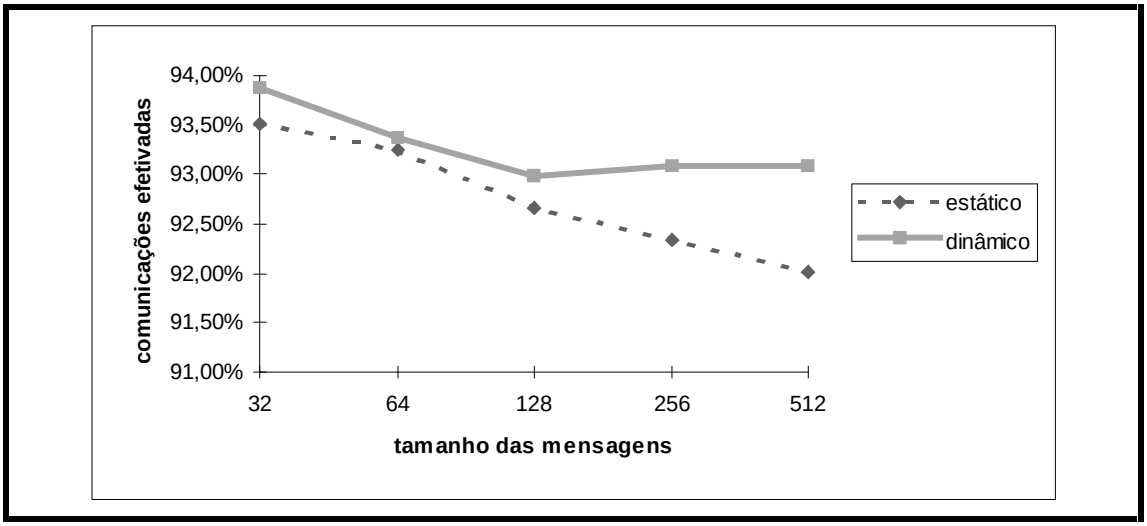

Figura 7 - Percentual de mensagens efetivadas de acordo com o tamanho (em *bytes*)

5. Considerações Finais

Aplicações de tempo real fazem, cada vez mais, parte do dia-a-dia e sua utilização vem ocorrendo em um número maior de áreas nos últimos tempos. O suporte a essas aplicações no ambiente CRUX permite o desenvolvimento de aplicações com restrições temporais em um ambiente de processamento paralelo. Essas aplicações envolvem um espectro variado de assuntos, tais como: multimídia, telecomunicações, robótica, controle de manufatura, controle de processos, sistemas médico-hospitalares, dentre outros.

Através do estudo de sistemas operacionais de tempo real já existentes e do padrão POSIX foi possível apresentar um conjunto básico de primitivas de suporte para tempo real para o ambiente CRUX. Além da definição do conjunto de primitivas, foi realizada a avaliação de desempenho, através de simulação, para analisar o desempenho e a previsibilidade dos mecanismos de comunicação. A importância dessa análise está no fato de que, em geral, o ponto fundamental em aplicações de tempo real encontra-se na previsibilidade, e somente em plano secundário existe uma preocupação com a performance (velocidade de computação, de comunicação, de acesso ao sistema de arquivos, dentre outros), pois apesar da velocidade ser desejável, os SOTR devem levar em consideração apenas o desempenho no pior caso.

Em virtude da grande abrangência do POSIX, de apenas uma parte desse padrão ser obrigatória, da peculiaridade de ambientes e aplicações ou da atual inexistência de

padronização em determinados pontos, a realidade mostra que o suporte total é algo inviável, pois, em geral, a inclusão de todas as primitivas do padrão torna o sistema operacional "pesado". Outro fator relevante, em se tratando de sistemas de tempo real, é a necessidade de garantia temporal. A padronização POSIX permite portabilidade de plataformas, mas não garante uma "portabilidade temporal". Esses fatos contribuíram para que este trabalho também apresentasse uma "compatibilidade parcial" com o padrão, isto é, as primitivas propostas são compatíveis com o padrão, mas são em um número mínimo, de forma a atender as necessidades de suporte de tempo real para a máquina CRUX, não sendo no número proposto no POSIX. Essa "compatibilidade parcial" não ocorre somente no CRUX, mas na maioria dos sistemas de tempo real existentes [14], o que reforça os argumentos apresentados como causa para a não compatibilidade total.

6. Referências Bibliográficas

[1] Shin, K. G., **Real-time computing: a new discipline of computer science and engineering**. Proceedings of IEEE, v. 82, n. 1, Jan 1994.

[2] Stankovic, J. A., **Misconceptions about real-time computing: a serious problem for next generation systems**. IEEE Computer, v. 21, n. 10, p. 10-19, Oct 1988.

[3] Wilner, D. **Vx-Files: what really happened on Mars?**. Keynote speaker, In: The 18th IEEE Real-Time Systems Symposium, (http://cs-www.bu.edu/ftp/IEEE-RTTC/bboard/rtss97/#keynote) San Francisco/CA-USA, 1997.

[4] Montez, Carlos Barros, **Um sistema operacional com micronúcleo distribuído e um simulador multiprogramado de multicomputador**. Dissertação de mestrado, CPGCC/UFSC, Florianópolis/SC-Brasil, mai 1995.

[5] Campos, Rodrigo A., **Um sistema operacional fundamentado no modelo cliente-servidor e um simulador multiprogramado para multicomputador**. Dissertação de mestrado, CPGCC/UFSC, Florianópolis/SC-Brasil, jun 1995.

[6] Gallmeister, B. O., **POSIX.4: programming for the real world**. O'Reilly & Associates, 1995.

[7] Nichols, B., Buttlar D. e Farrel J., **Pthreads programming**. O'Reilly & Associates, 1996.

[8] Laplante, P. A., **Real-time systems design and analysis: an engineer's handbook**, 2. ed., IEEE Press: Los Alamitos/CA-USA, 1997.

[9] Corrêa, Edgard de Faria, **Aplicações de tempo real em um ambiente baseado em multicomputador: serviços de suporte e avaliação de desempenho**. Dissertação de mestrado, CPGCC/UFSC, Florianópolis/SC-Brasil, mar 1998.

[10] Liu, C. L., Layland, J. W., **Scheduling algorithms for multiprogramming in a hard real-time environment**. Journal of the ACM, v. 20, n. 1, p. 46-61, Jan 1973.

[11] Leung, J. Y. T., Whitehead, J., **On the complexity of fixed-priority scheduling of periodic, real-time tasks**. Performance Evaluation, v. 2, n. 4, p. 237-250, Dec, 1982.

[12] Lo, S. L. A., Hutchinson, N. C. e Chanson, S. T., **A flexible real-time scheduling abstraction: design and implementation**. Software-Pratice and Experience, v. 27, n. 9, p. 1055-1066, Sep 1997.

[13] Jain, R. **The art of computer systems performance analysis**. John Wiley & Sons Inc., 1991.

[14] Corrêa, Edgard de Faria e Friedrich, Luis Fernando, **Padronização POSIX *x* sistemas operacionais de tempo real: uma análise comparativa**. III Congreso Argentino de Ciencias de la Computación, La Plata/BA-Argentina, oct 1997.