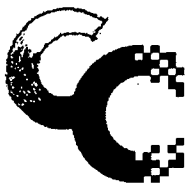

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Aplicações de Tempo Real em um Ambiente Baseado em Multicomputador: Serviços de Suporte e Avaliação de Desempenho

Dissertação submetida à Universidade Federal de Santa Catarina
para a obtenção do grau de Mestre em Ciência da Computação

Edgard de Faria Corrêa

Florianópolis, março de 1998.

Aplicações de Tempo Real em um Ambiente Baseado em Multicomputador: Serviços de Suporte e Avaliação de Desempenho

Edgard de Faria Corrêa

Esta dissertação foi julgada adequada para obtenção do título de

Mestre em Ciência da Computação

especialidade **Sistemas de Computação**

e aprovada em sua forma final pelo Curso de Pós-Graduação em Ciência da Computação

Prof. Luis Fernando Friedrich, Dr.
Orientador

Prof. Jorge Muniz Barreto, Dr.
Coordenador do Curso

Banca Examinadora:

Prof. Luis Fernando Friedrich, Dr. (Presidente)

Prof. Paulo José de Freitas Filho, Dr.

Prof. Thadeu Botteri Corso, M.Sc.

Prof. Rômulo Silva de Oliveira, Dr.

Dedicatória

A vida e o tempo
nos levam a sermos
céticos e conformados.
É preciso que lutemos sempre
pelos nossos ideais e sejamos...

... eternos inconformados.

A meus pais, irmãos e

ao meu avô Juvenal,

no ano do seu 90º aniversário.

Agradecimentos

A Deus.

À minha família pelo apoio, carinho e incentivo ao longo de minha vida.

Ao meu orientador Friedrich pela paciência e dedicação durante o desenvolvimento deste trabalho.

Ao professor Paulo Freitas pela paciência e valiosa contribuição.

Aos professores Thadeu e Rômulo pela colaboração.

A professora Sílvia Nassar pela ajuda no *sprint* final.

Aos professores Murilo, Mazzola e Beth pela amizade e incentivo.

Aos professores Jorge Dantas e Sérgio Fialho (UFRN) pela força, incentivo e apoio.

Ao amigo e grande colaborador Carlos Montez pelas diversas dicas e esclarecimentos.

À Viviana e ao Leandro pela acolhida e amizade desde os primeiros até os "últimos dias" desses anos em Floripa.

Ao Adhemar pela força e colaboração.

À Verinha e à Valdete pela atenção, amizade e bom humor constantes.

Ao Hoff e ao Sobral pelo incontáveis *helps* no laboratório.

À Mirian e à Solange pelo apoio, incentivo e amizade.

À galera *del* Paraguai 201 pela convivência, amizade e companheirismo.

Aos amigos nesses anos de Floripa: Diego, Cerutti, Márcio, Caimi, Bertini, Loureiro, entre outros.

Aos colegas de baias e mestrado pela agradável convivência e amizade.

À UFRN pela oportunidade e por acreditar na necessidade de mais e melhor qualificação profissional.

"Quando um homem
tem um relógio
ele sabe que horas são.
Se ele tem dois relógios
nunca tem certeza disso."

Sumário

Lista de Ilustrações

Lista de Tabelas

Lista de Abreviaturas e Siglas

ANSI	*American National Standards Institute*
BS	Barramento de Serviço
CB	*Crossbar*
CPU	*Central Process Unit*
EDF	*Earliest Deadline First*
FIFO	*First In First Out*
IEEE	*Institute of Electrical and Electronics Engineers*
ISSO	*International Standardization Organization*
MFT	Multiprogramação com um número Fixo de Tarefas
MVT	Multiprogramação com um número Variável de Tarefas
NASA	*National Aeronautics & Space Administration*
NC	Nó de Controle
NCE	Nó de Comunicação Externa
NT	Nó de Trabalho
POSIX	*Portable Operating System Interface*
RAM	*Random Access Memory*
ROM	*Read Only Memory*
SOTR	Sistemas Operacionais de Tempo Real
SRTF	*Shortest Run Time First / Shortest Remaining Time First*
STR	Sistemas de Tempo Real

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 trabalho está inserido no contexto do projeto CRUX, em desenvolvimento no Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina, que visa a concepção de um ambiente completo para programação paralela. Esse projeto inclui a construção de um multicomputador que servirá como base para o projeto e a implementação de um sistema operacional e uma linguagem de programação paralela. Apresenta-se neste trabalho um conjunto básico de primitivas de suporte para aplicações de tempo real no ambiente do multicomputador CRUX baseados no padrão POSIX da IEEE. Com o fornecimento desse conjunto mínimo de primitivas objetiva-se adequar o ambiente de processamento paralelo para a execução de aplicações de tempo real. Em complemento, 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 work is inserted in the context of the project CRUX, on development in the Computer Science Department of the Federal University of Santa Catarina. The project aims the conception of a complete environment for parallel programming, including the construction of a multicomputer, the implementation of an operating system and a language of parallel programming. This work presents a basic group of primitives of support for real time applications in the environment of the multicomputer CRUX. The primitives of support are based on the standard POSIX of IEEE. 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 analyze the performance and the predictability of the communication mechanism.

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 [Stankovic 1988]. 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 [Shin 1994]: 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.

A forte interação de um sistema tempo real com o seu ambiente pode ser visualizada através de algumas de suas características [Audsley 1990]:

- um STR interage de forma próxima e fortemente acoplada ao seu ambiente. Uma computação ativada por um estímulo do ambiente deve ser completada até o seu prazo final (*deadline*);
- em muitas aplicações é impossível exercer controle explícito sobre um ambiente tempo real.

Dentre os diversos conceitos errôneos sobre STR citados em [Stankovic 1988], 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.

De acordo com os critérios de segurança, segundo a intensidade e repercussão das falhas, podemos classificar os sistemas de tempo real em:

- sistema tempo real *soft*: uma falha temporal é da mesma ordem de grandeza que os benefícios do sistema em operação normal, ou seja, não há uma "destruição" do sistema.
- sistema tempo real *hard*: as conseqüências de uma falha temporal excedem em muito os benefícios normais obtidos pelo funcionamento correto do sistema, ou seja, é fatal para o sistema.

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 capaz 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* (multímidia, 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 [Wilner 1997].

Este trabalho está inserido no contexto do projeto CRUX, em desenvolvimento no Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina, que objetiva a concepção de um ambiente completo para programação paralela. Esse projeto inclui a construção de um multicomputador que servirá como base para o projeto e a implementação de um sistema operacional e uma linguagem de programação paralela.

Este trabalho visa investigar as principais questões envolvidas em SOTR, adequando o ambiente CRUX para a execução de aplicações de tempo real através do fornecimento de algumas primitivas básicas de SOTR. Dessa forma, pretende-se permitir o desenvolvimento de aplicações com restrições temporais, tais como: multimídia, sistemas de controle, dentre outros.

Além deste capítulo inicial, este trabalho contém outros 5 capítulos. O capítulo 2 trata de SOTR, descrevendo seus componentes e a padronização (POSIX) proposta para esses sistemas. O capítulo 3 apresenta o ambiente de aplicação da máquina CRUX, descrevendo sua arquitetura e sistema operacional. O capítulo 4 apresenta o conjunto básico de primitivas de suporte para tempo real para o ambiente CRUX, que é o objetivo central deste trabalho. No capítulo 5 é feita uma análise de desempenho dos mecanismos de comunicação, através da simulação de seus parâmetros e de sua previsibilidade. As conclusões deste trabalho e as propostas para trabalhos futuros são apresentadas no capítulo 6.

2. Sistemas Operacionais de Tempo Real

2.1 Introdução

Dentre os diversos requisitos de projeto em STR, alguns podem ser ressaltados, tais como [Tanenbaum 1995]: sincronização de relógios; previsibilidade; suporte a linguagens; tolerância a falhas; e "orientação" do sistema (disparado por evento ou disparado pelo tempo).

SINCRONIZAÇÃO DE RELÓGIOS

A sincronização de relógios é um fator importante, principalmente, quando tratamos de STR em ambientes distribuídos. Pois, um sistema distribuído possui relógios locais, independentes entre si, que apresentam diferentes desvios em relação a uma fonte externa, fornecedora do "tempo real". Isso faz com que, mesmo que todos os relógios partissem em um instante específico com o mesmo valor de tempo, rapidamente iriam divergir em seus valores.

O tempo é a base do correto funcionamento STR e a divergência de valores nos diversos relógios em um ambiente distribuído pode ser problemática. A necessidade de manter próximos os valores dos relógios locais, leva a existência de algoritmos de sincronização interna de relógios e algoritmos de sincronização externa de relógios.

A sincronização interna de relógios visa corrigir periodicamente os relógios locais, mantendo-os próximos entre si. Ela envolve trocas de mensagens entre os nós possuidores de relógios locais. A sincronização externa de relógios faz-se necessária quando a aplicação precisa manter os relógios com valores próximos aos originados por uma fonte externa. Essa sincronização externa resulta em uma sincronização interna de relógios, mas o mesmo não ocorre em sentido inverso.

PREVISIBILIDADE

Uma das mais importantes propriedades em qualquer STR é a necessidade de previsibilidade do comportamento temporal do sistema. Essa previsibilidade pode ser de natureza determinista ou probabilista [Montez 1997].

A previsibilidade determinista envolve o conhecimento antecipado da carga computacional do sistema, através de uma análise estática do comportamento temporal do mesmo, diante dos recursos disponíveis. Isso implica em considerar os picos de carga computacional, atrasos máximos nas comunicações, tempos máximos de acesso a dispositivos de entrada/saída, etc. Cada análise de pior caso deve considerar as hipóteses de faltas do sistema, isto é, assumir no sistema os tipos e as freqüências com que essas faltas podem vir a ocorrer.

Na impossibilidade de prever-se o comportamento da carga e as hipóteses de faltas, a previsibilidade é probabilista. O conhecimento do provável comportamento de um sistema pode ser obtido através de simulações. Simulações, entretanto, nunca poderão garantir que todas as situações extremas de funcionamento do sistema ocorreram durante o experimento.

SUPORTE A LINGUAGENS

É desejável que os requisitos temporais de um STR possam ser tratados de forma adequada, necessitando o suporte a uma linguagem apropriada, que possibilite a utilização de recursos para tratar as características de tempo real. Uma linguagem que trate, por exemplo, *threads* e as escalone independentemente, que atenda as restrições de exclusão mútua.

Linguagens de tempo real devem ter a capacidade de expressar atrasos máximos e mínimos (como por exemplo o atraso máximo para que um processo seja suspenso) ou se um determinado evento ocorre ou não dentro de um certo intervalo de tempo.

De forma similar, se uma mensagem enviada não tem confirmação de recebimento depois de decorrido um certo tempo (*timeout*) deve haver o desbloqueio do emissor nos casos de envio bloqueante.

TOLERÂNCIA A FALTAS

Muitos STR controlam os serviços críticos de segurança em veículos, aviões, hospitais e usinas. Conseqüentemente, a tolerância a faltas é um requisito freqüente.

Em sistemas críticos em segurança, é especialmente importante que o sistema possa ser capaz de tratar o cenário de pior caso. Não é suficiente dizer que a probabilidade de que um determinado número de componentes falhem ao mesmo tempo é tão pequena que possa ser ignorada. É relevante destacar que raramente as falhas são independentes. Em uma planta nuclear, por exemplo, alarmes de temperatura, de pressão, de radioatividade podem ser disparados ao mesmo tempo devido ao rompimento de um simples duto. Assim, STR tolerante a faltas devem ser capazes de tratar com a ocorrência simultânea do número máximo de falhas e da carga máxima de processamento [Kopetz 1993].

ORIENTAÇÃO POR EVENTOS OU POR TEMPO

Em um STR disparado por eventos, quando um evento significante ocorre, por exemplo, no ambiente externo, ele é detectado por algum sensor, causando uma interrupção. Sistemas disparados a eventos são, portanto, orientados a interrupções.

Os sistemas disparados por eventos funcionam adequadamente para o STR *soft* com um poder computacional reservado. Em sistemas mais complexos, entretanto, eles funcionam de forma adequada se o compilador puder analisar o programa e conhecer todos os eventos, sabendo sobre o comportamento do sistema para a ocorrência de cada evento. O principal problema com sistemas disparados por eventos é o fato que eles podem falhar sob condições de carga máxima, isto é, quando muitos eventos ocorrem ao mesmo tempo. No exemplo do duto, o simples rompimento poderá ocasionar o envio de vários alarmes.

Sistemas de tempo real disparados por tempo utilizam uma interrupção de relógio a cada intervalo de ΔT milisegundos. A cada interrupção de relógio, sensores são amostrados e atuadores são acionados, e nenhuma outra interrupção ocorre, além da interrupção de relógio. O intervalo ΔT deve ser escolhido de modo a não ser muito pequeno, o que causaria um grande número de interrupções de relógio, nem muito grande, pois eventos críticos poderiam ser notificados quando já fosse muito tarde. Alguns eventos, também, podem ocorrer em

intervalos de tempo bem menores que o *tick* (ou resolução) do relógio, e precisam ser armazenados para evitar que sejam perdidos. Isso pode ser feito eletricamente por circuitos *flip-flop* ou por microprocessadores embutidos em dispositivos externos.

2.2 Classificação

Em geral, sistemas operacionais de tempo real devem prover escalonamento, execução e comunicação entre tarefas. Essas funções podem ser implementadas por *software* ou *hardware*. Considerando a complexidade dos serviços implementados nos seus núcleos (Figura 2.1), de acordo com [Laplante 1997], os sistemas operacionais de tempo real podem ser classificados em:

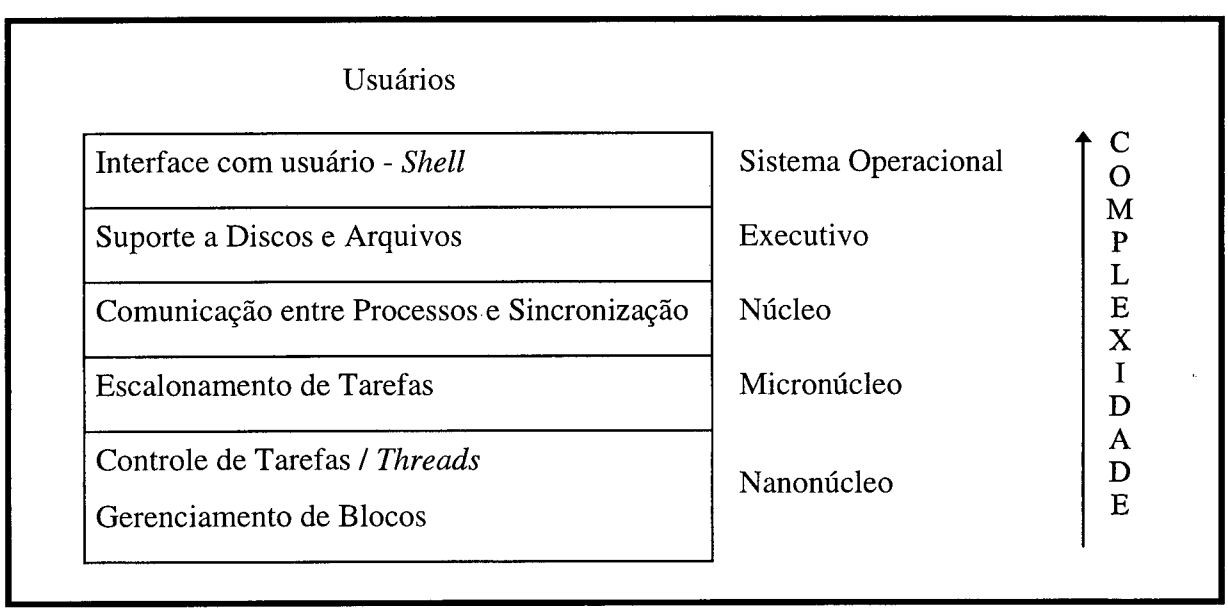

Figura 2.1 - Hierarquia de núcleos tempo real

- nanonúcleo: gerenciamento simples de execução de *threads* (como "fluxo de controle"); provê essencialmente execução de tarefas.
- micronúcleo: um nanonúcleo que fornece escalonamento para tarefas.

- núcleo: um micronúcleo que fornece sincronização e comunicação entre tarefas através de semáforos, caixas postais ou outros métodos.
- executivo: um núcleo que inclui blocos de memórias privativas, serviços de entrada/saída e outras características complexas. A maioria dos núcleos tempo real são na realidade executivos.
- sistema operacional: um executivo que fornece para um usuário em geral interface ou processador de comandos, segurança e um sistema de gerenciamento de arquivos.

Existem diversas classificações propostas por outros autores [Rozier 1991] [Tanenbaum 1992], que consideram fatores como o tamanho ou o número de primitivas suportadas. No entanto, existe na maioria, um consenso de que um núcleo (micronúcleo ou nanonúcleo) deve ser capaz de prover serviços considerados básicos sem acarretar ônus para a sua funcionalidade e portabilidade.

2.3 Descrição dos Componentes

SOTR são parte fundamental de sistemas de tempo real, devendo ser capaz de realizar de forma integrada o escalonamento de CPU e a alocação de recursos de maneira que o conjunto de tarefas cooperantes possam obter os recursos que elas necessitam, no tempo correto, dentro das suas restrições temporais. Esta previsibilidade requer, entre outras coisas, primitivas de sistemas operacionais que sejam limitadas, ou seja, tenham o tempo máximo de execução definido.

A utilização de paradigmas de sistemas operacionais convencionais, que permitem esperas arbitrárias por recursos ou eventos, ou tratam uma tarefa como um processo randômico, não pode ser considerada quando se trata de alcançar requisitos mais complexos.

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.

2.3.1 Gerência de Processos

Processo pode ser definido como um programa em execução, ou seja um conjunto de operações, em ordem seqüencial, sendo executadas.

Do ponto de vista do suporte, o processo é definido pelo bloco de controle. Esse bloco de controle é uma estrutura de dados que contém ou permite localizar todas as informações chaves do processo, tais como: identificação do processo; estado corrente do processo; prioridade; apontador para localizar a memória do processo; apontador para localizar os recursos alocados e área de salvamento de registros.

Um processo ativo pode ter sua execução encerrada por pedido espontâneo de parada ou por interrupção. O pedido espontâneo de parada pode ser um bloqueio, devido a falta de recursos ou dados, ou uma suspensão por encerramento de uma função. A interrupção pode ocorrer, dentre outras razões, devido a um evento ou *timeout*. Tanto no encerramento por pedido espontâneo de parada como no encerramento por interrupção, um outro processo deve ser escolhido para execução. Entretanto, é necessário antes executar um armazenamento do estado do processo interrompido para o posterior retorno (mudança de contexto). O tempo despendido na mudança de contexto é um fator importante para aumentar o tempo de resposta. A minimização desse tempo é feita salvando apenas o estritamente necessário: conteúdo dos registradores e *flags* do processador, endereço da instrução onde o processo foi interrompido, valores das variáveis locais do processo e recursos em uso.

THREADS

Threads são fluxos de execuções independentes no interior de um processo. Os processos tradicionais tem um espaço de endereçamento e uma única *thread* (Figura 2.2a) de controle. [Tanenbaum 1995].

Um grupo de *threads* (Figura 2.2b) semelhantes compartilham código, espaço de endereçamento, e recursos de sistemas operacionais. O ambiente no qual uma *thread* executa é chamado de tarefa ou processo. Um processo não faz nada se não existe *threads* nele, e uma *thread* deve estar contida em um processo. Uma *thread* individual tem ao menos o seu próprio registrador de estado, e geralmente sua própria pilha.

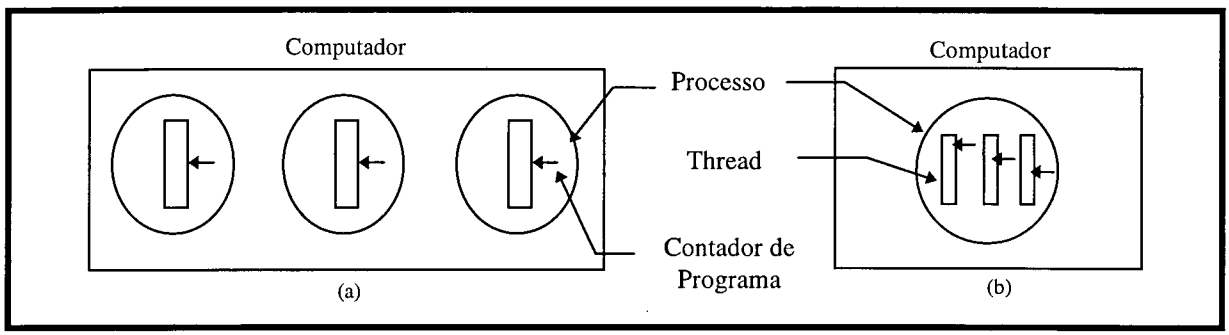

Figura 2.2 - (a) Três processos com uma *thread* cada. (b) Um processo com três *threads*.

O compartilhamento extensivo faz com que a comutação de CPU entre *threads* semelhantes e as criações de *threads* sejam inexpressivas, comparadas com a comutação de contexto entre processos tradicionais. Assim a utilização de *threads* é uma solução razoável para o problema de se manusear eficientemente muitas requisições [Silberschatz 1991].

ESCALONAMENTO

No âmbito de sistemas tempo real, escalonamento pode ser definido como o procedimento de alocar temporalmente recursos computacionais a tarefas [Stemmer 1997]. Sendo o próprio processador o recurso mais importante, o escalonamento pode ser visto, de forma mais restrita, como o procedimento que consiste em decidir qual processo deve receber o controle do processador em um dado momento, em função da ocorrência de eventos internos ou externos.

Considerando a preempção ou não do processador, podemos ter uma classificação da política de escalonamento em dois tipos (Figura 2.3):

- escalonamento não-preemptivo: o processo que faz uso do processador não pode perder o controle deste, exceto por iniciativa própria. A realocação do processador acontece somente após o término do processo em curso (ativo). A política de não-preempção é mais simples de implementar, evita *overhead* de chaveamento de processos e, geralmente, é mais previsível. Por outro lado, algumas tarefas podem apresentar um tempo de resposta muito longo devido ao fato de esperar pelo término de outras.

- escalonamento preemptivo: o processador pode ser tomado do processo em curso em função da ocorrência de um evento físico ou lógico. A preempção pode ser orientada por tempo (somente interrupções do relógio afetam o escalonamento) ou por eventos (tanto interrupções do relógio quanto do ambiente afetam o escalonamento). A política de preempção é útil para sistemas com processos de alta prioridade que requerem atendimento rápido (alarmes, controladores) como é o caso dos sistemas tempo real. Por outro lado, essa política tem alto custo devido à complexidade de implementação e *overhead* do chaveamento de processos.c

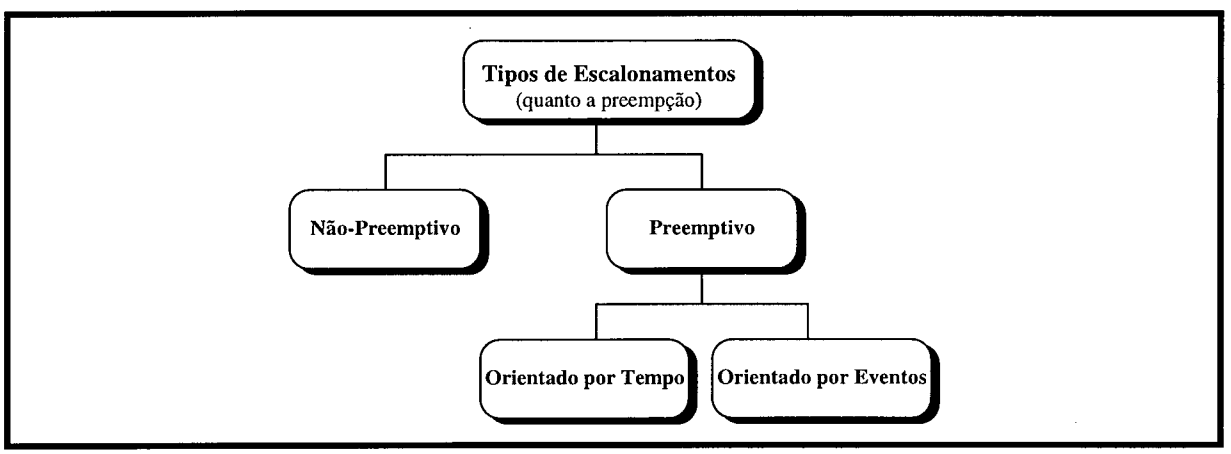

Figura 2.3 - Técnicas de escalonamento, quanto a preempção

De acordo com essa classificação (Figura 2.3) [Stemmer 1997], algumas técnicas utilizadas para implementar essas políticas de escalonamento podem ser citadas:

- FIFO (*First-in, First-out*): o primeiro processo a chegar será o primeiro a ser executado (primeiro que chega, primeiro que sai). Esse tipo de escalonamento não-preemptivo tem o inconveniente de que tarefas urgentes ou tarefas de curta duração podem ter que esperar um longo tempo para serem atendidas.

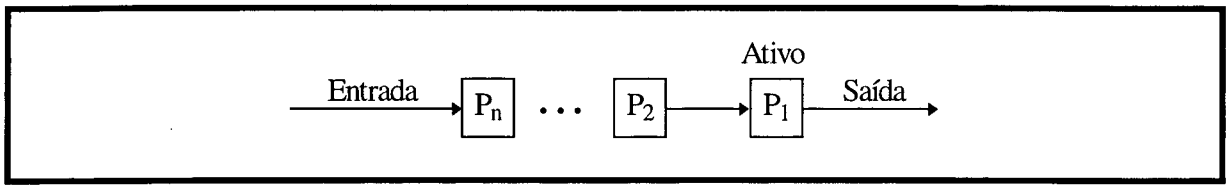

Figura 2.4 - Fila de processos prontos para FIFO

- circular ou Round-Robin: esse escalonamento é preemptivo orientado pelo tempo. Ele não usa prioridades, tem uma única fila de processos prontos, e cada processo tem um tempo determinado de execução Q (*quantum* de tempo), controlado pelo relógio. Expirado esse *quantum* o processo é interrompido e enviado ao fim da fila. Esse tipo de escalonamento favorece processos curtos, apresentando *overhead* quando há muitos processos longos, sendo bastante utilizado em sistemas de tempo compartilhado (*time sharing*).

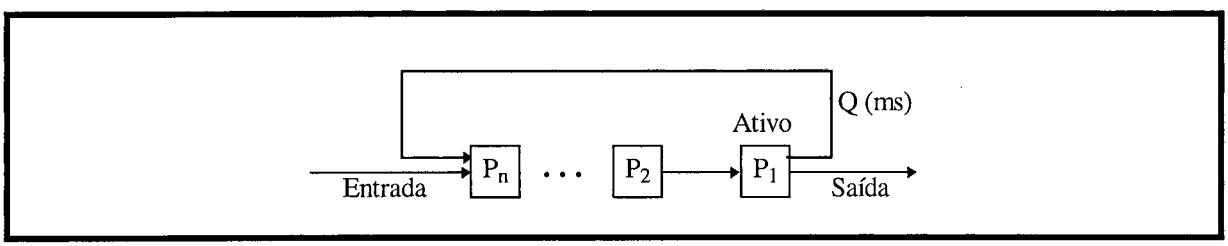

Figura 2.5 - Fila de processos prontos para Round-Robin

- cascata (*foreground/background*): é preemptivo, orientado pelo tempo. Utiliza diversas filas de execução com prioridades decrescentes, onde a fila de maior prioridade tem um *quantum* menor. Caso o processo não termine no tempo estipulado para a fila em que se encontra é passado para a fila com prioridade imediatamente inferior. Os processos das filas de baixa prioridade só recebem o controle do processador quando as filas de prioridade superior estiverem vazias. A última fila utiliza escalonamento Round-Robin. Esse tipo de escalonamento diminui o *overhead* para processos longos em relação ao escalonamento circular.

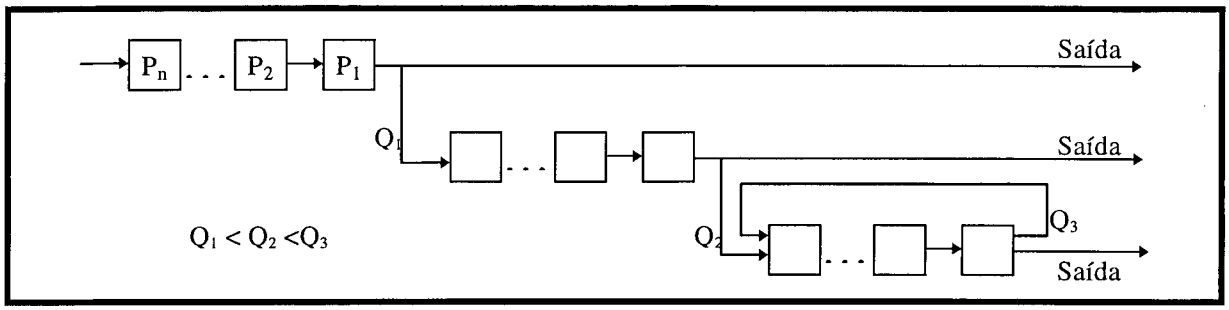

Figura 2.6 - Fila de processos prontos para Escalonamento em Cascata

- orientado a eventos (*Event Trigger*): é preemptivo, orientado por eventos. Os processos prontos são organizados por prioridade (urgência de execução), que pode ser dinâmica ou estática. Os eventos podem implicar na remoção ou introdução de novos processos na lista de prontos, cuja fila é reorganizada quando ocorrem eventos que tornem nova tarefa "pronta", ocupando o início da fila e recebendo o processador a tarefa de maior prioridade. Esse é um tipo de escalonamento usado em sistemas tempo real.

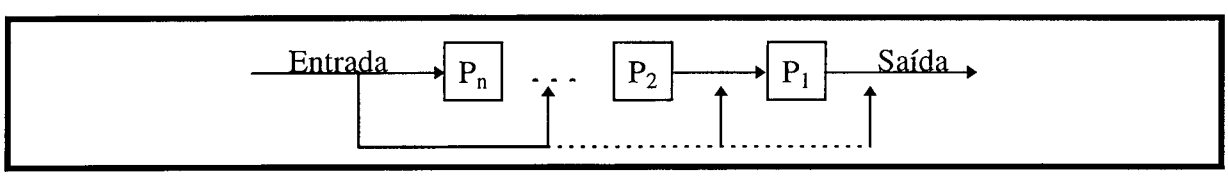

Figura 2.7 - Fila de processos prontos para Escalonamento *Event Trigger*

- SRTF (*Shortest Run Time First / Shortest Remaining Time First*): a prioridade inicial do processo é definida em termos de tempo de execução estimado (tempo menor de execução, executa primeiro). O processo que possui um menor tempo de execução estimado recebe a maior prioridade. Esse tipo de escalonamento é preemptivo, e cada vez que o processo se põe em espera ou é interrompido, o tempo de execução decorrido é subtraído da estimativa inicial, aumentando a prioridade. A necessidade de registrar o tempo decorrido gera *overhead*. Essa política de escalonamento é utilizada em sistemas de tempo real e tempo compartilhado (*time sharing*).

Considerando sistemas operacionais tempo real, pode-se classificar as possíveis abordagens para escalonamento tempo real, com base em seus aspectos mais relevantes [Oliveira 1997]: carga; previsibilidade; utilização de recursos; momento do cálculo da escala de execução e algoritmos de escalonamento empregados.

As abordagens de escalonamento para tempo real (Figura 2.8), são divididas em dois grupos, de acordo com o tipo de previsibilidade oferecida. O primeiro grupo (garantia no projeto) é aquele capaz de oferecer uma previsibilidade determinista. Este grupo é empregado em sistemas onde é necessário garantir, em tempo de projeto, que todas as tarefas serão executadas dentro de seus *deadlines*. Essa garantia é obtida a partir de hipóteses de carga e de faltas. O segundo grupo (melhor esforço na execução) não oferece garantia, em tempo de projeto, de que os *deadlines* serão cumpridos. O escalonamento de melhor esforço (*best effort*), quando muito, oferece uma previsibilidade probabilista sobre o comportamento temporal do sistema, a partir de uma estimativa de carga. Alguns podem oferecer, em tempo de execução, uma garantia dinâmica de quais prazos serão ou não atendidos.

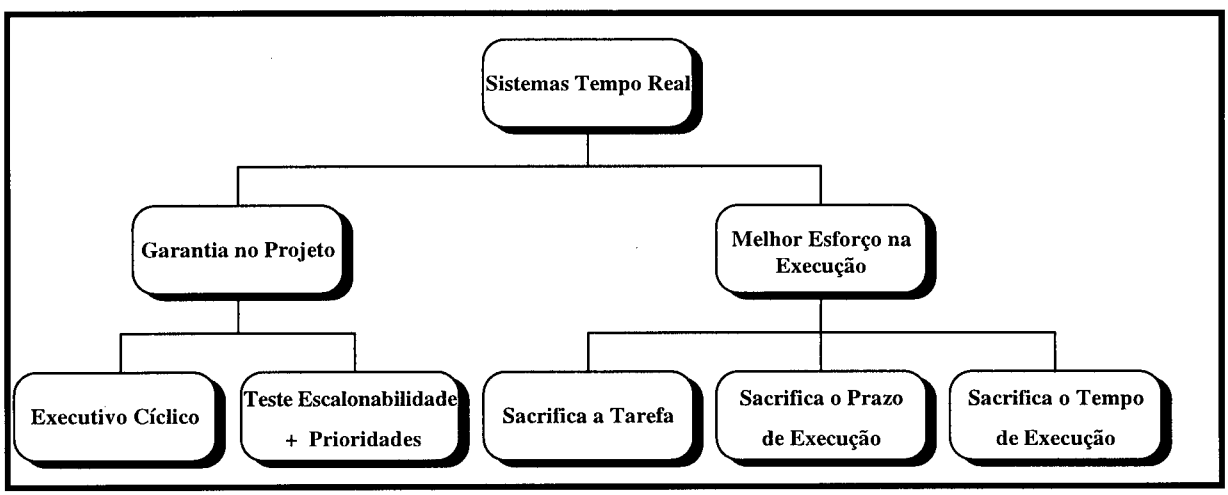

Figura 2.8 - Abordagens de escalonamento para tempo real

Em geral, o escalonamento de um conjunto de tarefas é dividido em: teste de escalonabilidade, que determina se é possível atender os *deadlines*; e o cálculo da própria escala de execução das tarefas. As abordagens baseadas em garantia no projeto têm como característica a execução do teste de escalonabilidade em tempo de projeto (*off-line*).

- executivo cíclico: além do teste, todo o trabalho de escalonamento é feito em tempo de projeto. O resultado é uma grade de execução (*time grid*) que determina qual tarefa executa, em que momento e em qual processador. Essa grade é repetida indefinidamente por um pequeno escalonador denominado executivo cíclico. Qualquer conflito, com por exemplo, utilização de recursos, precedência de tarefas é resolvido durante a construção da grade. Em tempo de execução um programa executivo simplesmente dispara as tarefas no momento indicado pela grade, que é repetida indefinidamente. Assim, é possível garantir que todos os *deadlines* serão cumpridos a partir de uma simples inspeção da grade gerada. Essa grade caracteriza as tarefas periódicas de acordo com os seus períodos, *deadlines*, tempo de computação no pior caso. As tarefas esporádicas são transformadas em periódicas, utilizando a freqüência definida pelo intervalo mínimo entre suas ativações.

- teste de escalonabilidade + prioridades: as tarefas recebem uma prioridade e o teste de escalonabilidade verifica se existe garantia de que todas as tarefas terão seus *deadlines* cumpridos. Esse teste é feito em tempo de projeto. Em tempo de execução, um escalonador preemptivo é responsável por produzir a escala de execução utilizando as prioridades das tarefas. A atribuição de prioridades às tarefas pode ser feita estaticamente, em tempo de projeto, ou dinamicamente, em tempo de execução. No primeiro caso, durante o teste de escalonabilidade, as tarefas vão recebendo valores de prioridades conforme critérios prévios. Na atribuição dinâmica, em determinados momentos, o escalonador verifica a situação temporal de cada tarefa, atribuindo prioridades e escolhendo, para executar, a tarefa de mais alta prioridade naquele momento. Os algoritmos para escalonamento mais clássicos são [Liu 1973]: o Taxa Monotônica (*Rate Monotonic*) e o Próximo *Deadline* Primeiro (*Earliest Deadline First*). No primeiro, as prioridades são fixas e maiores para as tarefas de menor período. O segundo algoritmo trabalha com prioridades variáveis, executando antes a tarefa cujo *deadline* está mais próximo naquele instante. Outro algoritmo semelhante é o *Deadline* Monotônico (*Deadline Monotonic*), onde tarefas com *deadline* menor

ou igual ao período de processamento recebem maior prioridade, que são atribuídas em tempo de projeto [Leung 1982].

Essas abordagens que oferecem uma previsibilidade determinista (garantia em tempo de projeto) implicam na necessidade de uma reserva de recursos suficientes para o pior caso de carga computacional. Isso pode representar uma enorme subutilização de recursos, como, por exemplo, nos casos onde o tempo médio de execução é consideravelmente menor que o tempo de execução no pior caso. É exigido o sacrifício de recursos e da flexibilidade do sistema para se obter previsibilidade.

Nas abordagens de melhor esforço (*best effort*) não existe garantia, em tempo de projeto, de que as restrições temporais serão cumpridas. Os testes de escalonabilidade e o cálculo da escala de execução das tarefas são feitos em tempo de execução. Uma conseqüência do uso de abordagens dinâmicas é ocorrência de sobrecargas (*overload*) no sistema, quando, em um determinado instante, os recursos disponíveis não são suficientes para que as tarefas sejam executadas dentro dos seus respectivos prazos. Isso não caracteriza uma situação anormal, mas uma situação esperada em sistemas que utilizam a abordagem do tipo melhor esforço. Assim, existem procedimentos para tratar essas sobrecargas: sacrifício total de algumas tarefas; execução de todas as tarefas com sacrifício do prazo de execução de algumas tarefas; e execução de todas as tarefas com sacrifício do tempo de execução de algumas tarefas.

- sacrifício de tarefas: na chegada de uma nova tarefa (aperiódica) é feito um teste de "aceitação" para verificar se o novo conjunto de tarefas é escalonável. Caso não seja, alguma tarefa é sacrificada [Ramamritham 1989]. Em geral, é sacrificada a tarefa que acabou de chegar, pois até a sua chegada o conjunto de tarefas existentes era escalonável. No sacrifício de uma outra tarefa, que não seja a recém-chegada, torna-se necessário a realização de um teste de escalonabilidade do novo conjunto, que implica em gasto de tempo do processador. Nesse tipo de abordagem são utilizados alguns algoritmos baseados em heurísticas.
- sacrifício do prazo de execução: a tarefa é aceita no conjunto de tarefas, mesmo que algumas delas percam seus *deadlines*. Essa abordagem considera um modelo

de tarefas onde existe um benefício mesmo que a tarefa conclua sua execução depois do seu *deadline* [Jensen 1985].

- sacrifício de tempo de execução: quando ocorre a sobrecarga, o tempo de computação de algumas tarefas é sacrificado. É considerado um modelo de tarefas, onde o benefício resultante da computação assume valores crescentes conforme o tempo de execução. Em função do resultado da computação não poder ser determinado com exatidão *a priori*, essa abordagem costuma ser denominada técnica de computação imprecisa.

Existe um grande número de abordagens de escalonamento. Deve-se destacar, principalmente, o fato das abordagens para sistemas operacionais convencionais objetivarem o uso equalitário dos recursos (aqui, especificamente, o processador). Já os sistemas operacionais de tempo real buscam a garantia das restrições temporais das tarefas, ocasionando, muitas vezes, o privilégio no uso dos recursos pelas tarefas prioritárias.

2.3.2 Comunicação entre Processos / Sincronização

Durante a execução, um processo pode necessitar interagir com outros processos para trocar informações ou para tentar acessar recursos comuns. O primeiro caso, a cooperação, ocorre quando um processo está aguardando recursos fornecidos por outro processo. Quando os processos tentam acessar recursos comuns eles estão em competição.

Na cooperação, os processos que são assíncronos vão precisar se sincronizar e, eventualmente, se comunicar durante a sincronização. Na competição o acesso ao recurso será garantido por uma exclusão mútua. Ela deve garantir, inclusive, que não haja *deadlock*, que é o impasse gerado pela indisponibilidade dos recursos e bloqueio dos processos, que ocorre quando recursos alocados a um processo não são devolvidos por este ou quando um processo espera pelos resultados de outro processo para prosseguir e este espera por recursos do primeiro.

Os mecanismos utilizados para permitir a cooperação entre tarefas e a competição por recursos são: a comunicação através de área compartilhada de memória e a comunicação através de passagem de mensagem.

COMUNICAÇÃO ATRAVÉS DE ÁREA COMUM DE MEMÓRIA (OU VARIÁVEIS COMPARTILHADAS)

Os processos compartilham variáveis que podem representar o estado de recursos físicos compartilhados por eles ou que podem ser utilizadas para comunicar dados entre processos em cooperação. A utilização de variáveis compartilhadas apresenta problemas relacionados com o acesso coordenado ao recurso compartilhado (exclusão mútua) e com a seqüência de execução dos processos (sincronização condicional).

A exclusão mútua ocorre quando o acesso de um processo a um determinado recurso impede que qualquer outro processo o faça simultaneamente. Em cada processo as seções de código que tratam os dados compartilhados são denominadas de regiões críticas. Quando um processo está executando sua região crítica, os outros processos que compartilham o dado só poderão executar as outras regiões de código. A implementação da exclusão mútua é possível com a utilização de protocolos de entrada e de saída das regiões críticas.

A sincronização condicional é necessária quando é preciso coordenar a execução de processos concorrentes ou passar dados entre eles.

Um mecanismo básico utilizado para permitir a exclusão mútua e a sincronização condicional entre processos é o semáforo, proposto por [Dijkstra 1965]. Um semáforo é uma variável inteira não-negativa que pode ser acessada e alterada somente por duas operações: ***wait*** e ***signal***.

Essas primitivas são executadas de forma atômica, isto é, sem interrupções possíveis. A operação ***wait*** suspende qualquer chamada de programa até que o semáforo seja liberado. A operação ***signal*** libera o semáforo. Esse tipo de semáforo é chamado de binário (assumem valores 0 ou 1) ou *mutex* (*mutual exclusion*), pois usualmente são usados para garantir a exclusão mútua. Outro tipo é o semáforo contador ou geral que assume valores inteiros não negativos. Ele pode ser utilizado para proteger um conjunto de recursos; para saber o número de recursos disponíveis antes que o processamento tempo real possa começar; ou ainda, para controlar o número de acessos a vetores (*buffers*) de tamanho limitado.

COMUNICAÇÃO ATRAVÉS DE PASSAGEM DE MENSAGEM

A comunicação através de passagem de mensagem permite que se realize ao mesmo tempo a comunicação entre processos e a sincronização. A comunicação é obtida pelo envio de uma mensagem de um processo a um outro e pela recepção da mensagem vinda de um processo. A sincronização existe porque as mensagens só podem ser recebidas caso já tenham sido enviadas, o que vai limitar a ordem de ocorrência dos eventos.

Esse tipo de comunicação por troca de mensagens pode ser utilizado tanto em sistemas com vários processadores sem memória comum, interligados por redes de comunicação (sistemas distribuídos, por exemplo) como em sistemas com vários processadores com uma memória comum (multiprocessadores, por exemplo), ou ainda por sistemas com um processador multiplexado entre vários processos (sistemas multitarefas).

As operações primitivas utilizadas na comunicação e sincronização por troca de mensagens são *send* (para envio) e *receive* (para recepção). Na definição dessas primitivas é necessário analisar alguns aspectos: características das mensagens; formas de especificação dos canais de comunicação; tipos de sincronização; tratamento e recuperação de erros.

a) características das mensagens: definição do tamanho e do tipo de mensagens a serem trocadas e do seqüenciamento no recebimento.

b) formas de especificação dos canais de comunicação: os aspectos quanto ao tipo de destino, quanto à escolha do endereçamento e quanto à definição da capacidade do canal.

- o tipo de destino do envio da mensagem pode ser de um processo para um único processo (*um a um*); de vários processos para um único processo (*N a um*); destinado para um grupo de processos (*multicasting*) ou para todos os processos (*broadcasting*).

- a escolha do endereçamento pode ser entre direto ou indireto, e neste último com a definição do tipo de mecanismo a ser utilizado (caixas postais, portas, etc.). O endereçamento direto é simples de implementar e de utilizar, equivalendo a criação automática de um canal único entre cada par de processos que deseja se comunicar. Cada processo só necessita conhecer a identidade do outro. No

endereçamento indireto por caixas postais, as mensagens são enviadas e recebidas em lugares específicos (caixas postais) que são baseados no uso de nomes globais. Uma caixa postal pode ser utilizada como a especificação do destino em qualquer comando *send* e como especificação da origem em qualquer comando *receive*. Portas são tipos especiais de caixas postais, onde todos os comandos *receive* que se referenciam a uma mesma porta devem ocorrer necessariamente no mesmo processo. A forma de endereçamento vai ser definida de acordo com o tipo de destino. O endereçamento direto pode ser utilizado no tipo *um a um*. Já uma comunicação do tipo cliente/servidor necessita, por exemplo, da utilização de portas (*N a um*) para que cada porta tenha um receptor (servidor) que pode ter vários emissores (clientes). Quando houver vários servidores para vários clientes, não será possível a utilização de portas. O indicado seria a utilização de caixas postais.

- a capacidade do canal indica o número de mensagens que podem ser armazenadas temporariamente em filas. Essas filas de mensagens podem ter capacidade zero, limitada ou ilimitada. Quando a capacidade da fila é zero, não existem mensagens armazenadas. O emissor tem de esperar que o receptor tenha recebido a mensagem. Os dois processos devem ser sincronizados para permitir a transferência da mensagem (*rendezvous*). Na capacidade limitada, o emissor só será bloqueado quando a capacidade da fila estiver esgotada. Caso contrário, a mensagem é colocada na fila e o processo emissor continua sua execução sem esperar. Para filas de capacidade ilimitada o processo emissor nunca é bloqueado. A dificuldade neste último caso é prática, pois a memória física de um computador não é ilimitada.

c) quanto ao tipo de sincronização, as primitivas podem ser bloqueantes ou não bloqueantes. As bloqueantes ou síncronas não necessitam armazenar mensagens pois utilizam mecanismo de *rendezvous*, o qual só transfere a mensagem do emissor para o receptor quando o último estiver pronto. A sincronização não-bloqueante ou assíncrona utiliza mecanismos do tipo caixa postal que garante a liberação do emissor logo após o armazenamento da mensagem na fila.

d) tratamento e recuperação de erros: as falhas podem ser de ordem lógica ou física. As lógicas ocorrem devido a um *deadlock* ou pela tentativa de comunicação com um processo já destruído. As falhas físicas ocorrem no sistema de comunicação ou no *hardware* associado ao outro processo envolvido. O modo mais usual no tratamento de falhas é a utilização de *timeout*, ou seja, decorrido um tempo determinado o processo é desbloqueado, evitando que o bloqueio siga indefinidamente.

2.3.3 Gerência de Memória

A CPU executa código em memória principal, logo, todo programa deve ser carregado para poder ser executado. A alocação dos endereços de memória podem ser em tempo de compilação (estática), de carga (código realocável) ou de execução (dinâmica).

Processos de aplicações usam memória explicitamente (requisição para área de armazenamento temporário de memória) ou implicitamente (manutenção de memória em tempo de execução). Sistemas operacionais necessitam de gerenciamento de memória para manter as tarefas isoladas.

Na alocação de memória deve-se tomar o cuidado de não se excluir o determinismo dos sistema, ao mesmo tempo que tenta-se a redução de *overhead* pela alocação de memória [Laplante 1997].

Gerenciamento de memória dinâmica de qualquer tipo em tempo real, ainda que usualmente necessário, é prejudicial para a performance e análise de escalonamento tempo real. Uma alocação descuidada pode prejudicar o determinismo temporal, por exemplo, se insere um *deadlock* no sistema. Além disso, a "coleta de lixo" (*garbage colletion*) usualmente implica em suspender as aplicações por tempos não deterministas.

GERENCIAMENTO DA PILHA DE PROCESSOS

Em um sistema multitarefas, o contexto para cada tarefa necessita ser salvo e restaurado na ordem da troca de processos. Isso pode ser feito com o uso de uma ou mais pilhas em tempo de execução ou com o modelo de bloco de controle de tarefas.

Quando um modelo de bloco de controle de tarefas é utilizado, uma lista de blocos de controle é mantida. Essa lista pode ser fixa ou dinâmica. Em um sistema multitarefas tempo real, o gerenciamento principal é a manutenção de listas encadeadas para tarefas prontas e suspensas.

O gerenciamento da pilha é feito através de rotinas de salvamento (*save*) e restauração (*restore*) do contexto. A rotina *save* é chamada por um manuseador de interrupções para salvar o contexto corrente da máquina na área da pilha. Essa chamada deve ser feita logo após as interrupções terem sido desabilitadas para prevenir corrupção dos dados. A rotina *restore* deve ser executada logo antes da habilitação das interrupções e antes do retorno do manuseador de interrupções.

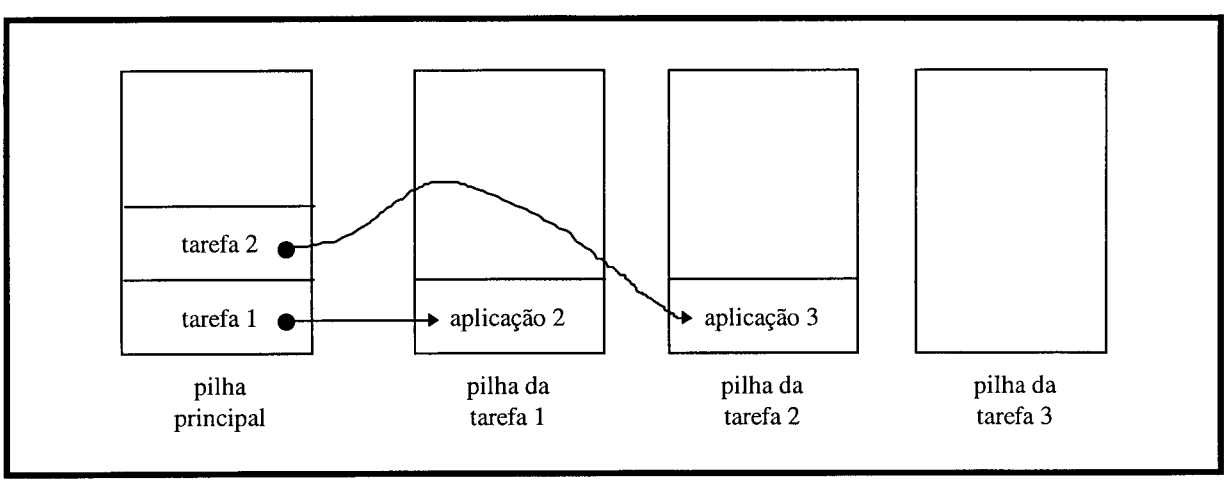

Figura 2.9 - Combinação de múltiplas pilhas de memória

A quantidade máxima de espaço necessário para a pilha em tempo de execução necessita ser conhecido *a priori*. Em geral, o tamanho da pilha pode ser determinado se não é usado recursão. Pode-se utilizar também um esquema de múltiplas pilhas (Figura 2.9) , onde

uma única pilha principal é pilha em tempo de execução e as demais são pilhas de aplicação. A utilização de múltiplas pilhas apresenta vantagens em sistemas embutidos de tempo real, tais como: tarefas podem interromper a elas próprias, permitindo, para manuseio transiente, condições de sobrecarga ou, para detecção, interrupções espúrias; o sistema pode ser escrito em uma linguagem que suporte recursão e código reentrante.

ALOCAÇÃO DINÂMICA

A alocação dinâmica é utilizada para satisfazer requisições individuais de tarefas por memória. Ela é garantida pelo uso de uma estrutura de dados do tipo lista ou árvore binária. Em geral, esse tipo de gerenciamento de memória não é adequado para os sistemas embutidos (*embebed systems*). Alguns esquemas que permitem a alocação dinâmica da memória são: a permuta ou *swapping*; a segmentação; a multiprogramação com um número fixo ou variável de tarefas; e a paginação por demanda.

- permuta de memória (*swapping*) é o esquema mais simples de alocação de memória para dois processos simultaneamente. Nesse caso, o sistema operacional está sempre residente na memória, e um processo pode "co-residir" na memória, no espaço não requerido pelo sistema operacional, chamado de espaço de usuário. Quando um segundo processo necessita executar, o primeiro processo é suspenso e movido para a memória secundária, juntamente com o seu contexto, para um dispositivo de armazenamento secundário, geralmente um disco. O segundo processo, juntamente com o seu contexto, é então carregado no espaço de usuário e inicializado pelo despachante (*dispatcher*). Esse tipo de esquema pode ser usado juntamente com Round-Robin ou sistemas prioritários preemptivos, sendo desejável, entretanto que o tempo de execução de cada processo fosse longo em relação ao tempo de permuta. O tempo de acesso a memória secundária é o principal fator para o *overhead* na comutação de contexto e para os atrasos de resposta em tempo real.
- *overlay* é a técnica que permite a um único programa ser maior que o espaço disponível na área de usuário. Nesse caso, o programa é dividido em códigos

dependentes e sessões de dados, os quais são encaixados na memória disponível. Deve-se incluir um código especial do programa para permitir que novas segmentações sejam permutadas na memória quando necessário (sobre segmentações já existentes). Tanto na segmentação quanto na permuta de memória uma parte da memória nunca é permutada para o disco ou é segmentada. Essa parte contém o gerenciador de segmentação ou permuta. As implicações negativas para tempo real ocorrem pelo fato dos segmentos terem de ser permutados de dispositivos de memória secundária, que são, geralmente, mais lentos e com tempos de acesso não deterministas.

- multiprogramação com um número fixo de tarefas (MFT) pode ser utilizado quando o número de tarefas é conhecido e fixo (sistemas embutidos, por exemplo). A permuta particionada para o disco pode ocorrer quando a tarefa é "preemptada". Entretanto, as tarefas devem residir em partições contíguas, e a alocação/desalocação dinâmica da memória pode causar problemas. Outro problema é a fragmentação interna, quando um processo requer espaço menor do que o tamanho fixado. O desperdício de memória e fragmentação interna podem ser reduzidos pela criação de partições fixas de tamanhos variados. Esse esquema não é aplicável em sistemas operacionais de tempo real porque ele usa a memória de forma ineficiente e gera um *overhead* no ajuste de um processo para a memória disponível e a permuta de disco.

- multiprogramação com número variável de tarefas (MVT) é utilizado quando o número de tarefas é desconhecido ou variável. O tamanho da memória alocada é determinado somente no momento em que um processo é carregado na memória. A fragmentação interna ocorre em menor escala, mas a fragmentação externa poderá ocorrer devido a natureza dinâmica de alocação/desalocação de memória e, pelo fato da memória ser alocada para um processo continuamente. Esse tipo de esquema gera um *overhead* considerável na comutação de contexto, não sendo apropriado para sistemas de tempo real embutidos. Sua utilização é freqüente em sistemas de tempo real comerciais.

- paginação por demanda permite o carregamento em memória não-contígua de segmentos de programa. Eles são demandados em pedaços de tamanho fixo

chamados "páginas". Esse esquema ajuda a eliminar fragmentação externa de memória. Quando uma referência de memória é feita para um local com uma página não carregada na memória principal, uma exceção de falta de página é gerada. O manuseador de interrupções para essa exceção verifica se existe uma página livre na memória. Caso não encontre, uma página deve ser selecionada e permutada para o disco. A paginação é mais eficiente quando suportada pelo *hardware* apropriado.

Alguns aspectos devem ser observados no gerenciamento de memória em sistemas de tempo real *hard*, pois, geralmente, não se pode dispor de *overhead* extra associado com esquemas de permuta de memória, segmentação ou paginação.

Um modo de reduzir o tempo de acesso a memória é se trabalhar com grupos de instruções. O exame de uma lista de instruções de programas, executados recentemente, em um analisador lógico, mostra que alguns grupos de instruções tem uma freqüência considerável. A idéia é manter esses grupos de instruções em *cache* e o acesso a localização de memória feito por grupo e não por instrução. A utilização de *cache*, no entanto, leva a um não determinismo, uma vez que no pior caso considera-se que a informação não está em *cache*.

Outro fator importante a ser tratado no gerenciamento de memória é quanto aos resíduos. Resíduo é memória que tenha sido alocada, mas era menor que a necessária pela tarefa. Os resíduos podem acumular devido a tarefas terminadas abruptamente sem liberação de recursos de memória.

2.3.4 Mecanismos de Entrada/Saída

As aplicações tempo real necessitam que os mecanismos de entrada/saída sejam determinísticos. Além disso, deve-se ter a garantia da atualização dos arquivos em disco, para que não sejam obtidos dados inválidos ou desatualizados.

Os mecanismos podem ser síncronos ou assíncronos, priorizados ou não. Outro problema no acesso a recursos é possibilidade de *deadlock*. Em geral, implementam-se sinais

de *timeout* para que seja indicada a falha na obtenção de um recurso depois de expirado o tempo estipulado.

Os mecanismos assíncronos muitas vezes necessitam ser sincronizados. Essa sincronização difere do mecanismo síncrono. Trata-se de uma garantia de que o mecanismo de entrada/saída tenha concluído uma operação subjacente antes de retornar ao dispositivo em questão. Um exemplo disso pode ser uma versão sincronizada de uma primitiva de gravação, que não retornará resposta ao usuário até que a operação tenha sido concluída no dispositivo (disco, fita, etc.) e este tenha sido atualizado.

Um outro fator a considerar em mecanismos de entrada/saída é a detecção e o tratamento de erros, quando, por exemplo, um dispositivo não completa a sua operação e tem que informar esse fato a outro dispositivo que esteja aguardando o término dessa operação.

Os mecanismos assíncronos podem ainda ter tratamento diferenciado de acordo com a prioridade associada, uma vez que eles se utilizam de filas e a ordem de enfileiramento pode ser alterada com base na prioridade de cada operação.

2.4 Padronização de Serviços de Tempo Real

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 ("*Portable Operating System Interface*" - Interface para Portabilidade de Sistemas Operacionais) é o conjunto de documentos produzido pela IEEE, e adotado pela ANSI e ISO.

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".

2.4.1 Conjuntos de Especificações

O padrão POSIX engloba diversos assuntos além da padronização para sistemas de tempo real, como: segurança; administração de sistemas. Os padrões mais relevantes para sistemas de tempo real são:

- POSIX.1: núcleo básico de interface entre sistemas operacionais. É necessário em qualquer sistema 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; manipulação de terminal.
- POSIX.1b[1] ou POSIX.4: acrescenta extensões para tempo real [Gallmeister 1995], tratando assuntos como: entrada/saída síncrona e assíncrona; bloqueio de memória; memória compartilhada; passagem de mensagem; semáforos; escalonamento (prioritário e round-robin); relógios de tempo real (com granularidade da ordem de nanosegundos).
- POSIX.1c ou POSIX.4a: incorpora extensões adicionais ao POSIX.1 e POSIX.1b, provendo a habilidade de executar múltiplos *threads* concorrentes [Nichols 1996] de um único processo. Ele trata de gerenciamento e escalonamento de *threads*; variáveis condicionais; sinais; escalonamento de processos.
- POSIX.1d ou POSIX.4b: acrescenta mais extensões adicionais [Laplante 1997]: criação de processos; funções de bloqueio e *timeout*; monitoração de tempo de

[1] Nova numeração do padrão POSIX.4 da IEEE, assim como também foram renomeados POSIX.4a, POSIX.4b para POSIX.1c e POSIX.1d, respectivamente.

execução; escalonamento de servidores esporádicos; 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.

2.4.2 Áreas Funcionais

Uma análise das áreas funcionais de acordo com os padrões POSIX.1b e POSIX.1c permite uma visão ampla de suas características. No POSIX.1b a única parte obrigatória do padrão são alguns mecanismos de sinais (sinais de tempo real e enfileiramento de sinais).

GERENCIAMENTO DE PROCESSOS

O POSIX.1b fornece primitivas (anexo A) para gerenciamento de processos, tais como:

- sinais: as sinalizações adicionais que o POSIX.1b acrescenta ao POSIX.1 são de mensagem numa fila vazia, sinais de expiração de relógio (*timeout*), de conclusão de entrada/saída assíncrona, e de exceção.
- escalonamento: são previstas três políticas de escalonamento. Uma prioritária do tipo FIFO, outra Round-Robin e outra definida pela aplicação.
- *clocks* e *timers*: *clock* é o dispositivo que contém o tempo (relógio) e *timers* são utilizados para emitir alarmes ou acionar algum evento específico, sendo geralmente baseados em *clocks* particulares [Gallmeister 1995]. Além do *clock* padrão, podem ser utilizados outros *clocks* específicos. Apresenta todos os relógios suportados pelo POSIX.1, com algumas diferenças: eles podem ser criados/deletados dinamicamente e permitem a inclusão de atrasos da ordem de nanosegundos.

THREADS

Thread é o modelo utilizado para dividir programas em tarefas, cuja execução pode ser intercalada ou paralela. *Pthreads*[2] são tratadas no POSIX.1c (anexo A), onde ele especifica gerenciamento e escalonamento de *threads*; variáveis condicionais; sinais; escalonamento de processos.

- gerenciamento de *threads*: as primitivas permitem o gerenciamento em dois níveis: um a nível de sistema (as *threads* são escalonadas em relação a todas as outras) e outro a nível de processo (as *threads* são escalonadas em relação as *threads* do mesmo processo).
- sincronização: é provida através do uso de variáveis condicionais, *mutexes* e bloqueio do tipo "leitor/escritor"[3].

COMUNICAÇÃO ENTRE PROCESSOS / SINCRONIZAÇÃO

A sincronização de múltiplos processos no POSIX.1b é resolvida através de semáforos contadores, enquanto o POSIX.1c possibilita a utilização de variáveis condicionais.

O POSIX.1 provê facilidades como sinais e dutos (*pipes*) para comunicação entre processos. O POSIX.1b acrescenta outros, como:

- passagem de mensagens: a troca de mensagens é realizada através de mecanismos de filas de mensagens, que são implementadas sem utilizarem chamadas ao sistema. A filosofia das filas de mensagens é semelhante aos arquivos do UNIX - nomenclatura, criação, acesso. O envio e recebimento das mensagens segue a prioridade da fila. A dificuldade no tratamento de filas aparece quando a aplicação não é baseada em fila. Outra desvantagem é a perda

---

[2] *PThreads* é o padrão POSIX para *threads* O "P" vem de POSIX.

[3] Tipo de bloqueio, onde o processo que está gravando (ou escrevendo) o dado tem a garantia de que é o único a estar acessando aquela área de memória, ou seja, não haverá ninguém "lendo". E quando um ou mais processos estão "lendo" uma área de memória os outros ficam impedidos de gravar.

de eficiência, pelo fato da fila de mensagens copiar dados no sistema operacional e de lá para o destino, gerando *overhead.*

- semáforos: os semáforos contadores controlam o acesso a um recurso, ou deixam o processo aguardando algum evento.
- memória compartilhada: a utilização de compartilhamento de memória implica no mapeamento e na necessidade de gerenciamento.

GERENCIAMENTO DE MEMÓRIA

O gerenciamento é provido pelo estabelecimento de proteção de memória, ou através do bloqueio de uma faixa de memória ou de toda a memória (para evitar *swapping* ou paginamento)

MECANISMOS DE ENTRADA/SAÍDA

O POSIX.1b possui primitivas que possibilitam mecanismos de entrada/saída síncrona e assíncrona. Permite também que entrada/saída assíncronas sejam priorizadas, modificando a ordem da fila de E/S. Garante também que o acesso aos dados seja feito a dados atualizados.

2.5 Sumário

Este capítulo trouxe uma breve descrição sobre SOTR, apresentando seus componentes, a padronização (POSIX) proposta pela IEEE e alguns comparativos com os sistemas convencionais. Nos capítulos seguintes são abordados o ambiente de aplicação da máquina CRUX, o conjunto básico de primitivas de suporte para tempo real para esse ambiente e a análise do desempenho dos mecanismos de comunicação.

3. O Ambiente CRUX

3.1 Introdução

O ambiente CRUX representa um completo ambiente para programação paralela, sendo composto pelo Multicomputador CRUX e pelo Sistema Operacional CRUX, desenvolvidos no Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina pelo grupo de pesquisa em Computação Paralela e Distribuída.

3.2 A Arquitetura CRUX

O multicomputador CRUX (Figura 3.1) 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 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*.

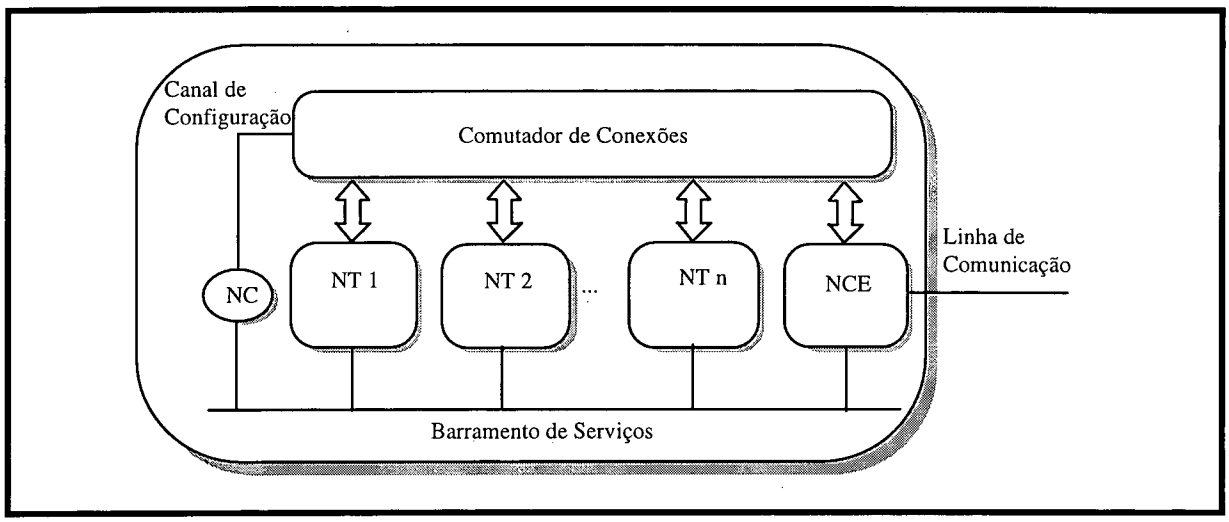

Figura 3.1 - A arquitetura do multicomputador CRUX

3.2.1 Os Nós de Trabalho (NT)

Cada nó de trabalho (Figura 3.2) dessa arquitetura possui um processador i486, memória RAM privativa de 4 *Mbytes*, uma pequena memória ROM com o seu respectivo microcódigo, interface com o comutador de conexões e interface com o barramento de serviço.

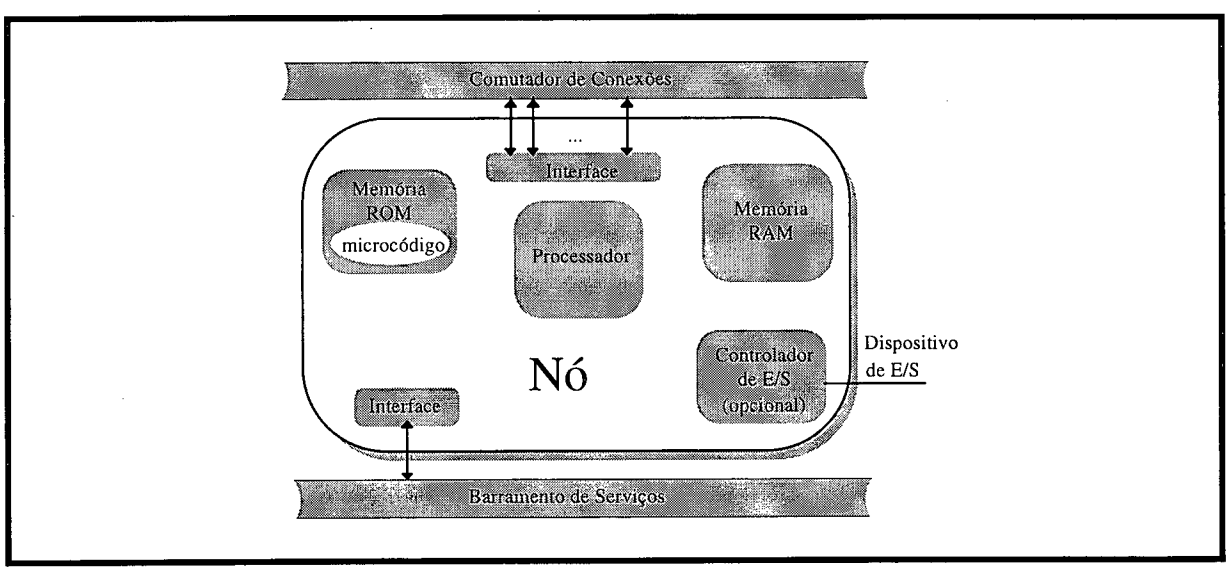

Figura 3.2 - A estrutura interna de um nó

3.2.2 O Nó de Controle (NC)

O nó de controle é responsável pelo controle do comutador de conexões (*crossbar*), através de um canal de configuração (Figura 3.2). Todos os demais nós podem requisitar conexões bipontuais entre si se comunicando com o NC através do barramento de serviços, que funciona como via de comunicação entre um nó qualquer e esse dispositivo.

De forma semelhante a um NT, o NC é constituído de processador, memória RAM e memória ROM com o seu microcódigo. Entretanto, a capacidade de comandar o comutador de conexões o distingue dos demais nós. Devido a essa importante atribuição, ele é dedicado exclusivamente às tarefas relacionadas com essas funções, não possuindo outras conexões com o *crossbar* e, conseqüentemente, não se comunicando com outros nós através desse dispositivo.

O código que executa em seu processador é armazenado em ROM e os dados em RAM; o código é compacto e otimizado para atender às mensagens com requisições de nós e conexões. Ele gerencia diversas filas mantidas para o controle dos pedidos de alocação e liberação de nós pelos processos, e de conexão e desconexão de canais de comunicação entre os nós.

3.2.3 O Nó de Comunicação Externa (NCE)

O nó de comunicação externa tem o objetivo de realizar a comunicação com o "mundo externo" através de uma linha de comunicação. É possível utilizar máquinas desse tipo como nós independentes em redes locais, na forma de um *pool* de processadores. Assim, os processadores podem ser alocados por demanda, segundo a necessidade dinâmica dos processos dos programas paralelos.

3.2.4 O Comutador de Conexões

O comutador de conexões é do tipo *crossbar*. Ele estabelece uma rede dinâmica de interconexão entre os nós processadores. Dessa forma, cada nó pode ser conectado a qualquer outro dinamicamente através de canais de comunicação bidirecionais.

Esses canais de comunicação permitem a transmissão eficiente de mensagens volumosas. Processos colocados em nós distintos precisam requisitar conexões entre si ao comutador de conexões para trocar mensagens através desses canais.

3.2.5 O Barramento de Serviços (BS)

Principalmente pelo fato de ser uma via compartilhada por todos os nós, o barramento de serviços é concebido para ser utilizado apenas para troca de mensagens curtas e pré-estabelecidas entre um NT qualquer e o NC. Colisões no acesso ao barramento de serviços não acontecem porque é o NC que determina com quem vai ser realizada a comunicação, verificando, um a um, quais os nós que desejam se comunicar com ele.

As mensagens que trafegam por esse barramento (Tabela 3.1) e que são recebidas e tratadas pelo NC são as seguintes:

Tabela 3.1 - Mensagens no barramento de serviço

MENSAGEM	PARÂMETRO	RETORNO
connect	*nid*	nada
connect_any	nenhum	*nid*
disconnect	*nid*	nada
allocate	*nid*	nada
allocate_any	nenhum	*nid*
deallocate	*nid*	nada

- *connect*: requisição ao NC para que ele estabeleça uma conexão bipontual entre o nó do processo que pediu a conexão e o nó *nid*. A conexão só é realmente estabelecida quando o processo no nó *nid* também pede uma conexão e, enquanto

isso não ocorre, a requisição permanece numa fila de requisições de conexões geridas pelo NC. Somente quando a conexão é realmente efetivada é retornado uma mensagem de confirmação para ao processo que requisitou a conexão.

- *connect_any*: semelhante à mensagem anterior, com a diferença que é solicitada uma conexão com qualquer nó que deseja se comunicar com o nó onde se encontra o processo que pediu a conexão. Quando a conexão é efetuada, é retornado o identificador do nó com o qual foi efetuada a conexão.

- *disconnect*: é um pedido ao NC para que ele desfaça a conexão entre o nó do processo que pediu a desconexão e o nó *nid* especificado. A conexão física do canal de comunicação entre os nós é imediatamente desfeita e o processo que executou a chamada fica liberado para requisitar outras conexões. O processo que executa no outro nó conectado precisa, também, posteriormente, pedir a desconexão, que nesse caso é apenas lógica, já que a conexão física já foi desfeita.

- *allocate*: é um pedido para o NC reservar o nó *nid* para o processo que enviou a mensagem. Após reservar o nó, o NC estabelece, uma conexão entre o nó reservado e o nó onde se encontra o processo que pediu a alocação. Se o nó requisitado já está alocado, essa chamada retorna uma mensagem de erro.

- *allocate_any*: semelhante à mensagem anterior, com a exceção que nela não se especifica qual nó se deseja alocar. O NC reserva um nó qualquer segundo uma fila interna de nós disponíveis que ele possui, e retorna o identificador do nó para o processo que pediu a alocação. Se não há nós disponíveis para alocar, essa chamada retorna uma mensagem de erro. De forma semelhante à mensagem anterior, uma conexão entre os nós é estabelecida.

- *deallocate*: pedido ao NC para que ele libere o nó onde executa o processo que enviou a mensagem.

3.3 O 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.

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, descrito em [Montez 1995] e [Campos 1995], foi implementado para executar na arquitetura baseada em barramento comum. Seguindo a tendência dos sistemas operacionais modernos (*Amoeba*, *Mach*, *Chorus*), o CRUX é formado por duas peças principais: um micronúcleo e um conjunto de processos servidores (Figura 3.3).

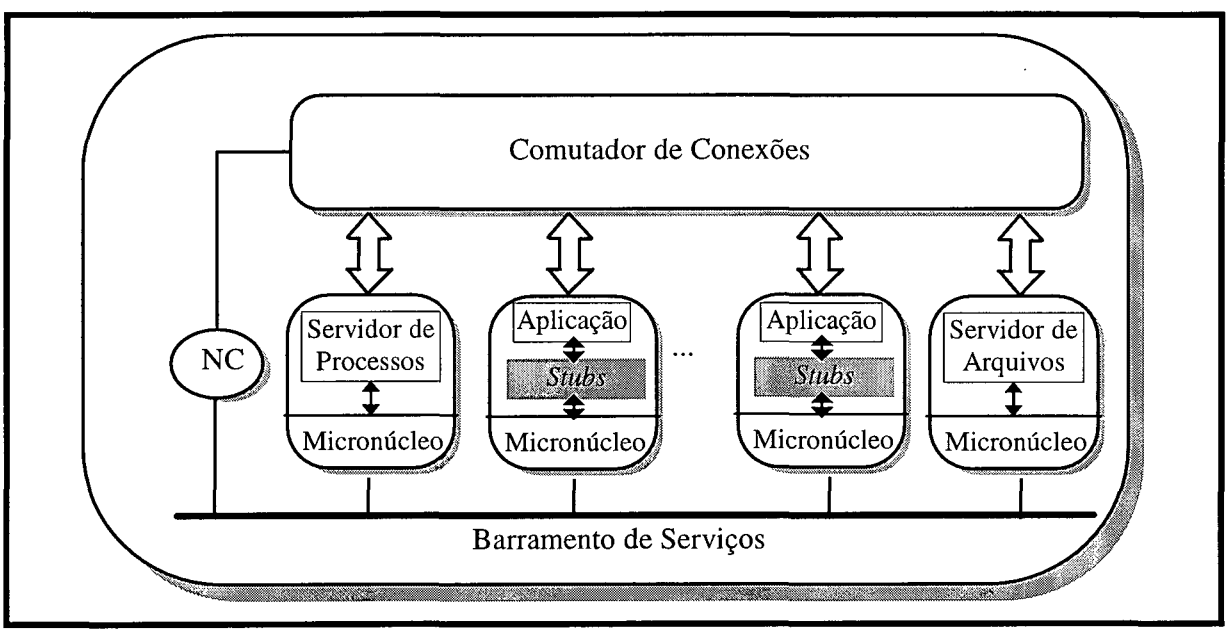

Figura 3.3 - A arquitetura do sistema CRUX

A Comunicação entre os processos no CRUX é efetuada através de canais bipontuais, com endereçamento direto, de maneira síncrona [Corso 1993]. A partir desse modelo simples,

pretende-se construir mecanismos mais aprimorados, como caixas postais ou comunicação assíncrona.

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.

3.3.1 Processos CRUX

Os processos CRUX (Figura 3.4) são programas que estão em execução em cada um dos nós de trabalho. Um processo colocado na memória de um nó é constituído pelo seu espaço de endereçamento, dados, pilha, valores dos registradores e outras informações necessárias para sua execução [Montez 1995].

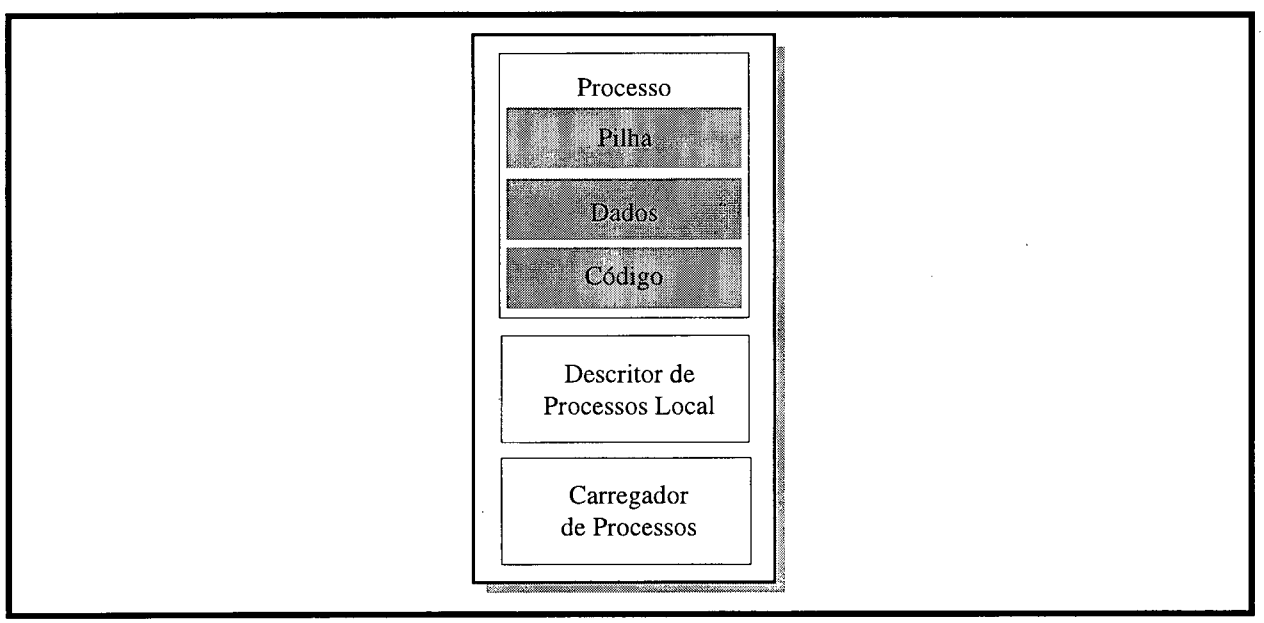

Figura 3.4 - Processo CRUX

Processos CRUX são criados através da chamada *fork*, que é uma chamada de sistema compatível com UNIX. Neste caso, o processo criado é enviado para um nó disponível que o

recebe e o "instala". O código responsável por receber e instalar o processo e seu descritor na memória é o carregador de processos.

O sistema operacional possui uma estrutura na memória de cada nó própria para armazenar o contexto do processo. Essa estrutura, denominada descritor de processos local, armazena localmente diversas informações que servem para descrever o processo para o sistema operacional.

Um processo colocado na memória é composto de seu código, sua área de dados e pilha. A área de código consiste em uma seqüência de padrões de *bytes* que o processador interpreta como instruções de máquina. A área de dados corresponde à seção de dados inicializados e não inicializados existentes no arquivo executável. A área de pilha é criada e pode ser alterada dinamicamente durante a execução do programa.

Entre a área de dados e a área de pilha (Figura 3.3), existe um espaço livre, que é utilizado para comportar o crescimento da pilha e para a implementação da chamada de alocação dinâmica de memória do sistema UNIX (*brk*), como será descrito mais adiante (3.3.4) quando for abordado a gerência de memória.

3.3.2 Camadas do Sistema CRUX

A visão lógica do sistema CRUX é apresentada na Figura 3.5. Ele pode ser visto como um conjunto de camadas, onde cada uma delas utiliza serviços da camada imediatamente inferior para oferecer serviços a camada imediatamente superior.

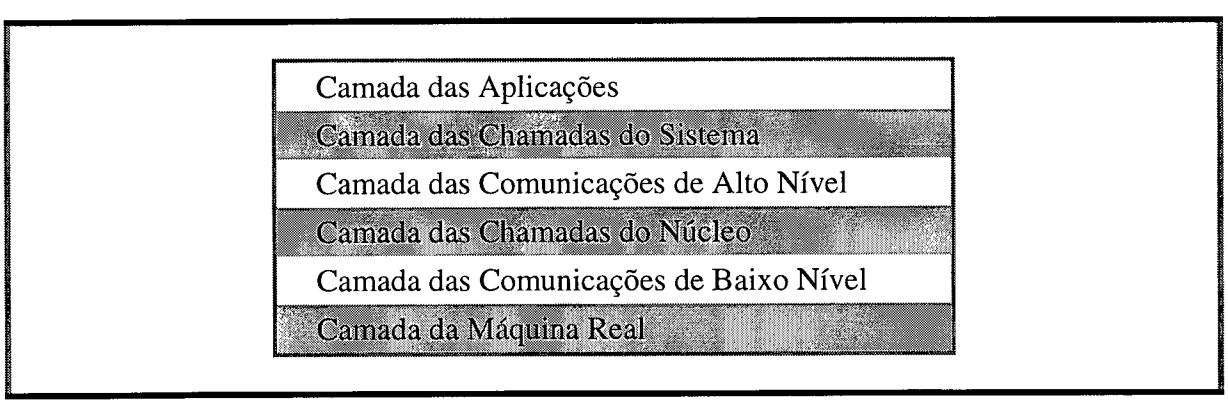

Figura 3.5 - Camadas do sistema CRUX

A camada de chamada do sistema fornece serviços equivalentes aos serviços fornecidos pelo sistema UNIX, acrescidos de serviços de comunicação do modelo das redes de processos comunicantes. A camada das comunicações de alto nível oferece serviços para o transporte de mensagens volumosas entre os nós de trabalho através da rede de comunicação. A camada das chamadas de núcleo fornece os serviços para o estabelecimento e o cancelamento de conexões no *crossbar*, além de alocação e da liberação de nós de trabalho. A camada das comunicações de baixo nível fornece serviços para a troca de mensagens de serviços entre os nós de trabalho e o nó de controle. A camada das aplicações refere-se a aplicações desenvolvidas sobre o sistema e a da máquina real refere-se aos recursos materiais do multicomputador CRUX.

3.3.3 Micronúcleo CRUX

O sistema operacional CRUX é fundamentado sobre um micronúcleo genérico que provê serviços para comunicação entre processadores e alocação de processadores. As primitivas de comunicação entre processadores são síncronas, com endereçamento direto, permitindo envio e recepção de mensagens de tamanhos variáveis. Cada nó processador da máquina tem um micronúcleo em uma memória ROM [Silva 1996].

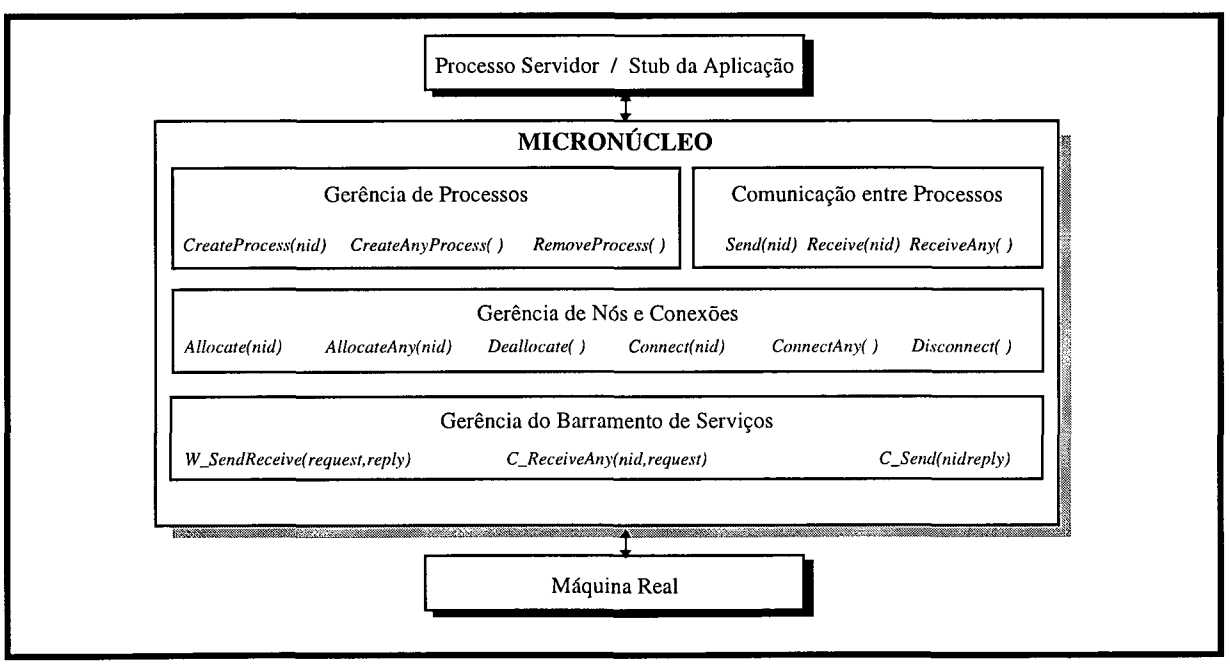

Figura 3.6 - Micronúcleo CRUX

O micronúcleo CRUX foi estruturado em um esquema de três camadas, onde as camadas inferiores fornecem serviços às camadas superiores (Figura 3.6).

- Gerência do Barramento de Serviços: o objetivo desta camada é prover serviços de comunicações, que permitem trocas de mensagens compactas, para as chamadas de procedimento remoto através do barramento de serviço (Tabela 3.2).

Tabela 3.2 - Funções da camada de gerência do barramento de serviço

FUNÇÃO	DESCRIÇÃO
W_SendReceive(request,reply)	NT envia mensagem ao NC e aguarda resposta.
C_ReceiveAny(nid,request)	NC aguarda uma mensagem.
C_Send(nid,reply)	NC envia ao NT *nid* uma mensagem.

- Gerência de Nós e Conexões: esta camada oferece serviços de conexão e alocação de nós, utilizando as funções da camada inferior. As funções dessa camada são listadas na Tabela 3.3.

Tabela 3.3 - Funções da camada de gerência dos nós e das conexões

FUNÇÃO	DESCRIÇÃO
Allocate(nid)	Aloca o nó com endereço *nid.*
AllocateAny(nid)	Aloca um nó qualquer.
Deallocate( )	Desaloca o nó solicitante.
Connect(nid)	Pede conexão com o nó *nid.*
ConnectAny( )	Pede conexão com um nó qualquer.
Disconnect( )	Pede desconexão do nó conectado.

- Gerência e Comunicação de Processos: esta camada oferece serviços aos processos externos ao micronúcleo. Na seção de gerência de processos existem três funções para criar e remover processos (Tabela 3.4). As interações entre os processos existentes no sistema seguem, na grande maioria das vezes, a disciplina de comunicação cliente-servidor.

Tabela 3.4 - Funções da camada de gerência e comunicação de processos

FUNÇÃO	DESCRIÇÃO
CreateProcess(nid)	Cria um processo no nó *nid.*
CreateAnyProcess( )	Cria um processo num nó qualquer.
RemoveProcess( )	Remove o processo do nó corrente.
Send(nid)	Envia uma mensagem para o nó *nid.*
Receive(nid)	Aguarda uma mensagem do nó *nid.*
ReceiveAny( )	Aguarda uma mensagem de um nó qualquer.

As funções *CreateProcess* e *CreateAnyProcess* são responsáveis por criar um contexto de execução para um processo. No CRUX, tal como no UNIX, processos são criados através da chamada de sistema *fork.* A implementação dessa chamada implica na chamada de *CreateAnyProcess* para que o micronúcleo execute as tarefas de "baixo-nível" na criação do contexto para o novo processo. Esse processo é colocado em um nó selecionado através da função *AllocateAny* que *CreateAnyProcess* chama.

A função *CreateProcess* é utilizada em casos onde é necessário a carga de um processo em um nó específico. Sua implementação implica numa chamada da função *Allocate* para alocar o nó especificado. Por exemplo, um servidor pode precisar ser carregado sempre em um determinado nó. Dessa forma, toda a chamada de procedimento remoto ao servidor é endereçada a aquele nó pelos processos clientes.

No UNIX, um processo termina ao executar a chamada *exit*. Neste caso, a função *RemoveProcess* do micronúcleo é chamada para remover o processo e seu contexto. Pela característica de haver apenas um processo por nó e pelo fato da chamada *exit* se referir ao próprio processo que termina, não é necessário passar nenhum tipo de identificador do processo a ser removido.

3.3.4 Gerência de Memória

O gerente de memória do CRUX consiste de uma peça simples, que é implementada na camada de *stubs* através da chamada *brk*, permitindo a alteração do tamanho da área de dados do processo "chamador" [Campos 1995].

A chamada *brk* recebe como parâmetro um endereço, que será o novo marcador de final da área de dados. Desse modo, é possível reduzir a área de dados, fornecendo um endereço menor que o marcador atual, ou aumentar a área de dados, fornecendo um endereço maior que o atual.

A implementação da chamada *brk* consiste da simples modificação de uma variável localizada na camada de *stubs*, que mantém o endereço final da área de dados do processo.

3.3.5 Biblioteca CRUX

Esta biblioteca possui funções que podem ser ligadas com os programas de usuário. Através desta biblioteca os processos acessam o conjunto das chamadas de sistema CRUX. A maioria das chamadas são atendidas pelo servidor CRUX. Essas chamadas envolvem uma comunicação cliente-servidor utilizando serviços oferecidos pela camada de comunicação entre processos existentes no micronúcleo.

3.3.6 Servidor CRUX

O servidor CRUX é o processo do sistema, responsável por gerenciar as funções de processos UNIX e funções de sistema de arquivos UNIX. As funções de processos UNIX

consistem em requisições relacionadas a gerenciamento de processos (*fork, exit, wait* e *exec*), para os quais o servidor CRUX mantém uma tabela de processos, onde são armazenadas as informações dos processos existentes no sistema. As funções de sistema de arquivos (*open, close, lseek, read* e *write*) também são tratadas pelo servidor CRUX. Cada processo pode manter vários arquivos abertos, cujos descritores são armazenados na tabela de processos mantida pelo servidor CRUX.

3.4 Sumário

Depois da apresentação, neste capítulo, da visão do ambiente de aplicação CRUX, composto pelo Multicomputador e pelo Sistema Operacional, desenvolvidos no Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina pelo grupo de pesquisa em Computação Paralela e Distribuída, os próximos capítulos tratam do suporte para tempo real para esse ambiente e da análise de desempenho dos mecanismos de comunicação.

4. Suporte a Tempo Real

4.1 Introdução

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 ainda não existe um suporte a aplicações de tempo real. Dessa forma, pretende-se propor um conjunto mínimo de primitivas de tempo real, baseadas no padrão POSIX[4] da IEEE.

Considerando-se, entretanto, a grande abrangência desse padrão e as características peculiares do ambiente e de aplicações de tempo real, optou-se pela utilização de um número mínimo de primitivas que possibilitassem atender as necessidades de suporte de tempo real para a máquina CRUX.

Este capítulo apresenta as primitivas de suporte a tempo real para o ambiente CRUX, dentro das áreas funcionais de gerência/escalonamento de processos e *threads*, gerência de memória e comunicação entre processos/*threads*.

4.2 Processos e *Threads*

Tendo em vista que na máquina CRUX temos 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 [Montez 1995] [Campos 1995].

A proposta, aqui apresentada, considera um modelo de processos *multithread*, ou seja, cada processo do ambiente CRUX, 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

---

[4] Aqui, trata-se especificamente dos conjuntos de padrões POSIX.1b e POSIX.1c

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 uma pilha (Figura 4.1). 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. Para criar um novo processo, o CRUX utiliza a chamada *fork*. Na criação de *threads* para esse processo, utiliza-se a primitiva *pthread_create* (vista em detalhes na seção 4.2.1).

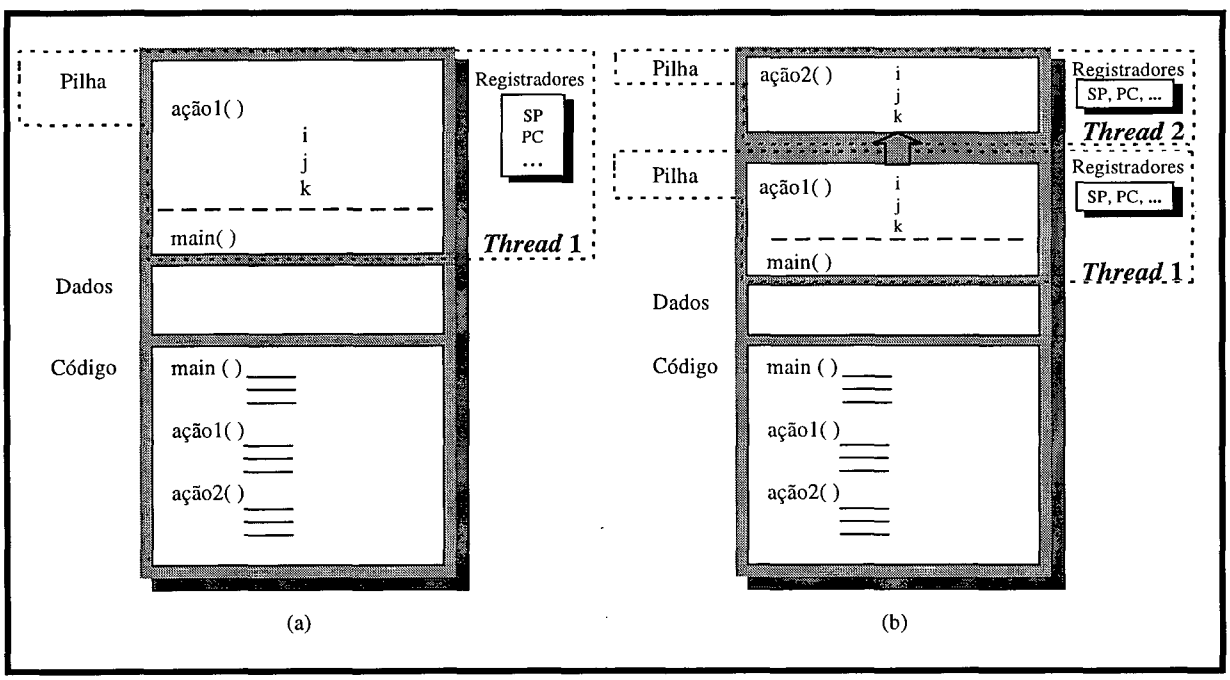

Figura 4.1 - Processo com uma *thread* (a) e com *multithreads* (b)

4.2.1 Primitivas de Criação e Gerenciamento de *Threads*

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 4.1), 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.

TABELA DE *THREADS*

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 4.1 - 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 mínimo para se inicializar a *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 mínimo para ser inicializada e o seu tempo de chegada.

PRIMITIVAS

Na criação e gerenciamento de *threads*, utiliza-se as primitivas propostas no padrão POSIX.1c [Nichols 1996], conforme mostrado na Tabela 4.2.

Na criação de *threads*, a primitiva *pthread_create*, recebe como parâmetro o tamanho da pilha da *thread*, localiza uma entrada disponível na tabela de *threads*, armazena o tamanho da pilha no item *thread_stack_tam*, retornando o valor de identificação da *thread* (*thread_id*). Os parâmetros referentes aos atributos da *thread* assumem os valores do processo quando não forem definidos. Os atributos da *thread* utilizados para o escalonamento são: sua prioridade (*thread_prio*), seu *deadline* (*thread_deadline*), seu tempo mínimo para inicialização (*thread_start_time*) e seu tempo de chegada (*thread_arrival_time*). Com exceção do último atributo, os demais podem ser estabelecidos no momento da criação da *thread*, através da primitiva *pthread_create*, ou alterados posteriormente com a primitiva *pthread_setschedparam*.

Tabela 4.2 - Primitivas de criação e gerenciamento de *threads*

PRIMITIVA	PARÂMETRO	VALOR RETORNADO	FUNÇÃO
pthread_create ( )	tamanho da pilha, política de escalonamento, atributos	identificação da *thread*	cria uma *thread* com o tamanho de pilha de *thread*, com as prioridades e a política de escalonamento especificados pelo usuário.
pthread_cancel ( )	identificação da *thread*	____	cancela uma determinada *thread*
pthread_join ( )	identificação da *thread*	valor	aguarda o término de uma outra *thread* ("filha").
pthread_exit ( )	____	valor	termina execução da *thread* ("filha") e retorna à *thread* que chamou pthread_join
pthread_detach ( )	identificação da *thread*	____	faz com que uma *thread* deixe de aguardar o término de outra.

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. A primitiva *pthread_cancel*, cancela a *thread* indicada como parâmetro. Esta primitiva não permite atribuir nenhum estado de cancelamento, nem tipo de cancelamento. Para se lançar mão de restrições ao cancelamento de uma *thread* é necessário utilizar outras primitivas: *pthread_setcancelstate* e *pthread_setcanceltype*. Com estas primitivas é possível se definir o estado de cancelamento de uma *thread* (desabilitado ou habilitado) e neste último caso, se o cancelamento vai ocorrer em qualquer ponto (modo assíncrono) ou em pontos determinados (automaticamente ou pelo usuário). Os pontos de cancelamento automáticos são aqueles onde podem ocorrer bloqueios da *thread* executante (*pthread_join*, *pthread_cond_wait*, dentre outros). Os pontos de cancelamento definidos pelo usuário são aqueles onde ele faz uso da primitiva *pthread_testcancel*. No intuito de manter o suporte para um número mínimo de primitivas optou-se por incluir apenas a primitiva *pthread_cancel*, deixando as demais primitivas (*pthread_setcancelstate*, *pthread_setcanceltype*, *pthread_testcancel*) a critério da aplicação.

As primitivas *pthread_join* e *pthread_detach* permitem, respectivamente, que a *thread* que as executou, aguarde, ou deixe de aguardar, o término de uma outra *thread.* Uma *thread* pode desistir de aguardar o término de uma outra por esta estar bloqueada, por não ter terminado sua execução normalmente, ou por outros fatores. Essas duas primitivas necessitam, como parâmetro, apenas a identificação da *thread* que deve ser aguardada / "abandonada". A primitiva *pthread_join* recebe como retorno um valor específico, que é passado pela *thread* aguardada, quando esta executa a primitiva *pthread_exit.*

4.2.2 Primitivas de Escalonamento de *Threads*

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.

Tabela 4.3 - Primitiva de escalonamento de *threads*

PRIMITIVA	PARÂMETROS	VALOR RETORNADO	FUNÇÃO
pthread_setschedparam ( )	identificação da *thread,* política de escalonamento, atributos.	____	atribui os valores indicados, como parâmetros, nos atributos, para a *thread* identificada, de acordo com a política de escalonamento indicada.

Com o objetivo de manter a flexibilidade optou-se pela utilização de um escalonador mínimo, sobre o qual podem ser desenvolvidos mecanismos de escalonamento mais complexos, do tipo: Taxa Monotônica (*Rate Monotonic*), Próximo *Deadline* Primeiro (*Earliest Deadline First*) [Liu 1973], *Deadline* Monotônico (*Deadline Monotonic*) [Leung 1982], dentre outros. Essa flexibilidade, permite, entre outras coisas, uma maior portabilidade e adaptabilidade.

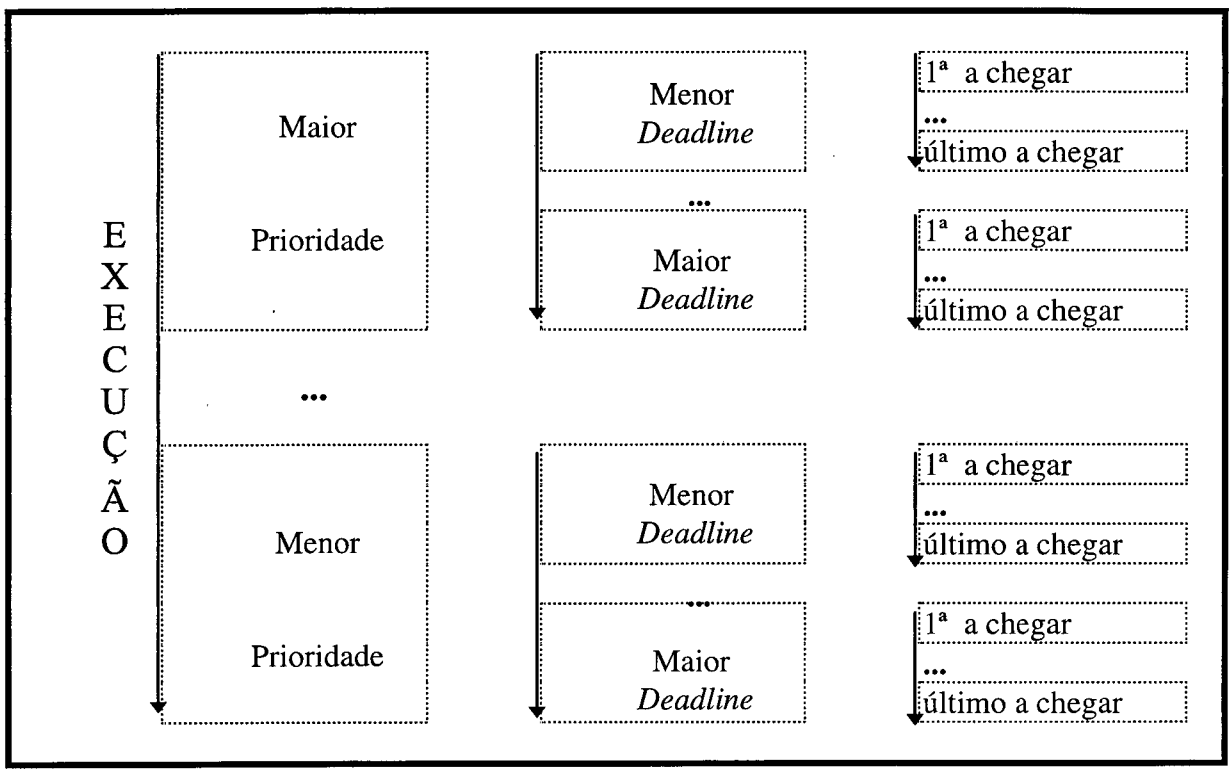

Figura 4.2 - Esquema de escalonamento hierárquico

Os atributos da *thread* utilizados para o escalonamento (prioridade, *deadline*, tempo mínimo para inicialização e tempo de chegada) com exceção deste último, podem ser estabelecidos no momento da criação da *thread*, através da primitiva *pthread_create* (Tabela 4.2), ou em um momento posterior com a primitiva *pthread_setschedparam* (Tabela 4.3). 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 4.2) segue modelo semelhante ao apresentado por [Lo 1997], 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.

4.2.3 Primitivas de 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).

A Tabela 4.4 apresenta as primitivas que permitem realizar a sincronização das *threads* e garantir o acesso exclusivo a determinado recurso.

As primitivas *pthread_mutex_init* e *pthread_mutex_destroy*, respectivamente, criam e deletam um *mutex* (também chamado, em outras nomenclaturas, de semáforo).

Para efetuar o bloqueio de um *mutex* utiliza-se a primitiva *pthread_mutex_lock*. Caso este *mutex* já esteja bloqueado, a *thread* que efetuou a chamada fica bloqueada e é colocada numa fila aguardando a liberação do *mutex*. Quando se deseja efetuar o bloqueio de um *mutex*, sem permitir o bloqueio da *thread* que executou a chamada (*mutex* já bloqueado) pode-se utilizar a primitiva *pthread_mutex_trylock*, que retorna um erro quando o *mutex* já está bloqueado (sem bloquear a *thread*) e bloqueia o *mutex* quando este está liberado. Na liberação do *mutex*, a primitiva *pthread_mutex_unlock* desbloqueia o *mutex*, reassumindo a *thread* de maior prioridade, dentre as que estavam aguardando a liberação.

A variável condicional difere do *mutex* na maneira de tratar a liberação da variável. Quando um *mutex* é liberado (*pthread_mutex_unlock*), 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.

Tabela 4.4 - Primitivas de sincronização de *threads*

PRIMITIVA	PARÂMETRO	VALOR RETORNADO	FUNÇÃO
pthread_mutex_init ( )	identificação do mutex	_____	cria um mutex.
pthread_mutex_destroy ( )	identificação do mutex	_____	deleta um mutex.
pthread_mutex_lock ( )	identificação do mutex	_____	bloqueia um mutex não bloqueado, ou a *thread* caso o mutex já esteja bloqueado.
pthread_mutex_trylock ( )	identificação do mutex	_____	bloqueia um mutex não bloqueado, retornando sem bloquear a *thread* caso o mutex já esteja bloqueado.
pthread_mutex_unlock ( )	identificação do mutex	_____	desbloqueia um mutex.
pthread_cond_init ( )	identificação da variável condicional	_____	cria variável condicional.
pthread_cond_destroy ( )	identificação da variável condicional	_____	deleta variável condicional.
pthread_cond_signal ( )	identificação da variável condicional	_____	libera variável condicional.
pthread_cond_wait ( )	identificação da variável condicional e identificação do mutex	_____	libera o mutex especificado, colocando a *thread* em estado de espera até a liberação da variável condicional.

As primitivas que tratam de variáveis condicionais são semelhantes as que tratam de *mutex* (Tabela 4.4).

As primitivas *pthread_cond_init* e *pthread_cond_destroy* criam e deletam uma variável condicional, respectivamente.

Para liberar uma variável condicional é utilizada *pthread_cond_signal*, que libera a *thread* de mais alta prioridade dentre aquelas que estão bloqueadas. Quando não existir nenhuma *thread* bloqueada, a primitiva não surtirá nenhum efeito.

A primitiva *pthread_cond_wait* desbloqueia, automaticamente, o *mutex* passado como parâmetro, e coloca a *thread* em estado de espera pela variável condicional. Quando essa variável é liberada, essa primitiva bloqueia o *mutex* e libera a *thread* para continuar sua execução.

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.

4.3 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 variáveis condicionais.

4.4 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 da Tabela 4.5 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, associando esses mecanismos a uma determinada fila de mensagem.

Tabela 4.5 - Primitivas para comunicação entre processos / *threads*

PRIMITIVA	PARÂMETRO	VALOR RETORNADO	FUNÇÃO
mq_open ( )	identificação da fila e *flags**	valor	cria ou acessa uma fila de mensagens
mq_close ( )	identificação da fila de mensagens	0 = sucesso -1 = falha	deleta uma fila de mensagens
mq_send ( )	identificação da fila, mensagem e prioridade.	0 = sucesso -1 = falha	envia mensagem para uma fila de mensagens
mq_receive ( )	identificação da fila.	mensagem, prioridade. -1 = falha	recebe mensagem de uma fila de mensagens

* Possíveis *flags*, atribuídos na criação ou acesso de uma fila de mensagens são:
- ⇨ *rdonly*, *wronly* ou *rdiwr*: somente escrita, somente leitura ou escrita/leitura;
- ⇨ *nonblock*: indica se a(s) primitiva(s) *mq_send* e *mq_receive* serão não-bloqueantes;
- ⇨ *creat*, *excl*: indica se a fila de mensagem está sendo criada ou está sendo acessada.

Através da primitiva *mq_open* pode-se criar ou acessar uma determinada fila de mensagens. Para tanto, é necessário indicar como parâmetros o identificador da fila, e o valor das *flags*, 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 (*creat*, *excl*).

Quando se deseja fechar uma fila utiliza-se *mq_close*, passando como parâmetro 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 novamente se abrir (*mq_open*) a menos que sejam abertas antes por outro processo ou *thread*.

No envio das mensagens para uma fila deve-se passar, como parâmetros para *mq_send*, 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 em *mq_open*.

O recebimento de mensagens que estejam armazenadas em uma fila é feito através de *mq_receive*, que necessita apenas do nome da fila como parâmetro, retornando a mensagem e o valor da prioridade associada a esta. Assim como em *mq_send* ela pode ser bloqueante ou não, dependendo do que foi definido em *mq_open*.

4.5 Sumário

Este capítulo apresentou as primitivas de suporte a tempo real para o ambiente CRUX, dentro das áreas funcionais de gerência/escalonamento de processos e *threads*, gerência de memória e comunicação entre processos/*threads*. Esse conjunto mínimo de primitivas para suporte a aplicações de tempo real no ambiente CRUX foi baseado no padrão POSIX[5] da IEEE.

O próximo capítulo apresenta a análise de desempenho dos mecanismos de comunicação, realizada através da simulação de seus parâmetros e de sua previsibilidade.

---

[5] Aqui, trata-se especificamente dos conjuntos de padrões POSIX.1b e POSIX.1c

5. Avaliação de Desempenho do Mecanismo de Comunicação CRUX

5.1 Introdução

Avaliação de desempenho pode ser realizada através de técnicas, tais como: modelagem analítica, medição efetiva do sistema ou simulação. A escolha da técnica adequada depende do tempo, dos recursos disponíveis para resolver o problema e do nível desejável de exatidão [Jain 1991].

Alguns pontos relevantes, que também devem ser considerados são:

- a modelagem analítica não deverá envolver sistemas muito complexos devido à complexidade dos cálculos.
- a medição efetiva do sistema necessita do sistema em funcionamento para a coleta dos dados.
- a simulação é uma ferramenta poderosa, mas precisa de uma máquina com alta capacidade de processamento e *software* adequados.

A escolha de uma dessas ferramentas está também relacionada ao tipo de problema a ser tratado e aos resultados que se deseja obter. No caso da avaliação de desempenho do mecanismo de comunicação da máquina CRUX para aplicações de tempo real, optou-se pela simulação. Essa escolha deveu-se, principalmente, ao fato da simulação poder ser realizada sem a máquina real. Isso é desejável para se fazer avanços na área de sistemas operacionais independente da completa finalização da arquitetura da máquina.

5.2 Simulação

Simulação é o processo de projetar o modelo de um sistema real, e com esse modelo realizar experimentos objetivando a compreensão do seu comportamento e avaliação das estratégias para o funcionamento do sistema. A simulação deve abranger a construção do modelo e a sua experimentação, tratando-se de uma metodologia experimental e aplicada que procura descrever o comportamento do sistema, construir teorias ou hipóteses que descrevam o comportamento observado e finalmente utilizar o modelo para fazer previsões futuras, para verificar os efeitos produzidos por trocas no sistema ou no seu método de operação.

A simulação é uma das mais poderosas ferramentas de análise disponíveis para o projeto e operação de sistemas ou processos complexos. Em um mundo altamente competitivo a simulação torna-se uma ferramenta poderosa para o planejamento, projeto e controle de sistemas [Pegden 1994].

A utilização da simulação é propícia no estudo de pesquisa, na utilização e uso de técnicas de pesquisas operacionais, devido à sua grande versatilidade, poder e flexibilidade. A maioria dos sistemas já foram ou podem ser simulados [Valle 1997]. Alguns exemplos que se pode citar são:

- sistemas computacionais: componentes de *hardware*, sistemas operacionais, sistemas de *software*, redes de computadores, gerenciamento e estruturação de dados e processamento de informação.
- manufatura: linhas de montagem, produção automatizada, armazenagem automatizada, sistemas de controle de inventário, estudos de manutenção e integridade.
- negócios: análise de ações e mercadorias, política de preços, estratégias de mercado, estudos para aquisição, análise de fluxo de caixa e previsão.
- governo: táticas militares, resgate médico, proteção contra incêndios, serviços policiais, desenhos de estradas e controle de tráfego.
- ecologia e ambiente: poluição da água e purificação, previsão do tempo, análise de terremotos e tempestades, sistemas de energia solar e escoamento de produção.

Apesar de muitas das características da simulação serem desejáveis, existem vantagens e desvantagens na utilização da simulação. Dentre as vantagens pode-se citar:

- novos procedimentos, regras de decisões, estruturas organizacionais, fluxos de informações, podem ser explorados sem interromper operações em andamento.
- novos projetos de *hardware*, *layouts* físicos, sistemas de transporte, programas de *software*, podem ser testados sem comprometer recursos para aquisição ou implementação.
- hipóteses sobre como ou porque certo fenômeno ocorre, podem ser testadas no que diz respeito a viabilidade.
- o tempo pode ser controlado para executar várias operações rapidamente ou para executar lentamente, permitindo a observação de detalhes.
- engarrafamentos podem ser observados no fluxo de produto.
- um estudo de simulação pode apresentar provas de como o sistema funciona, em oposição a opiniões contrárias.

Por outro lado, existem também algumas desvantagens:

- a construção de modelos requer pessoal especializado, a qualidade da análise depende muito da qualidade do modelo e do talento do modelador.
- os resultados de uma simulação são, algumas vezes, difíceis de interpretar. Quando se coloca variáveis randômicas em um sistema real é difícil determinar se uma observação está relacionada a uma combinação interna do sistema ou se é efeito da casualidade randômica.
- a análise por simulação pode consumir muito tempo e ter um alto custo. Uma análise, muitas vezes, pode não ser viável em certo tempo e com determinados recursos. Pode ser mais viável um cálculo mais simples utilizando métodos analíticos.

5.2.1 Procedimentos para Avaliação de Desempenho

Na realização de uma avaliação de desempenho (incluindo a simulação) é necessário seguir alguns procedimentos e métodos. Esses aspectos são definidos em [Jain 1991], tais como:

- estabelecimento dos objetivos e definição do sistema: definir claramente os objetivos; planejar suporte, recursos, computadores, *hardware*, *software*; estudo do funcionamento do sistema.
- lista dos serviços e resultados: listar serviços, traçar o modelo conceitual do sistema com a definição dos seus componentes, interelações e fluxo de informações. Isso facilita a correta seleção das medidas a serem observadas e das cargas de trabalho a serem utilizadas.
- seleção das medidas: definir quais medidas são relevantes e que deverão ser consideradas.
- definição dos parâmetros: os parâmetros definidos são aqueles que afetam o funcionamento do sistema, que podem ser parâmetros do sistema ou da carga de trabalho.
- escolha dos fatores: definir quais fatores variam e quais ficam fixos durante a avaliação.
- seleção da técnica de avaliação: escolher a técnica adequada depende do tempo e dos recursos disponíveis para resolver o problema e o nível desejável de exatidão.
- escolha da carga de trabalho: a escolha da carga de trabalho adequada é um fator muito importante para a correta obtenção dos resultados.
- execução dos experimentos: escolher uma seqüência de experimentos que ofereça o máximo de informações com o mínimo de esforço.
- análise e interpretações dos dados: analisar e interpretar os dados gerados por uma simulação é uma arte, sendo um dos passos mais importantes na simulação.
- apresentação dos resultados: a finalização do trabalho ocorre com a apresentação e a documentação dos resultados.

O procedimento completo, entretanto, consiste realizar vários ciclos nos diversos passos citados.

5.2.2 Simulação do Mecanismo de Comunicação CRUX

De acordo com os aspectos apresentados em 5.2.1, os passos adotados foram os seguintes:

- estabelecimento dos objetivos e definição do sistema: o objetivo da avaliação é verificar o atendimento dos requisitos de previsibilidade para o sistema de comunicação do ambiente CRUX. 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: 0, 32, 64, 128, 256 e 512

  ⇨ número de canais[6]: 1, 2 e 4

  ⇨ taxa de chegada de mensagens: taxa[7]

---

[6] 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.

[7] Varia de um valor mínimo a um valor máximo, como será visto com maiores detalhes no item 5.3.3

- 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.
- 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 5.4.

Um maior detalhamento da modelagem é apresentado em 5.3. Além da apresentação esquemática dos modelos estático e dinâmico, através de figuras, é mostrado também a formulação e determinação de alguns parâmetros e fatores utilizados.

5.3 Modelos do Mecanismo de Comunicação CRUX

O mecanismo de comunicação do CRUX foi modelado de forma estática e de forma dinâmica. O diferencial, entre estes modelos, é o número de canais utilizado 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 5.1) 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).

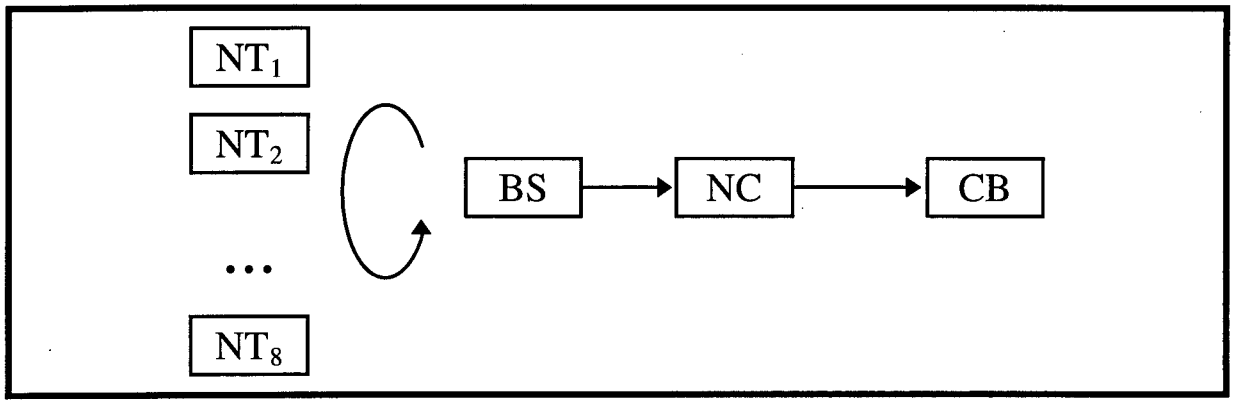

Figura 5.1 - Elementos básicos dos modelos de comunicação CRUX

- **Nós de Trabalho (NT):** responsáveis pela geração de mensagens (requisitando comunicação com outro nó), que podem ser de tamanho zero, quando é apenas estabelecida a conexão, até tamanho de 512 *bytes*. O tempo que eles utilizam para processar o envio da mensagem pela aplicação é t_{proc}, e aguardam pelo *pooling* do BS ordenando as mensagens pelos seus respectivos *deadlines*. Cada NT (Figura 5.2) possui 4 canais de comunicação com o CB e um canal com o NC. A estrutura do NT é semelhante em ambos os modelos.

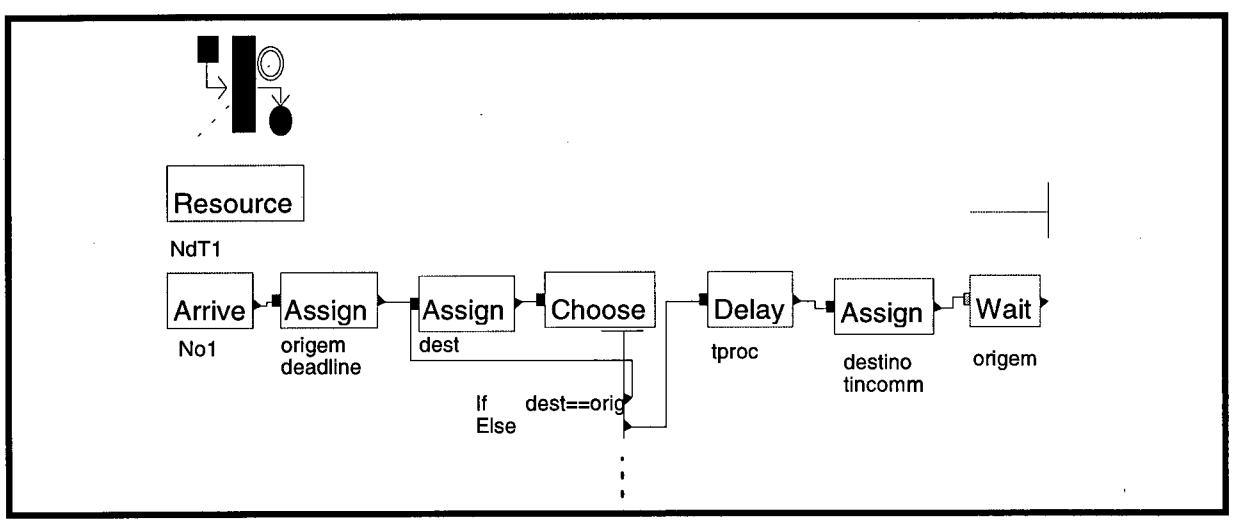

Figura 5.2 - Modelo do nó de trabalho no Arena

- **Barramento de Serviços (BS):** realiza a verificação contínua, a uma freqüência *freq*, nos NT. A estrutura do BS é idêntica nos dois modelos.

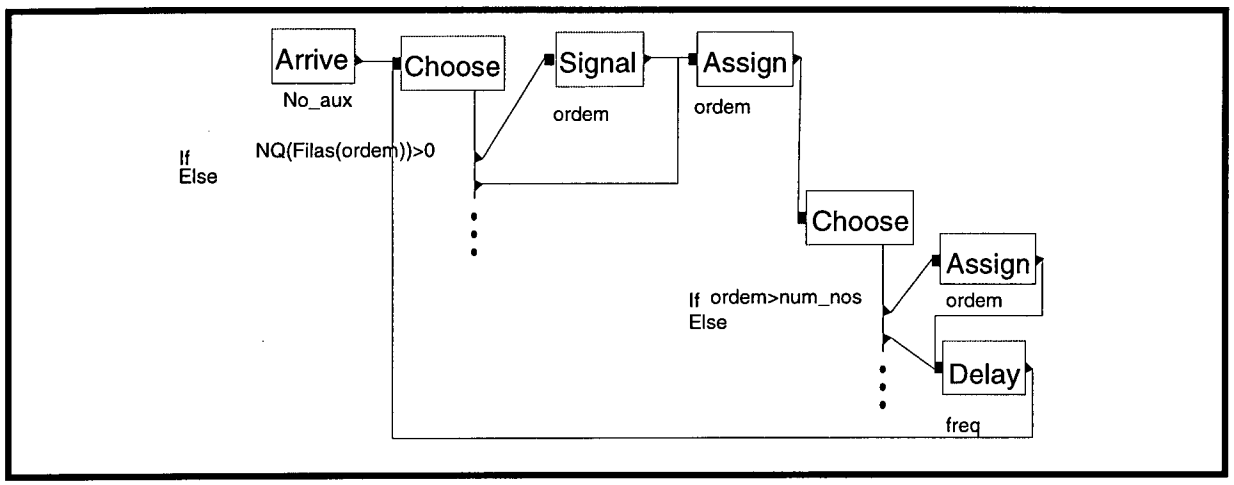

Figura 5.3 - Modelo do barramento de serviços no Arena

- **Nó de Controle (NC):** processa todos os pedidos de conexão em um tempo t_{ocup}, atendendo primeiramente as mensagens com maior prioridade. O modelo estático é mostrado na Figura 5.4 e o modelo dinâmico na

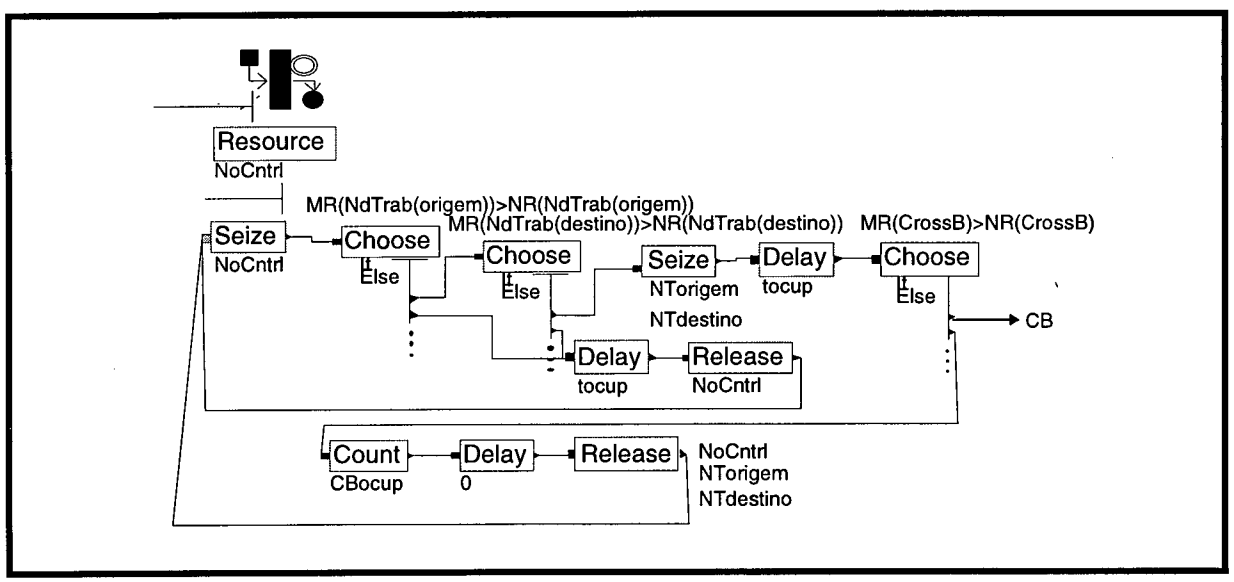

Figura 5.4 - Modelo estático do nó de controle no Arena

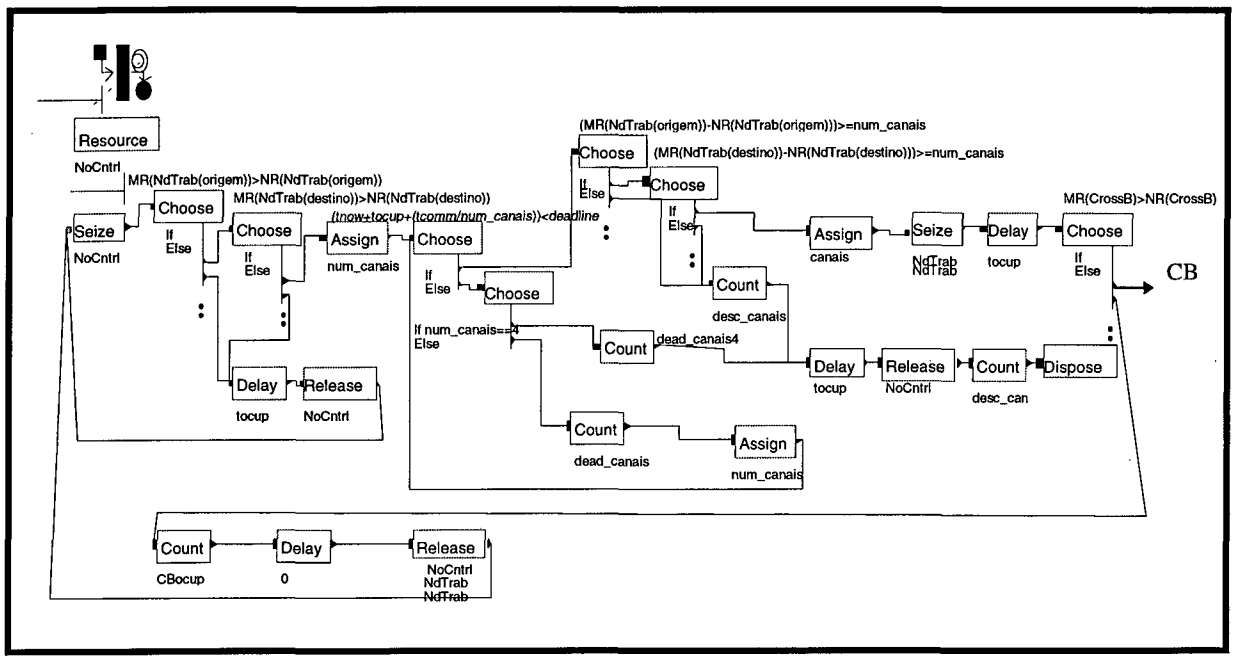

Figura 5.5 - Modelo dinâmico do nó de controle no Arena

- ***Crossbar* (CB):** 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. Esse número de canais é definido de forma fixa para o experimento no modelo estático e de forma dinâmica pelo NC no modelo dinâmico.

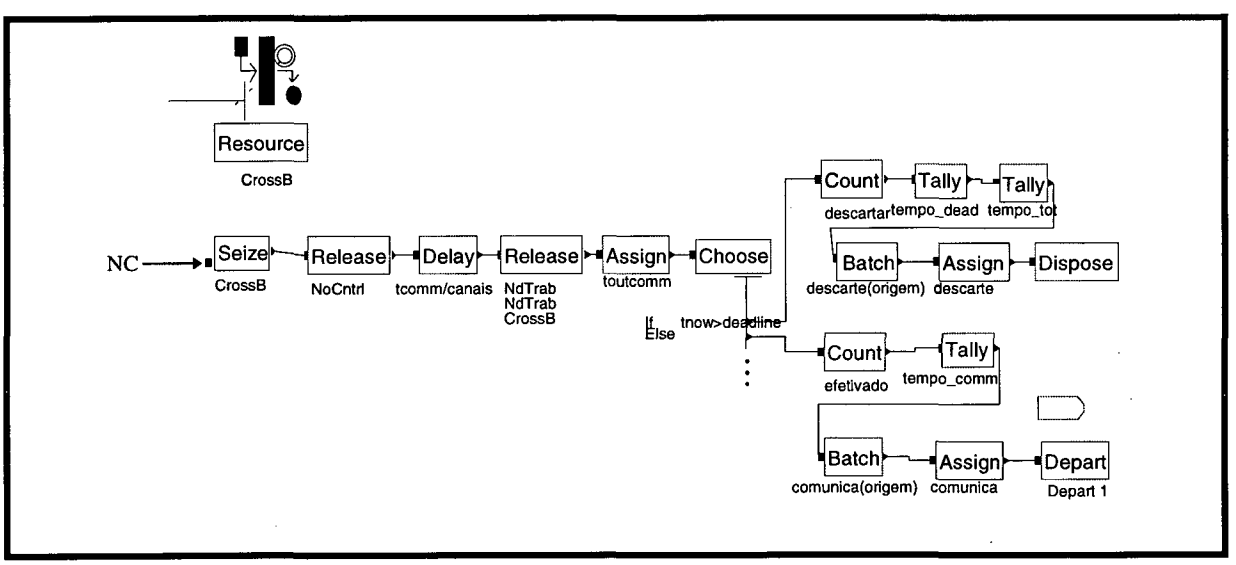

Figura 5.6 - Modelo do *crossbar* no Arena

A Figura 5.7 mostra, esquematicamente, o modelo do mecanismo de comunicação CRUX, na forma como foi construído no Arena. 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 utilizando a totalidade canais não for suficiente para cumprir o *deadline*, a mensagem é descartada. Obtido o número de canais necessários, é verificado a disponibilidade de recursos e feita a alocação destes. Decorrido o tempo de comunicação é contabilizado o número de mensagens que atendeu os seus respectivos *deadlines*.

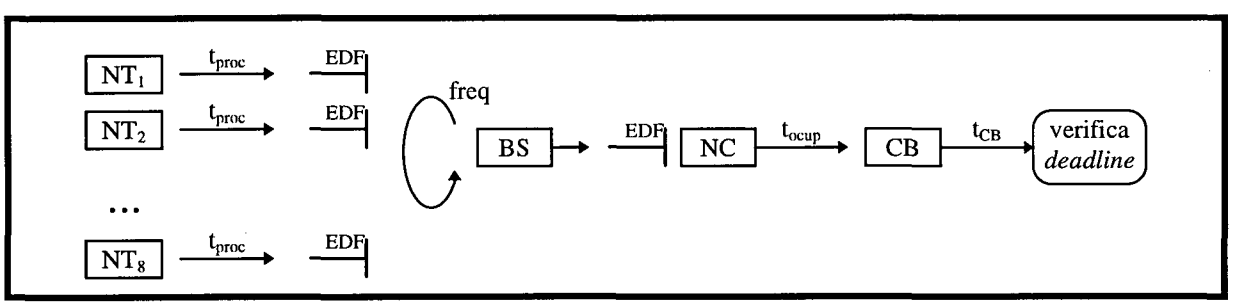

Figura 5.7 - Modelo de comunicação CRUX

5.3.1 Cálculo dos Tempos Mínimo e Máximo do Sistema

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.

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

5.3.2 Valores de *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. Deve-se ressaltar, entretanto, que o tempo gasto na comunicação segue a equação (5) e não a (2), pois o número de canais é um dos fatores que se deseja avaliar. Uma mensagem, cujo *deadline* não seria atendido utilizando apenas um canal, poderá ter seu *deadline* atendido se utilizar dois ou mais canais.

$$t_{CB} = t_{comm} \qquad (5)$$

5.3.3 Taxas de Geração de Mensagens

A taxa de geração de mensagens nos NT segue os valores das equações (6) (7) e (8). O valor mínimo da taxa corresponde aquele tempo mínimo estimado para que a mensagem percorra o sistema. O valor máximo da taxa equivale ao tempo estimado com todos os atrasos conhecidos. E o valor médio da taxa apresentado na equação (8) representa o valor intermediário entre a taxa mínima e a taxa máxima.

$$taxa_{minima} = t_{proc} + freq + t_{ocup} + t_{comm} \qquad (6)$$

$$taxa_{maxima} = t_{proc} + (NT * freq) + (NT * canais * t_{ocup}) + t_{comm} \qquad (7)$$

$$taxa_{media} = (taxa_{maxima} + taxa_{minima}) / 2 \qquad (8)$$

5.4 Apresentação dos Resultados

Os dados obtidos nos modelos dinâmico e estático foram resultados de um número diferente de experimentos, uma vez que, no modelo estático a variação no número de canais era feita de forma fixa para cada experimento.

No modelo estático os resultados foram obtidos através de nove experimentos em cada um dos seis níveis, o que totalizou 54 medidas realizadas. Os seis níveis correspondem a cada grupo de nove experimentos, onde o tamanho da mensagem não é variado. Os nove experimentos são obtidos pela variação nos valores do número de canais utilizados e da taxa de geração de mensagens. Em cada experimento foram realizadas dez medições e obtido um valor médio. Os valores apresentados na Tabela 5.1 correspondem ao percentual de comunicações efetivadas em cada experimento, ou seja, o número de mensagens que atenderam os seus respectivos *deadlines*. Os fatores variados nos experimentos foram:

- tamanho das mensagens: 0, 32, 64, 128, 256 e 512.
- número de canais utilizados: 1, 2 e 4.
- taxa de geração de mensagens: mínima, média e máxima.

No modelo dinâmico foram realizados três experimentos para cada um dos seis níveis, o que totalizou 18 medições. Os seis níveis correspondem a cada grupo de três experimentos, onde o tamanho da mensagem não é variado. Os três experimentos são obtidos pela variação no valor da taxa de geração de mensagens. Em cada experimento foram realizadas dez medições e obtido um valor médio. Os valores apresentados na Tabela 5.2 correspondem ao percentual de comunicações efetivadas em cada experimento, ou seja, o número de mensagens que atenderam os seus respectivos *deadlines*, e o percentual de mensagem que não teve seu *deadline* atendido para cada valor possível do número de canais. Os fatores variados nos experimentos foram:

- tamanho das mensagens: 0, 32, 64, 128, 256 e 512.
- número de canais utilizados: 1, 2 e 4.
- taxa de geração de mensagens: mínima, média e máxima.

Tabela 5.1 - Resultados da simulação do modelo estático

tam_mens	nº canais	taxa_max	taxa_med	taxa_min
0	1	86,32%	86,44%	86,48%
	2	86,32%	86,44%	86,48%
	4	86,32%	86,44%	86,48%
32	1	85,79%	85,66%	85,60%
	2	96,89%	96,95%	96,90%
	4	97,93%	97,89%	97,93%
64	1	85,31%	85,39%	85,35%
	2	96,43%	96,42%	96,29%
	4	98,05%	98,00%	97,96%
128	1	84,94%	85,02%	85,03%
	2	96,00%	96,06%	95,96%
	4	97,11%	96,77%	96,98%
256	1	84,99%	84,96%	85,04%
	2	95,71%	95,84%	95,77%
	4	96,46%	96,36%	96,00%
512	1	85,56%	85,31%	84,99%
	2	95,66%	95,51%	95,36%
	4	95,67%	94,72%	95,41%

Tabela 5.2 - Resultados da simulação do modelo dinâmico

tam_mens	taxa	%1 *	%2 *	%4 *	efetivado **
0	max	13,20%	13,20%	13,20%	86,80%
	med	13,19%	13,19%	13,19%	86,81%
	min	13,40%	13,40%	13,40%	86,60%
32	max	14,57%	3,36%	0,43%	96,64%
	med	14,52%	3,36%	0,42%	96,64%
	min	14,68%	3,37%	0,45%	96,63%
64	max	15,63%	3,64%	0,52%	96,37%
	med	15,60%	3,72%	0,58%	96,28%
	min	15,56%	3,75%	0,63%	96,25%
128	max	15,68%	3,83%	1,21%	96,18%
	med	16,09%	3,84%	1,16%	96,16%
	min	16,22%	3,86%	1,17%	96,14%
256	max	15,68%	3,73%	1,19%	96,27%
	med	15,63%	3,76%	1,14%	96,24%
	min	15,97%	3,84%	1,18%	96,16%
512	max	15,76%	3,66%	1,07%	96,34%
	med	15,73%	3,69%	1,23%	96,31%
	min	15,99%	3,90%	1,16%	96,10%

* Percentual de mensagens descartadas com um, dois e quatro canais.
** Percentual de mensagens que atenderam seus respectivos *deadlines*

5.5 Análise da Variância

Após a coleta dos resultados dos experimentos, foi realizada a análise da variância com o objetivo de determinar a influência de cada um dos fatores na previsibilidade do sistema. Para esta análise foi utilizado o *software* Statistica, versão 5.0, da empresa StatSoft Inc. Adotou-se um nível de significância de 0,05. A avaliação foi feita considerando os fatores (tamanho da mensagem, taxa de geração e número de canais) dois a dois. Uma vez que os fatores foram considerados dois a dois, utilizou-se a correção de Bonferroni [Galambos 1996] para corrigir o nível de significância. Na análise da variância não foram considerados os grupos de experimentos cujo tamanho de mensagem era zero, pois os valores obtidos nesses experimentos representam apenas o tempo de conexão, sem haver influência dos fatores considerados.

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 5.8, 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 5.9), 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.

Uma análise relevante das medidas obtidas para mensagens de 0 *bytes* pode ser feita considerando que o percentual obtido representa apenas o tempo de conexão, não influindo a tamanho da mensagem. Quando se utiliza mensagens com tamanhos maiores, o número de canais utilizados influencia no funcionamento do sistema, e conseqüentemente nos resultados obtidos.

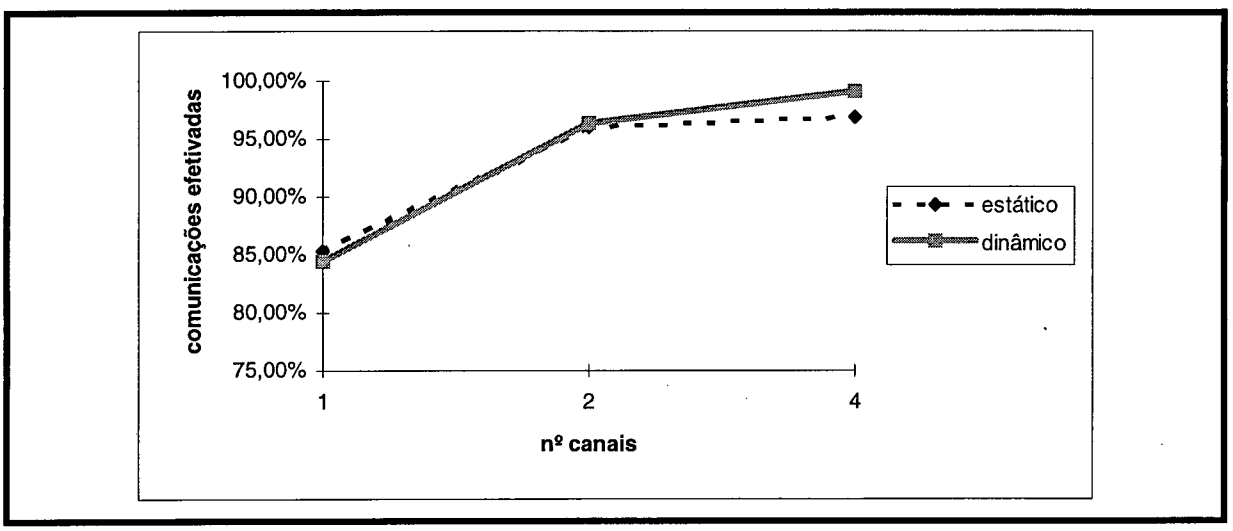

Figura 5.8 - Percentual de mensagens efetivadas de acordo com o número de canais

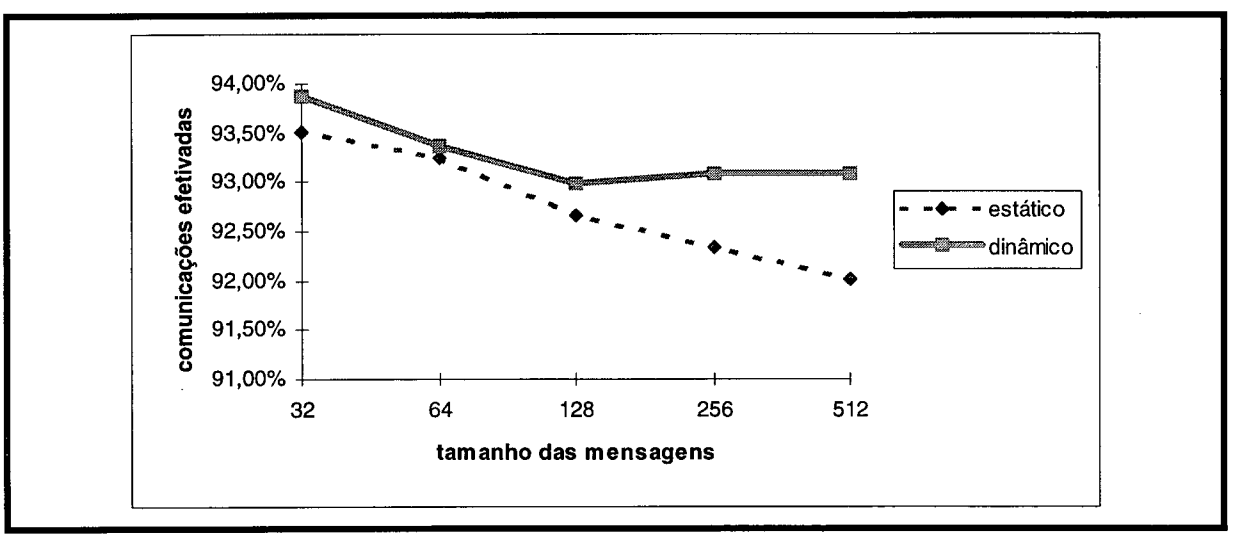

Figura 5.9 - Percentual de mensagens efetivadas de acordo com o tamanho (em *bytes*)

5.6 Sumário

Este capítulo tratou da avaliação de desempenho do mecanismo de comunicação do ambiente CRUX. Foram apresentados os métodos utilizados, os resultados obtidos e feita a análise da variância desses resultados. No próximo capítulo são apresentadas as conclusões finais e apresentadas as propostas para trabalhos futuros.

6. Conclusões

Aplicações de tempo real fazem, cada vez mais, parte do dia-a-dia e sua utilização ocorre em um número maior de áreas a cada dia. Recentemente, a importância dessas aplicações foi evidenciada 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 [Wilner 1997].

Este trabalho envolveu áreas, tais como: sistemas operacionais, processamento paralelo e avaliação de desempenho. Em sistemas operacionais, foi investigado questões envolvendo sistemas distribuídos e de tempo real, com o objetivo de adequar o ambiente de processamento paralelo CRUX para a execução de aplicações de tempo real, através do fornecimento de algumas primitivas básicas de sistemas operacionais de tempo real. A avaliação de desempenho foi realizada utilizando simulação, visando avaliar o desempenho e a previsibilidade dos mecanismos de comunicação.

O ponto inicial deste trabalho envolveu SOTR, onde foi realizado um estudo dos seus componentes, dos diversos tipos existentes (comerciais ou não) e da padronização POSIX proposta para esses sistemas pela IEEE. Esse estudo, juntamente com o levantamento das características do ambiente de aplicação da máquina CRUX permitiu apresentar um conjunto básico de primitivas de suporte para tempo real para esse ambiente. Em complemento, 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.

A contribuição desse trabalho está no fato de procurar permitir o desenvolvimento de aplicações com restrições temporais em um ambiente de processamento paralelo. Essas aplicações envolvem um espectro variado de assuntos, tais como: multimídia, robótica, sistemas médico-hospitalares, telecomunicações, controle de manufatura, controle de processos, dentre outros.

Durante o estudo de SOTR verificou-se que, dentre os sistemas existentes analisados, não existe um suporte total à padronização POSIX da IEEE. Em virtude da grande abrangência desse padrão, de apenas uma parte 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 um "portabilidade temporal". Esses fatos contribuíram para que este trabalho também apresente uma "compatibilidade parcial" com o padrão, isto é, as primitivas propostas são compatíveis com o padrão, mas são em um número mínimo que permita atender as necessidades de suporte de tempo real para a máquina CRUX e não no número proposto no POSIX.

Uma outra dificuldade encontrada foi na avaliação de desempenho dos mecanismos de comunicação do ambiente CRUX. Devido aos resultados obtidos na simulação serem muito próximos, foi necessário aumentar o número de replicações de cada experimento para, a partir de uma média desses valores, conseguir chegar a resultados conclusivos. Esse fato ocasionou uma revisão na modelagem e uma repetição nos experimentos. Mesmo com a utilização de um maior número de replicações ainda foi necessário realizar uma análise de variância dos resultados, obtidos na simulação, com o objetivo de confirmar as hipóteses levantadas e auxiliar nas considerações sobre os dados coletados.

Apesar das dificuldades encontradas no percurso deste trabalho, o objetivo de apresentar serviços de suporte para aplicações de tempo real e de realizar a avaliação do desempenho dos mecanismos de comunicação foram alcançados. Entretanto, existem outros pontos a serem tratados, tais como:

- implementação dos serviços de suporte para aplicações de tempo real no ambiente CRUX.
- novas análises de avaliação de desempenho quanto ao esquema de escalonamento proposto.
- análises de avaliação de desempenho dos mecanismos de comunicação com outros tipos de arquitetura, baseados em redes estáticas e redes dinâmicas.

Em virtude da escassez de tempo e outros fatores limitantes, não foi possível realizar todos os pontos desejáveis. Fica, então, a proposta para que esses pontos e outros que possam ser levantados sejam alvo de trabalhos futuros.

Referências Bibliográficas

[Abrossimov 1996] Abrossimov, V. et al., **STREAM-v2 kernel architecture and API specificacion version 1.0**. Esprit Project Documentation, STREAM Technical Committee, Mar 1996.

[Audsley 1990] Audsley, N. et al., **Timeliness: summary and conclusions**. Esprit Bra Project 3092: Predictably Dependable Computing Systems, First Year Report Task B, v. 2, 1990.

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

[Corso 1993] Corso, T. B., **Ambiente para programação paralela em multicomputador**. Relatório técnico n. 1, INE;UFSC, Florianópolis, nov 1993.

[Dijkstra 1965] Dijkstra, E. W., **Cooperating sequential processes**. Technical Report EWD-123. Eindhoven, Holanda: Technological University, 1965.

[Galambos 1996] Galambos, J. e Simoneli, I., **Bonferroni - type inequalities with applications (probability and its applications)**. Springer Verlog, Sep 1996.

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

[Gien 1995] Gien, M., **Evolution of the CHORUS open microkernel architecture: the STREAM project**. V IEEE Workshop on Future Trends in Distributed Computing Systems, Cheju Island, Aug 1995.

[Humphrey 1995] Humphrey, M., Wallace, G. e Stankovic, J. A., **Kernel-level threads for dynamic, hard real-time environments**. Spring Project Documentation, University of Massachusetts, Amherst, MA, 1995.

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

[Jensen 1985] Jensen, E. D., Locke, C. D., Tokuda, H., **A time-driven scheduling model for real-time operating systems**. Proceedings of the Real-Time Systems Symposium, Dec 1985.

[Kopetz 1993] Kopetz, H., Veríssimo, P., **Real time and dependability concepts**. In: Distributed Systems, chapter 16, 2. Ed., S. Mullender: Amsterdam, p. 411-446, Mar 1993.

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

[Leung 1982] 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.

[Levi 1990] Levi, S. T., Agrawala, A. K., **Real-time system design**. McGraw-Hill, 1990.

[Liu 1973] 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.

[Lo 1997] 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.

[Montez 1995] Montez, C. B., **Um sistema operacional com micronúcleo distribuído e um simulador multiprogramado de multicomputador**. Dissertação de mestrado, CPGCC/UFSC, Florianópolis, mai 1995.

[Montez 1997] Montez, C. B., **Um modelo de programação para aplicações de tempo real em sistemas abertos**. Monografia de exame de qualificação para o doutorado em engenharia elétrica, LCMI/UFSC, Florianópolis, jul 1997.

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

[Oliveira 1997] Oliveira, R. S. de, **Escalonamento de tarefas imprecisas em ambiente distribuído**. Tese de doutorado, LCMI/UFSC, Florianópolis, jan 1997.

[Pegden 1994] Pegden, C. D., Shanon, R. E. e Sadowski, R. P., **Arena professional edition reference guide**, Sewicley PA, 1994.

[QNX 1997] **QNX/neutrino microkernel**, Disponível pela Internet via www: <URL: http://www.qnx.com/docs/neutrino/sysarch/>, Jun 1997.

[Ramamritham 1989] Ramamritham, K., Stankovic, J. A., Zhao, W., **Distributed scheduling of tasks with deadlines and resources requirements**. IEEE Transactions on Computers, v. 38, n. 8, p. 1110-1123, Aug 1989.

[Ramamritham 1994] Ramamritham K. e Stankovic J. A., **Scheduling algoritms and operating systems support for real time systems**. Proceedings of the IEEE, v. 82, n. 1, p. 61-65, Jan 94

[Rozier 1991] Rozier, M. et. Al, **Chorus distributed operating systems**. Computing Systems Journal, v. 1, n. 4, p. 305-370, Dec. 1988, revised in ***Overview of Chorus distributed operating systems***. Chorus Systems, Feb 1991.

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

[Silberschatz 1991] Silberschatz, A., Peterson, J. L., Galvin, P. B., **Operating systems concepts**. 3. Ed., Addison-Wesley, 1991.

[Silva 1996] Silva, V. A. da, **Multicomputador nó//: implementação de primitivas básicas de comunicação e avaliação de desempenho**. Dissertação de mestrado, CPGCC/UFSC, Florianópolis, mai 1996.

[Stankovic 1988] 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.

[Stankovic 1991] Stankovic J. A. e Ramamritham K., **The spring kernel: a new paradigm for real time systems**. IEEE Software, v. 8, n. 3, p. 62-72, Mai 1991.

[Stemmer 1997] Stemmer, M. R., **Engenharia de sistemas tempo real II**. Notas de aula da disciplina engenharia de sistemas tempo real II do curso de pós-graduação em controle, automação e informática industrial, LCMI/UFSC, jun 1997.

[Tanenbaum 1992] Tanenbaum A. S., **Modern operating systems**. Prentice-Hall: New Jersey, 1992.

[Tanenbaum 1995] Tanenbaum, A. S., **Distributed operating systems**. Prentice-Hall: New Jersey, 1995.

[Teo 1995] Teo, M. S. E., **A preliminary look at spring and POSIX.4**. Spring Project Documentation, University of Massachusetts, Amherst, MA, 1995.

[Valle 1997] Valle Filho, A. M., **Desenvolvimento de templates para modelagem, simulação e avaliação de desempenho em computadores com arquitetura paralela**. Dissertação de mestrado, CPGCC/UFSC, Florianópolis, jun 1997.

[Wilner 1997] 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, California, 1997.

Anexo A

Neste apêndice são apresentadas tabelas com as primitivas para aplicações de tempo real do padrão POSIX da IEEE. A Tabela A.1 refere-se ao POSIX.1b [Gallmeister 1995] e a Tabela A.2 ao POSIX.1c [Nichols 1996].

Tabela A.1 - Primitivas do padrão POSIX.1b

Nome Opcional	Funcionalidade		
Gerenciamento de Processos:	sinais; escalonamento; "clocks" e "timers".		
Não Opcional	Enfileiramento de sinais; sinais tempo-real.		
	(SA_SIGINFO, SIGRTMIN, SIGRTMAX)		
_POSIX_REALTIME_SIGNALS	Funções adicionais de sinais		
	• *sigwaitinfo*	• *sigtimedwait*	• *sigqueue*
_POSIX_PRIORITY_SCHEDULING	Controle de escalonamento de processo		
	• *sched_setparam*	• *sched_getparam*	
	• *sched_setscheduler*	• *sched_getscheduler*	
	• *sched_yield,*	• *sched_rr_get_interval*	
	• *sched_get_priority_max*	• *sched_get_priority_min*	
_POSIX_TIMERS	"Clocks" e "Timers"		
	• *clock_settime*	• *clock_gettime*	• *clock_getres*
	• *timer_create*	• *timer_delete*	• *timer_settime*
	• *timer_gettime*	• *timer_getoverrun*	• *nanosleep*
Sincronização/Comunicação InterProcessos:	passagem de mensagem; semáforo e memória compartilhada		
_POSIX_MESSAGE_PASSING	Filas de Mensagens		
	• *mq_open*	• *mq_close*	• *mq_unlink*
	• *mq_send*	• *mq_receive*	• *mq_notify*
	• *mq_setattr*	• *mq_getattr*	
_POSIX_SEMAPHORES	Semáforos contadores		
	• *sem_init*	• *sem_destroy*	• *sem_unlink*
	• *sem_open*	• *sem_close*	• *sem_wait*
	• *sem_trywait*	• *sem_post*	• *sem_getvalue*
_POSIX_SHARED_MEMORY_OBJECTS	Memória compartilhada		
	• *mmap*	• *munmap*	• *shm_unlink*
	• *shm_open*	• *shm_close*	• *ftruncate*

Entrada e Saída:	E/S priorizada, síncrona e assíncrona.
_POSIX_ASYNCHRONOUS_IO	Entrada/Saída assíncrona • *aio_read* • *aio_write* • *lio_listio* • *aio_suspend* • *aio_cancel* • *aio_error* • *aio_return* • *aio_fsync* (*) • (*): se e somente se _POSIX_SYNCHRONIZED_IO
_POSIX_PRIORITIZED_IO	E/S assíncrona priorizada: modifica a ordem da fila de E/S.
_POSIX_SYNCHRONOUS_IO	Garantia de que os dados de um arquivo serão sempre os dados atualizados no disco e não de memória temporária. • *fdatasync* • *msync* (**) • *aio_fsync* (***) • adicionais para *open* e *fcntl* (**) se e somente se _POSIX_MAPPED_FILES (***) se e somente se _POSIX_ASYNCHRONOUS_IO
_POSIX_FSYNC	Garante que um arquivo no disco não estará desatualizado.
_POSIX_MAPPED_FILES	Arquivos mapeados na memória • *mmap* • *munmap* • *ftruncate* • *msync* (#) (#) se e somente se _POSIX_SYNCHRONOUS_IO
Gerenciamento de Memória:	bloqueio e proteção de memória.
_POSIX_MEMLOCK	Bloqueia toda a memória para evitar paginamento/swapping • *mlockall* • *munlockall*
_POSIX_MEMORY_RANGE	Bloqueia uma faixa de memória • *mlock* • *munlockall*
_POSIX_MEMORY_PROTETION	Habilidade de estabelecer proteção de memória • *mprotect*

Tabela A.2 - Primitivas do padrão POSIX.1c

SINCRONIZAÇÃO	
• pthread_join	suspende a thread até que outra termine
• pthread_once	garante que a inicialização de rotinas seja feita uma única vez
MUTEXES	
• pthread_mutexattr_destroy	destrói um atributo de objeto mutex
• pthread_mutexattr_getprioceiling	obtém a prioridade máxima do atributo
• pthread_mutexattr_getprotocol	obtém o protocolo do atributo
• pthread_mutexattr_getpshared	obtém situação do processo compartilhado do atributo de objeto
• pthread_mutexattr_init	inicializa um atributo
• pthread_mutexattr_setprioceiling	estabelece prioridade máxima do atributo
• pthread_mutexattr_setprotocol	estabelece protocolo do atributo: herança, protegido, nenhum.
• pthread_mutexattr_setpshared	estabelece situação do processo: compartilhado ou privado.
• pthread_mutex_destroy	destrói um mutex
• pthread_mutex_init	inicializa um mutex com os atributos especificados no objeto
• pthread_mutex_lock	bloqueia um mutex, suspendendo a thread caso ele já esteja bloqueado
• pthread_mutex_trylock	tenta bloquear um mutex, sem suspender a thread
• pthread_mutex_unlock	desbloqueia mutex, libera thread suspensa (se houver), conforme prioridade
VARIÁVEIS CONDICIONAIS	
• pthread_condattr_destroy	destrói um atributo de objeto variável condicional
• pthread_condattr_getpshared	obtém situação do processo compartilhado do atributo de objeto
• pthread_condattr_init	inicializa um atributo
• pthread_condattr_setpshared	estabelece situação do processo: compartilhado ou privado.
• pthread_cond_broadcast	desbloqueia todas as threads que esperam um variável condicional
• pthread_cond_destroy	destrói uma variável condicional
• pthread_cond_init	inicializa variável condicional com atributos especificados no objeto
• pthread_cond_signal	desbloqueia thread bloqueada, conforme prioridade de escalonamento
• pthread_cond_wait	desbloqueia mutex, suspende thread até sinalização à variável condicional
• pthread_cond_timedwait	desbloqueia mutex, suspende thread até sinalização à variável condicional ou expiração do tempo especificado
BLOQUEIO LEITURA/ESCRITA	
• pthread_rdwr_init_np	inicializa bloqueio leitura/escrita
• pthread_rdwr_rlock_np	bloqueia leitura
• pthread_rdwr_wlock_np	bloqueia escrita
• pthread_rdwr_runlock_np	desbloqueia leitura
• pthread_rdwr_wunlock_np	desbloqueia escrita

GERENCIAMENTO DE THREADS

- pthread_atfork — declara procedimentos chamados antes e depois da chamada "fork"
- pthread_attr_destroy — destrói um atributo de objeto thread
- pthread_attr_init — inicializa um atributo de objeto thread
- pthread_cancel — cancela thread específica
- pthread_create — cria uma thread com os atributos especificados no objeto
- pthread_detach — marca uma estrutura de dados interna da thread para deleção
- pthread_equal — compara uma thread com outra
- pthread_exit — termina a chamada de uma thread, retornando valor especificado
- pthread_getspecific — obtém os dados e a chave associada com a thread específica
- pthread_setspecific — estabelece os dados e a chave associada com a thread específica
- pthread_key_create — gera chave única de thread específica, visível a todas as demais no processo
- pthread_key_delete — deleta chave de uma thread específica
- pthread_setcancelstate — estabelece o estado de cancelamento: habilitado/desabilitado
- pthread_setcanceltype — estabelece se o estado de cancelamento pode ser mudado durante a execução da thread
- pthread_testcancel — requer que qualquer pedido de cancelamento pendente seja enviado a thread que solicitou o pedido
- pthread_self — obtém a thread que interage com a thread chamada
- pthread_attr_getdetachstate — obtém a situação em relação ao estado de isolamento de uma thread
- pthread_attr_setdetachstate — ajusta o estado de isolamento de uma thread: isolada ou associada
- pthread_attr_getstackaddr — obtém o endereço da pilha da thread
- pthread_attr_getstacksize — obtém o tamanho da pilha da thread
- pthread_attr_setstackaddr — estabelece o endereço de pilha da thread
- pthread_attr_setstacksize — estabelece o tamanho da pilha da thread
- pthread_cleanup_pop — remove uma rotina do topo da pilha de remoção da thread
- pthread_cleanup_push — coloca uma rotina no topo da pilha de remoção da thread

ESCALONAMENTO

- pthread_attr_getinheritsched — obtém a situação de herança de escalonamento de uma thread
- pthread_attr_getschedparam — obtém parâmetros associados com a política de escalonamento
- pthread_attr_getshedpolicy — obtém a situação da política de escalonamento
- pthread_attr_getscope — obtém a situação do escopo do escalonamento
- pthread_attr_setinheritsched — obtém a situação
- pthread_attr_setschedparam — estabelece parâmetros associados com a política de escalonamento
- pthread_attr_setshedpolicy — estabelece a situação da política de escalonamento
- pthread_getschedparam — obtém as políticas e parâmetros de escalonamento
- pthread_setschedparam — estabelece as políticas e parâmetros de escalonamento
- pthread_attr_setscope — estabelece a situação do escopo do escalonamento

SINAIS

- pthread_kill — envia sinal para terminar uma thread específica
- pthread_sigmask — examina ou modifica a máscara do sinal