UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Karlos H. Budag

Implementação do núcleo do sistema operacional distribuído *ACrux*

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

Prof. Thadeu Botteri Corso, Dr.
Orientador

Florianópolis, dezembro de 2002

Implementação do núcleo do sistema operacional distribuído *ACrux*

Karlos H. Budag

Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação Área de Concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

Prof. Fernando Alvaro Ostuni Gauthier, Dr.
Coordenador do Curso

Banca Examinadora

Prof. Thadeu Botteri Corso, Dr.
Presidente

Prof. José Mazzucco Jr., Dr.

Prof. Luis Fernando Friedrich, Dr.

Prof. Rômulo Silva de Oliveira, Dr.

Sumário

Lista de Figuras

Resumo

O presente trabalho está inserido no âmbito do projeto *Crux*, cujo objetivo é a construção de um *cluster* de computadores interligado por uma rede dinâmica implementada através de um *crossbar* e de um barramento. Tal arquitetura depende do desenvolvimento de uma série de componentes, como um sistema operacional distribuído, processos servidores (de arquivos, de comunicação, etc.) e *software* que possa fazer uso de tais recursos. A implementação das primitivas de comunicação do núcleo do sistema operacional distribuído da arquitetura *Crux*, o *ACrux*, é o objetivo deste trabalho. O sistema operacional *Linux* foi o escolhido para dar o suporte inicial ao *cluster*, e o método de implementação utilizado foi a criação dinâmica de novas chamadas de sistema através da instalação de módulos carregáveis do núcleo (*LKM*s). O servidor de comunicações, peça chave do sistema operacional, também está incluído. Sua implementação consiste em um processo de usuário capaz de utilizar as novas chamadas de sistema para gerenciar as comunicações realizadas entre os nós de trabalho do *cluster*. Na presente implementação, a arquitetura é capaz de utilizar tanto redes TCP/IP quanto *FireWire* (IEEE1394).

Abstract

The present work is part of the *Crux* project, which aims into developing a cluster of computers interconnected by a dynamic network built using a crossbar and a bus network. Such architecture depends of the development of a serie of components such as a distributed operating system, server processes (file server, communication server, etc.) and software capable of using such resources. The implementation of the communication primitives of *Crux*'s distributed operating system, named *ACrux*, is the main objective of this work. The *Linux* operating system was chosen to offer the initial support to the cluster, and the implementation method used was the dynamic creation of new system calls through installing loadable kernel modules (*LKM*s). The communication server, key part of the operating system, is also included. It's implementation consists in a user process capable of employing these new system calls to manage the communications between the worker nodes of the cluster. In this implementation, the architecture is able of using both TCP/IP and *FireWire* (IEEE1394) networks.

1 Introdução

O presente trabalho está inserido no âmbito do projeto *Crux*, cujo objetivo é a construção de um *cluster* de computadores interligado por uma *rede dinâmica* implementada através de um *crossbar* e de um barramento. Os *clusters* de computadores são tema de vários projetos, atualmente, pelas inúmeras vantagens que oferecem, sendo a principal a de permitir a construção de máquinas paralelas de alto desempenho a custos relativamente reduzidos. A arquitetura *Crux* se justifica pelas características que a diferenciam de outros *clusters* conhecidos, como a utilização da rede dinâmica que permite a *inexistência de roteamento* e as *comunicações diretas* entre processos.

Tal sistema depende do desenvolvimento de uma série de componentes, como um sistema operacional distribuído, processos servidores (de arquivos, de comunicação, etc.) e *software* que possa fazer uso de tais recursos. A implementação das primitivas de comunicação do núcleo do sistema operacional distribuído da arquitetura *Crux*, o *ACrux*, é o objetivo deste trabalho. O servidor de comunicações, peça chave do sistema operacional, também está incluído.

O primeiro capítulo deste trabalho contém a revisão da literatura, onde são apresentados diversos conceitos utilizados no decorrer de todo o trabalho. São abordados temas como taxonomia das arquiteturas paralelas, multiprocessadores, multicomputadores, suas vantagens e desvantagens.

O segundo capítulo apresenta o resultado do estudo da arquitetura *Crux*, necessário para se conhecer as especificações do núcleo do sistema operacional e o funcionamento esperado do servidor de comunicações. São apresentados os componentes físicos e lógicos da arquitetura, e são comentadas as características que a diferenciam de outras arquiteturas semelhantes.

O terceiro capítulo se dedica a expor o conhecimento adquirido no estudo do sistema operacional *Linux*. Por ter sido ele o sistema operacional escolhido para abrigar o *ACrux*, foi necessário conhecer com alguma profundidade o núcleo do *Linux* para atender aos requisitos da implementação. São apresentadas e comparadas duas técnicas para criação de

chamadas de sistema no núcleo do *Linux*, e algumas das particularidades da programação em modo privilegiado.

O quarto capítulo apresenta o resultado final do projeto. As implementações das primitivas e do servidor são analisadas, e algumas das alterações nas especificações originais da arquitetura *Crux* são comentadas. Características adicionais, como suporte a redes TCP/IP e a extensões da arquitetura, também são apresentadas.

2 *Arquiteturas paralelas*

Este capítulo pretende ser uma breve revisão de temas relacionados à área de estudo denominada *arquiteturas paralelas*. Essa revisão parece relevante na medida em que o presente trabalho tem como objeto um *cluster* de computadores, configuração englobada pela área em questão.

Este capítulo apresenta a definição de arquitetura paralela e as motivações (seção 2.1) por trás da construção de máquinas baseadas nesse paradigma. São abordadas também algumas taxonomias capazes de agrupar as máquinas paralelas de acordo com suas características comuns, facilitando o estudo da área (seção 2.2). Uma das categorias introduzida pela *taxonomia de Flynn*, a dos computadores MIMD, por ser mais relevante a este trabalho, é abordada em maiores detalhes (seção 2.3). Finalmente, discutem-se alguns dos problemas inerentes à utilização de máquinas paralelas (seção 2.4).

De modo geral, este capítulo pode ser considerado uma compilação do conhecimento disperso em várias obras. A não ser nos locais onde expressamente mencionado, a fonte das informações mencionadas a seguir é o conjunto da seguinte bibliografia: Projeto Beowulf [1], Bogo [2], Boing [3], Campos [4], Cancian [5], Comer [7], Corso [8], Projeto Gamma [9], Gavilan [10], Merkle [12], Montez [13], Plentz [14], Rech [15], Silberschatz [16], Silva [17], Tanenbaum [18] [19], Zeferino [21] e alguns sítios da *internet* [6] [11].

2.1 Definição e Motivação

Ao se tentar definir o conceito de *arquitetura paralela*, deixa-se implícito a existência de arquiteturas não-paralelas. Uma *arquitetura não-paralela* é toda arquitetura que possui apenas uma unidade de processamento (*CPU*) e que, por essa razão, em um dado momento, possui apenas um fluxo de instruções sendo executado.

Já uma *arquitetura paralela*, em contraste, é aquela que possui mais de uma unidade de processamento, permitindo que mais de um fluxo de instruções seja executado em um dado momento. Mais detalhes serão apresentados, posteriormente neste capítulo, ao

abordarmos algumas das classificações de arquiteturas paralelas existentes na literatura.

Existem várias motivações por trás da construção de uma arquitetura paralela. Algumas delas, que podem ser ditas principais, são *economia* e *desempenho*. A *economia* que se consegue com uma arquitetura paralela deve ser entendida em relação a um sistema não-paralelo de desempenho equivalente, e é possível devido a uma relação conhecida como *lei de Grosch* [18]. Essa lei afirma que a razão entre velocidade e o custo de um processador não é constante; um processador com o dobro de desempenho terá um custo quatro vezes maior. Dado um computador com velocidade x e custo y, e sendo que se deseja aumentar o desempenho do sistema para $2x$, o custo desse *upgrade* será de $4y$ se a opção for por um sistema monoprocessado. No entanto, ao se colocar dois sistemas de desempenho x em paralelo, teremos uma máquina com velocidade $2x$ a um custo de $2y$. É a lei de *Grosch* em ação.

A busca de *desempenho* que motiva a construção de uma máquina paralela parte do simples fato de que, mesmo que se construa um processador mais rápido do que qualquer configuração paralela existente atualmente, basta colocar dois desses superprocessadores em paralelo para que tenhamos um novo computador de desempenho, teoricamente, duas vezes superior. Pode mesmo ser impossível, com a tecnologia existente em um dado momento, construir um processador mais potente; no entanto, sempre se pode adicionar mais processadores a um sistema paralelo, tornando-o teoricamente cada vez mais veloz.

Nesse ponto, cabe uma observação. No parágrafo anterior, a palavra *teoricamente* aparece diversas vezes. Isso porque a freqüência de execução dos processadores de um sistema paralelo não é a única variável a ser considerada quando se avalia o desempenho de um sistema. Podem existir gargalos em outros integrantes do sistema, como no meio de comunicação entre os processadores ou na própria natureza da tarefa sendo executada. Apenas em condições ideais pode-se afirmar que a duplicação da velocidade total de execução dos processadores que compõem uma arquitetura paralela é capaz de reduzir pela metade o tempo de execução de uma determinada tarefa.

Pode-se considerar que a *confiabilidade* também é uma das principais motivações para a construção de uma arquitetura paralela. Ela faz mais sentido, no entanto, no contexto de *multicomputadores*, um subconjunto dessas arquiteturas. Nesse tipo de configuração, existem diversas máquinas independentes conectadas a um meio físico capaz de interligá-las. A confiabilidade se encontra no fato de que, se uma das máquinas da rede falhar, outra pode assumir seu lugar imediatamente, fazendo com que os serviços oferecidos pela máquina paralela nunca cessem completamente (pode haver, é claro, uma queda de desempenho), a não ser em casos extremos (a queda total do meio de comunicação, por

exemplo). Em um *multiprocessador*, configuração em que existem diversas *CPU*s dentro de uma mesma máquina, a confiabilidade fica, de certa forma, comprometida; se ocorrer um problema considerado corriqueiro como, por exemplo, uma falha na fonte de alimentação da máquina, todos os seus processadores serão desativados. Existem máquinas com fontes de alimentação redundantes, é claro, mas isso não vêm ao caso; se alguém tropeçar no cabo de alimentação da máquina multiprocessadora, a fonte redundante não será de grande ajuda. O que se deseja é ilustrar o fato de que, nesse tipo de arquitetura, vários componentes de um sistema multiprocessador dependem dos mesmos recursos. Se, para uma determinada aplicação, a confiabilidade é fundamental, e não se deseja abrir mão dos multiprocessadores, pode-se sempre construir um multicomputador composto por multiprocessadores. Os custos certamente irão subir, mas a confiabilidade global do sistema também. As arquiteturas do tipo multiprocessador e multicomputador serão abordadas mais adiante, com maiores detalhes, ainda neste capítulo (seção 2.3).

Outras motivações para a construção de máquinas paralelas podem ser encontradas na literatura. No contexto de multicomputadores, *Tanenbaum* [18] cita economia, desempenho, aplicabilidade, confiabilidade e escalabilidade. *Silberschatz* e *Galvin* [16] mencionam compartilhamento, desempenho, confiabilidade e comunicação.

2.2 Classificação das arquiteturas paralelas

As *classificações* desempenham um papel importante em todas as áreas do conhecimento humano. Elas permitem estruturar e agrupar as informações, facilitando a tarefa de manuseá-las. Não seria a área de arquiteturas paralelas a única a escapar das taxonomias, sendo que as mais conhecidas são abordadas a seguir.

2.2.1 Taxonomia de *Flynn*

A *taxonomia de Flynn* é a classificação de arquiteturas computacionais mais aceita atualmente. Ela organiza as arquiteturas levando em conta a unicidade ou não de dois de seus elementos: os fluxos de instruções e os fluxos de dados.

A cada fluxo de instruções em execução simultânea em uma arquitetura deve estar associado uma unidade de controle capaz de decodificar esse fluxo e comandar uma ou mais unidades de processamento. Já a quantidade de unidades de processamento não está diretamente relacionada com o número de fluxos de dados que uma máquina é capaz de processar simultaneamente; a quantidade de *CPU*s existente em uma arquitetura depende tanto dos fluxos de dados como dos fluxos de instruções.

Por estarmos relacionado duas variáveis entre si, cada uma com dois valores possíveis, temos, ao todo, quatro combinações. Esse é portanto o número de categorias existentes na taxonomia de *Flynn*, quais sejam:

SISD – Single Instruction Single Data stream: Nessa categoria se encontram as máquinas consideradas não-paralelas, como aquelas que seguem a arquitetura clássica de *Von Neumann*. É a categoria em que se encaixa a grande maioria das máquinas comerciais atuais. Nelas, existe apenas um fluxo de instruções, bastando então uma unidade de controle. Existindo apenas um fluxo de instruções e um fluxo de dados, basta uma unidade de processamento. A figura 1 descreve esquematicamente esse tipo de arquitetura.

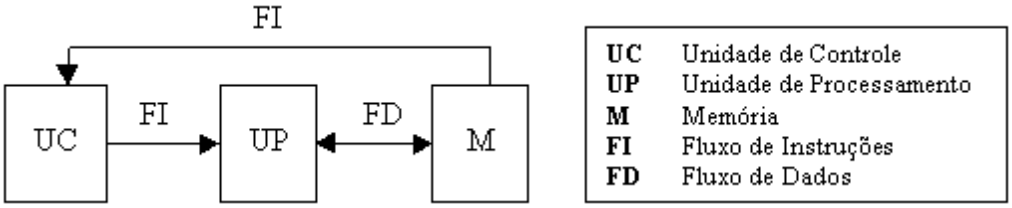

Figura 1: Arquitetura SISD, segundo *Flynn*

SIMD – Single Instruction Multiple Data stream: Neste tipo de arquitetura, temos um único fluxo de instruções, e, logo, uma única unidade de controle. No entanto, como existem diversos fluxos de dados, são necessárias múltiplas unidades de processamento. A unidade de controle decodifica o fluxo de instruções e comanda as várias *CPU*s existentes nesse tipo de máquina, todas elas realizando a mesma operação sobre dados diferentes. Os *computadores matriciais* são o exemplo mais conhecido dessa categoria de arquiteturas (veja figura 2).

MISD – Multiple Instruction Single Data stream: Nessa categoria, encontramos máquinas executando múltiplos fluxos de instruções simultaneamente, conseqüentemente possuindo várias unidades de controle. Elas operam sobre um único fluxo de dados, mas como possuem múltiplos fluxos de instruções, devem ser constituídas por múltiplos processadores. Na verdade, existe divergência na literatura com respeito a categoria dos computadores MISD. Alguns autores consideram que os *computadores sistólicos*, máquinas com vários processadores operando em *pipeline* sobre os mesmos dados, são representantes dessa categoria (figura 3); outros consideram que mesmo computadores com um único processador, mas que possuam o recurso de *pipelining*, podem ser considerados máquinas MISD; outros ainda afirmam que tais máquinas não existem.

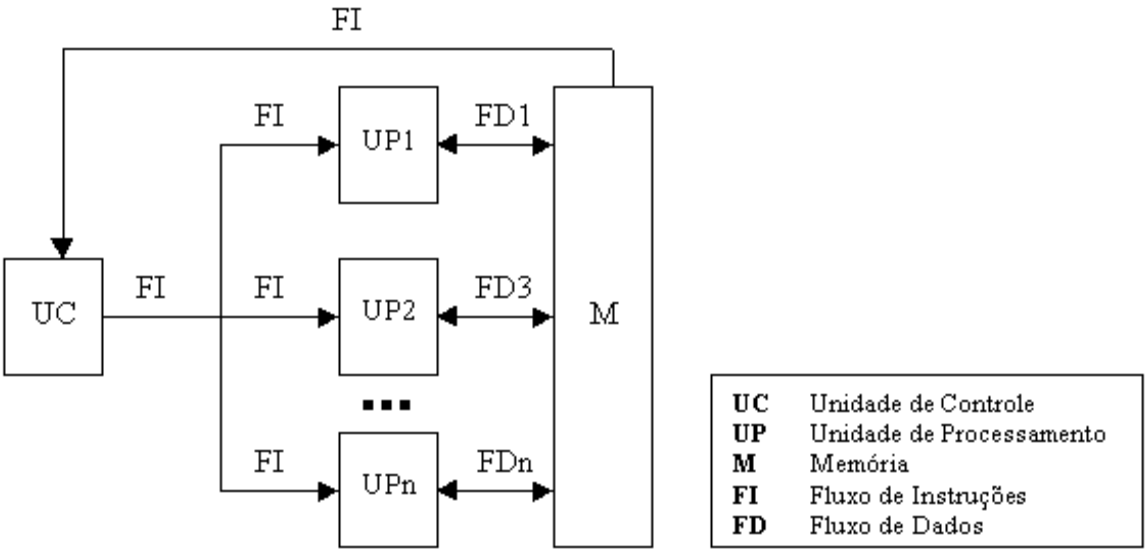

Figura 2: Arquitetura SIMD, segundo *Flynn*

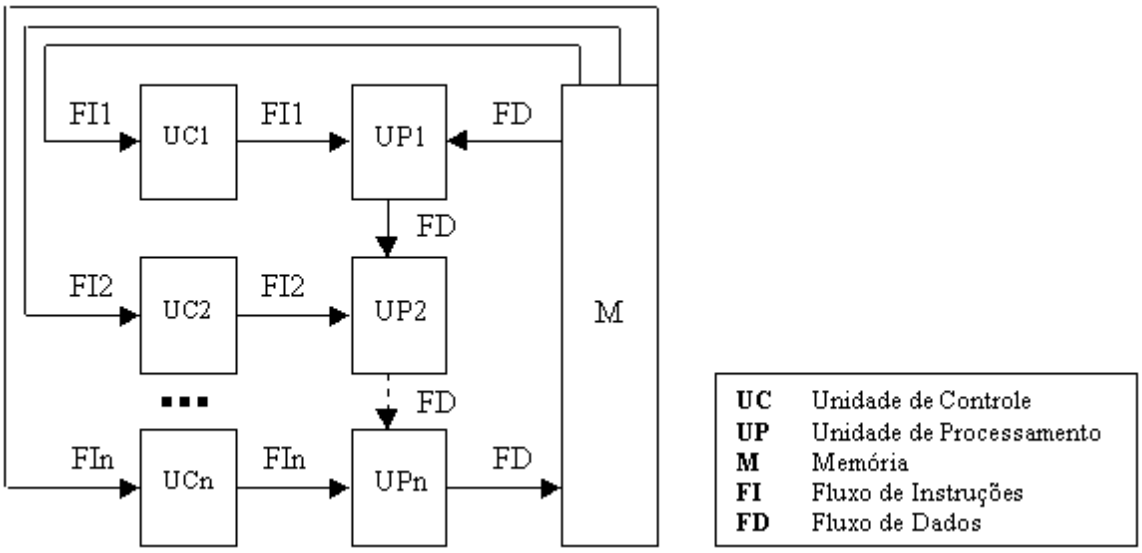

Figura 3: Arquitetura MISD, segundo *Flynn*

MIMD – Multiple Instruction Multiple Data stream: Essa é a categoria mais importante para o presente trabalho. Nela encontramos computadores que executam diversos fluxos de instruções ao mesmo tempo, possuindo diversas unidades de controle. Por existirem múltiplos fluxos tanto de instruções como de dados, elas devem conter também múltiplos processadores (figura 4). Tanto as máquinas denominadas *multiprocessadores* quanto aquelas a que chamamos de *multicomputadores* fazem parte dessa categoria. Tais arquiteturas serão examinadas com maior detalhe posteriormente neste capítulo (seção 2.3).

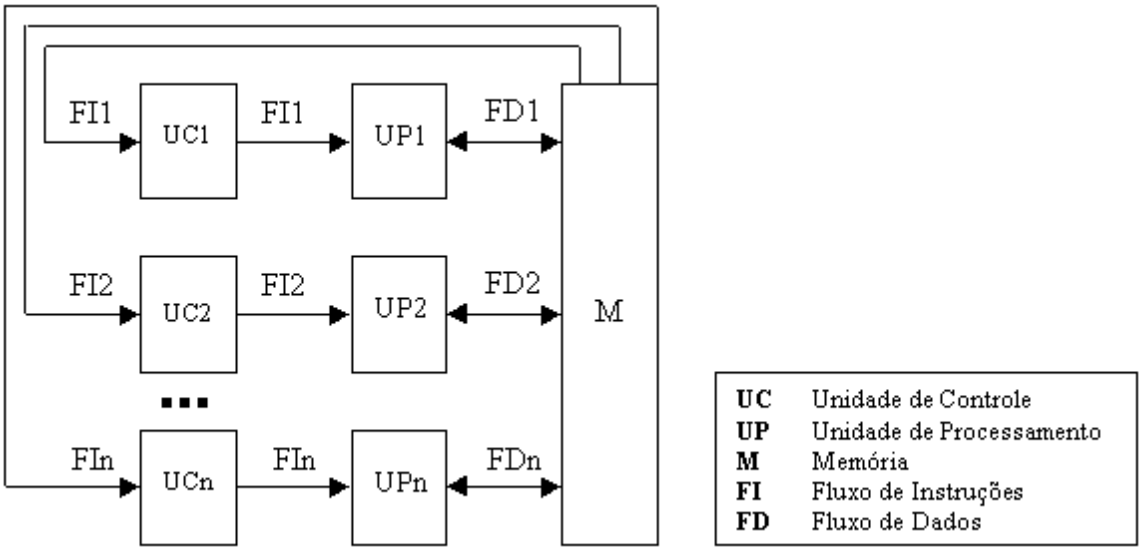

Figura 4: Arquitetura MIMD, segundo *Flynn*

2.2.2 Taxonomia de *Duncan*

A *taxonomia de Duncan* é considerada mais abrangente do que a de *Flynn*, por permitir que arquiteturas diferentes possam ser nela encaixadas. É constituída por três categorias:

Arquiteturas síncronas: Nessa categoria, encontramos máquinas com vários fluxos de instruções operando sobre um ou mais fluxos de dados. A principal característica dessas máquinas é a de que todas as suas operações são baseadas em um relógio único. Os computadores SIMD e MISD, da categoria de *Flynn*, podem se encaixar nessa nova definição, assim como os multiprocessadores MIMD.

Arquiteturas MIMD: Em analogia com a categoria MIMD da taxonomia de *Flynn*, aqui encontramos máquinas que executam diversos fluxos de instruções sobre diver-

sos fluxos de dados. A diferença é que os fluxos de instruções não são executados sincronamente. Assim, as máquinas que se encaixam nessa descrição são os multicomputadores MIMD.

Arquiteturas MIMD estendidas: Se encaixam nessa categoria computadores MIMD com características especiais. Na literatura, encontram-se como exemplos dessa categoria os *computadores sistólicos assíncronos*, as arquiteturas baseadas em fluxos de dados e máquinas MIMD que se comportam, sob certas circunstâncias, como máquinas SIMD.

2.2.3 Acoplamento

Outra forma de classificar as arquiteturas paralelas é levando em conta o conceito de *acoplamento*. O acoplamento, de modo geral, é uma indicação do grau de integração existente entre os componentes de um sistema paralelo. *Tanenbaum* [18] estabelece uma distinção entre o *acoplamento de hardware* e o *acoplamento de software*.

Em termos de *hardware*, o acoplamento de um sistema pode ser relacionado com a velocidade de comunicação entre os processadores que o compõe. Em um multiprocessador, em que todos os processadores residem na mesma máquina e são interligados por um barramento de alta velocidade, o sistema é dito *fortemente acoplado*. Já em uma rede de computadores, em que as diversas máquinas estão ligadas por uma rede comercial de 10 ou 100 Mbps, o sistema é dito *fracamente acoplado*.

Quando o assunto é *software*, o acoplamento de um sistema é uma medida do quão perceptível são os componentes individuais de um sistema ao usuário. Em um *software* fracamente acoplado, como os sistemas operacionais de rede utilizados atualmente, os usuários necessitam, de uma maneira ou outra, identificar a máquina em que residem os recursos que eles gostariam de utilizar; além disso, existe muito pouca cooperação entre as máquinas para, por exemplo, balancear a carga do sistema. Já em um *software* fortemente acoplado, como um verdadeiro sistema operacional distribuído, os usuários não são capazes de identificar individualmente as máquinas que compõem o sistema, não necessitando sequer saber em quais máquinas suas tarefas estão sendo executadas.

Combinando o acoplamento de *hardware* e o de *software*, *Tanenbaum* apresenta três tipos de arquiteturas paralelas:

- **Sistemas operacionais de rede**, que são a combinação de *hardware* e *software* fracamente acoplados;

- **Multicomputadores**, combinando *hardware* com baixo acoplamento e *software* com alto acoplamento;
- **Multiprocessadores**, compostos de *hardware* e *software* fortemente acoplados.

O acoplamento, mais especificamente o de *hardware*, está ligado diretamente a outro conceito importantíssimo: a *granularidade* do *software*. A granularidade indica o tamanho das tarefas sendo executadas em paralelo por uma determinada computação. Quanto menor a granularidade, menor é o tamanho da tarefa, em termos do poder computacional que ela requer. Em um sistema com *paralelismo de grão fino*, as tarefas sendo executadas são pequenas e rápidas, o que exige uma constante interação entre os diversos processos envolvidos na computação. Em um sistema com *paralelismo de grão grosso*, as tarefas sendo executadas requerem maior tempo computacional, fazendo com que a interação entre os processos (a comunicação entre eles) seja menor.

O acoplamento se relaciona com a granularidade no sentido de que quanto maior o acoplamento de um sistema, maior é a taxa de transferência do meio de comunicação que interliga os processadores do sistema; nesse sistema, tanto computações com baixa ou alta granularidade podem ser executadas. Em um sistema com baixo grau de acoplamento, a velocidade do meio de comunicação é menor; isso prejudica a execução de computações com alta granularidade, pois a relação entre o tempo gasto transmitindo mensagens e o tempo efetivamente utilizado para computação pode crescer muito. Esse tipo de computador é mais adequado para a execução de computações de baixa granularidade.

Pode-se facilmente fazer uma correlação entre as categorias apresentadas pela taxonomia de *Flynn* e de *Duncan* com o conceito de acoplamento e, indiretamente, também com a granularidade. Na classificação proposta por *Flynn*, as máquinas SISD, SIMD e MISD são todas fortemente acopladas, e portanto adequadas a qualquer tipo de granularidade. Já na categoria MIMD, encontramos tanto os multiprocessadores, de grande acoplamento, quanto os multicomputadores, com baixo acoplamento. Na taxonomia de *Duncan*, a correlação é mais evidente ainda. A categoria de máquinas síncronas engloba as arquiteturas SISD, SIMD, MISD e multiprocessadores MIMD, e todas elas podem ser consideradas fortemente acopladas. E a categoria MIMD, onde encontramos os multicomputadores, é estritamente composta por máquinas fracamente acopladas.

2.3 Os computadores MIMD

Pela taxonomia de *Flynn*, a mais aceita atualmente, são as arquiteturas denominadas MIMD as de maior interesse para este trabalho. Nela, podemos encontrar dois tipos de máquinas que merecem ser abordados com algum detalhe: os *multiprocessadores* e os *multicomputadores*.

2.3.1 Multiprocessadores

Os *multiprocessadores* são arquiteturas fortemente acopladas, compostas por diversos processadores ligados por um barramento interno de alta velocidade. Operacionalmente, elas são semelhantes a uma máquina não-paralela SISD multiprogramada. Vários processos podem estar sendo executados simultaneamente; a diferença é que a concorrência pelo uso do processador é menor, haja visto que existem diversas *CPU*s disponíveis. Também na aparência externa os multiprocessadores podem se assemelhar a um SISD, pois todas as *CPU*s ficam instaladas em um mesmo equipamento. Olhando uma máquina monoprocessadora SISD e um multiprocessador MIMD como uma caixa-preta, não se é capaz, na maioria dos casos, de diferenciar um do outro.

No funcionamento interno, entretanto, as diferenças começam a aparecer. A principal característica de um multiprocessador (além de possuir diversas *CPU*s) é o fato de que seus processadores *compartilham* memória. É através desse compartilhamento que os processadores, ou melhor, os processos que eles executam, podem se comunicar e coordenar seus esforços. A importância da memória nessa arquitetura também pode ser medida pelo fato de que é através dela que se classificam os diferentes tipos de multiprocessadores. São abordados a seguir as três categorias mais encontradas na literatura.

UMA – Uniform Memory Access Nesse modelo, toda a memória de um multiprocessador é compartilhada pelas *CPU*s nele existentes. Ainda mais, o tempo de acesso a essa memória é o mesmo para todos os processadores. Esse tipo de arquitetura é também referida na literatura como *multiprocessador com memória centralizada*. Uma descrição esquemática dessa arquitetura pode ser observada na figura 5

NUMA – Non-Uniform Memory Access Nesse modelo, cada processador possui sua própria memória, chamada de memória local (veja figura 6). A memória local de um processador é acessível a todos os outros processadores, sendo que o conjunto de todas as memórias locais forma a memória global compartilhada. No entanto, como cada memória local está ligada a um determinado processador, os tempos de

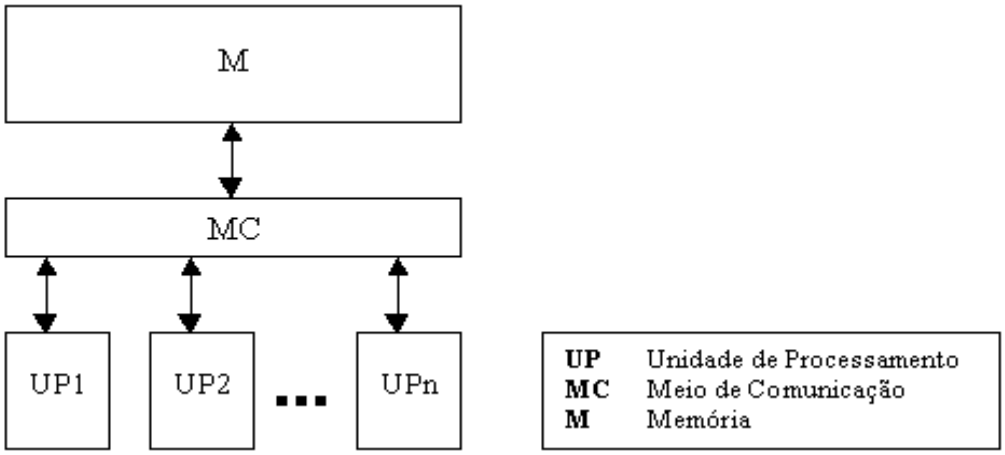

Figura 5: Multiprocessador UMA

acesso à memória variam com sua localização. Se um processador acessa sua memória local, o tempo gasto para essa operação é bem menor do que o tempo utilizado para acessar a memória local de outro processador. Outra denominação para esse tipo de arquitetura é a de *multiprocessador com memória distribuída*.

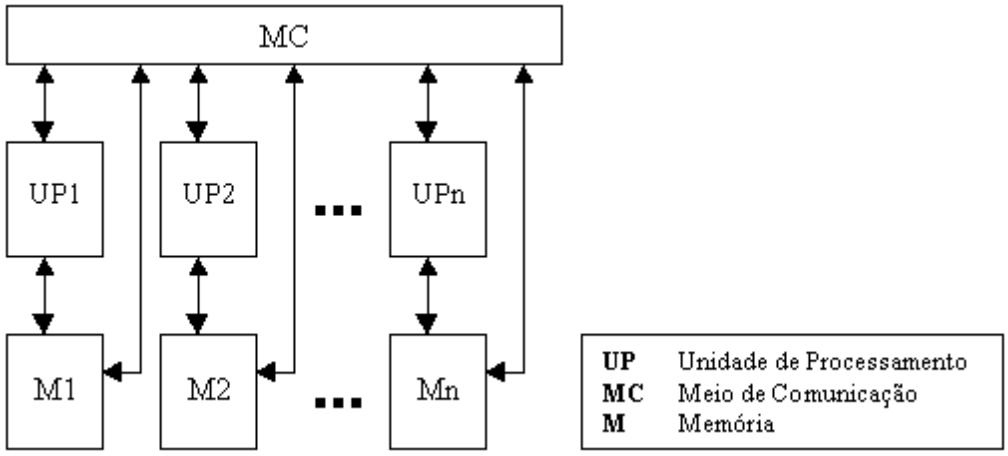

Figura 6: Multiprocessador NUMA

COMA – Cache-Only Memory Access Esse modelo pode ser considerado um caso especial do modelo NUMA. Nessa arquitetura, nota-se a introdução de uma memória *cache* ligado a cada processador (figura 7). Essa *cache* tem o intuito de minimizar o custo do acesso à memória local de outros processadores. Surge então o problema de gerenciar as alterações da *cache*. Sempre que um processador alterar o conteúdo de uma posição da sua memória local, todas as *caches* remotas que estão mapeando essa mesma posição devem remover esse mapeamento ou atualizar seu conteúdo.

Uma quarta classificação que pode ser encontrada na literatura é denominada *NORMA*

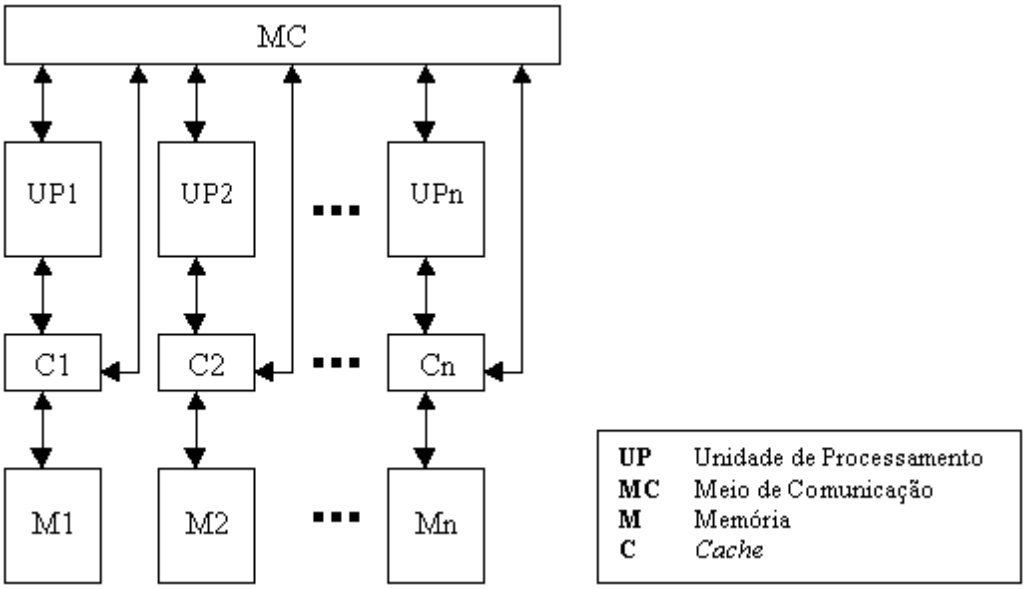

Figura 7: Multiprocessador COMA

(No Remote Memory Access). Ela descreve arquiteturas cujos processadores possuem memória local mas nenhum deles tem acesso à memória de outra *CPU* (figura 8). Ou seja, não existe compartilhamento de memória. Assim, na verdade, em tal classificação se encaixam perfeitamente os multicomputadores, objeto de estudo da seção 2.3.2.

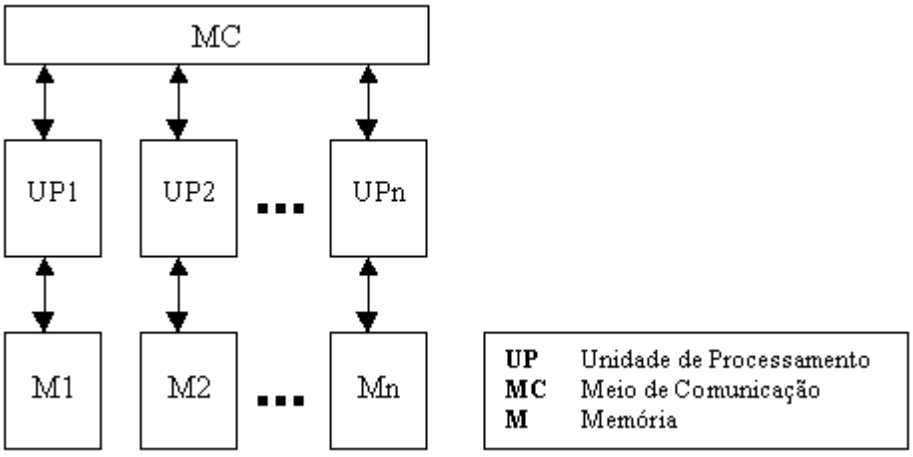

Figura 8: Multicomputador NORMA

Os multiprocessadores, por serem uma arquitetura fortemente acoplada, são adequados a computações de granularidade alta ou baixa. No entanto, esse mesmo acoplamento que parece uma grande vantagem traz limitações à utilização dessa arquitetura. Isso porque as *CPU*s de um multiprocessador dependem, para sua comunicação, de um meio de interconexão (normalmente, um barramento). Esse é um recurso que pode se tornar escasso com a adição de muitos processadores a uma mesma máquina. Uma arquitetura

desse tipo pode possuir, portanto, um baixo grau de escalabilidade.

2.3.2 Multicomputadores

Os *multicomputadores* são máquinas MIMD fracamente acopladas. São sistemas compostos por diversas máquinas independentes interligadas por um meio de comunicação que permite a elas interagir e cooperar. Em sua aparência externa, eles não se confundem com computadores não-paralelos SISD, pelo fato de que, mesmo através de um exame superficial, são facilmente perceptíveis as várias unidades independentes que compõem o sistema. No tocante ao modo de operação, no entanto, as máquinas MIMD tentam ser o mais semelhante possível a máquinas SISD multiprogramadas.

A característica mais notável dos multicomputadores é a ausência de memória compartilhada (o que os encaixa na categoria de máquinas NORMA discutida na seção 2.3.1). A interação entre as máquinas componentes do sistema deve ser efetuada através de *troca de mensagens*. Essas mensagens são enviadas pelo meio de comunicação, o que o torna o elemento fundamental desse tipo de arquitetura. Pode-se dividir as redes de interconexão utilizadas pelos multicomputadores em dois grandes grupos: *redes estáticas* e *redes dinâmicas*.

2.3.2.1 Redes estáticas

O meio de comunicação utilizado pelos multicomputadores, tanto nos que usam redes estáticas quanto naqueles que usam redes dinâmicas, consiste de canais de comunicação bidirecionais interligando os nós que compõem o sistema. Nas redes estáticas, entretanto, certas características da rede *não* podem ser alteradas durante a operação do sistema – o número de canais e os nós que eles conectam.

Desse modo, um determinado nó do sistema somente pode estabelecer comunicações diretas com um número predeterminado de outros nós. Mais ainda, os nós com os quais ele pode se comunicar são fisicamente determinados pela distribuição dos canais de comunicação. O número máximo de conexões de um nó deve ser levado em conta ao se construírem os programas que serão executados pelo multicomputador. Uma aplicação só conseguirá executar em um determinado multicomputador se a *rede lógica* formada pelas conexões entre seus processos puder ser mapeada para a *rede física* formada pela rede de conexões entre seus nós.

Um nó somente pode ultrapassar seu limite de conexões (e assim se conectar a outros nós que não são seus vizinhos diretos) se o sistema operacional do multicomputador for

capaz de efetuar o *roteamento* de mensagens. Nessa operação, basicamente, cada nó irá passar adiante uma mensagem não endereçada a ele, até que ela chegue ao seu destinatário. No entanto, a capacidade de rotear mensagens adiciona complexidade ao sistema operacional e diminui a eficiência da comunicação, razões suficientes para se tentar evitar sua utilização quando possível.

A seguir, serão apresentadas algumas das topologias de redes estáticas mais encontradas na literatura. Estão presentes breves considerações sobre suas características, como *grau* (número de canais de comunicação) dos nós e *diâmetro* (distância entre os nós mais distantes) da rede.

Completamente conectada Esse é o modo mais eficiente de se interconectar os nós de uma rede; é também o mais caro. Nessa topologia, cada nó possui um canal de comunicação que o conecta a todos os outros nós da rede (figura 9). Uma rede completamente conectada de ordem 4 pode ser observada na figura. O número de canais de comunicação necessários para uma rede de ordem n é dado pela fórmula $n(n\text{-}1)/2$; o crescimento do número de canais acompanha portanto o quadrado do número de nós. Em cada ponta do canal pode ser necessária, dependendo da tecnologia utilizada, uma interface de comunicação;, o número total de interfaces necessárias é, portanto, igual ao dobro de canais, donde a fórmula $n(n\text{-}1)$.

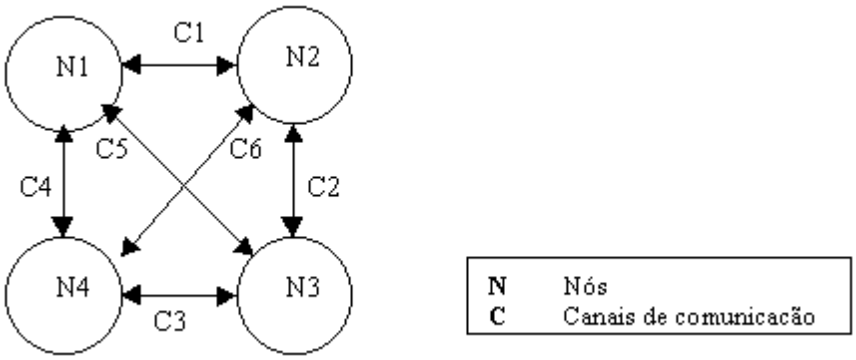

Figura 9: Rede estática completamente conectada

Árvore binária A principal característica desse tipo de topologia é a ausência de ciclos, o que aumenta o diâmetro da rede e pode tornar custosas comunicações que exijam roteamento. O grau dos nós é variável; uma árvore binária (figura 10), por exemplo, possui um nó raiz com grau 2, e todos os outros com graus entre 3 e 1. Nessa categoria se encaixam também redes com topologias de árvores não binárias, irregulares, não balanceadas, entre outras. O número de canais necessários é dado

pela fórmula n-1, ou seja, o número de canais é igual ao número de nós menos 1; o crescimento do número de canais necessários em uma rede do tipo árvore binária, quando comparada a uma rede de topologia completamente conectada, tem um ritmo bem menor, associado linearmente ao número de nós.

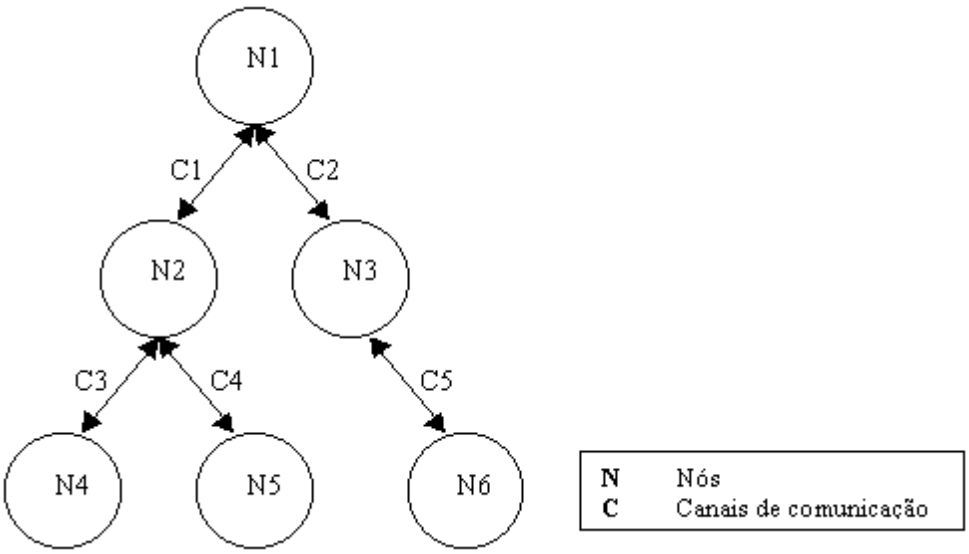

Figura 10: Rede estática em forma de árvore binária

Anel A existência de um ciclo diminui o diâmetro da rede e melhora os custos de comunicação. O grau dos nós, no entanto, é pequeno – todos eles possuem grau 2 (veja figura 11).

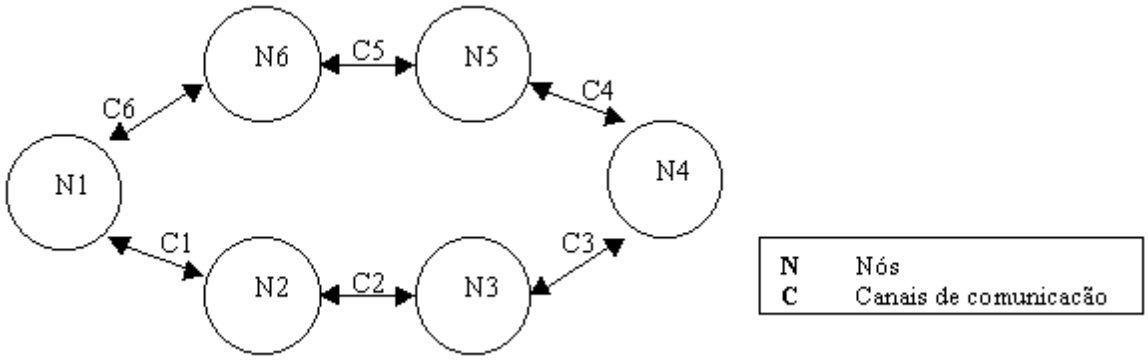

Figura 11: Rede estática em forma de anel

Grelha A grelha é uma das configurações mais conhecidas e utilizadas devido à sua versatilidade e à sua vantajosa relação custo–benefício. Nessa topologia, temos três tipos diferentes de nós, em ordem decrescente de grau: os nós centrais, os nós laterais, e os vértices. Em uma grelha bidimensional podemos encontrar nós centrais

de grau 4, laterais de grau 3 e vértices com grau 2. Supondo que essa mesma grelha tivesse *ordem* 4 (um quadrilátero de lado 4, contendo, portanto, 16 nós ao todo), teríamos um diâmetro de rede igual a $2(y\text{-}1) = 2(4\text{-}1) = 6$, onde y é a ordem da rede. Uma grelha de ordem 3 pode ser observada na figura 12.

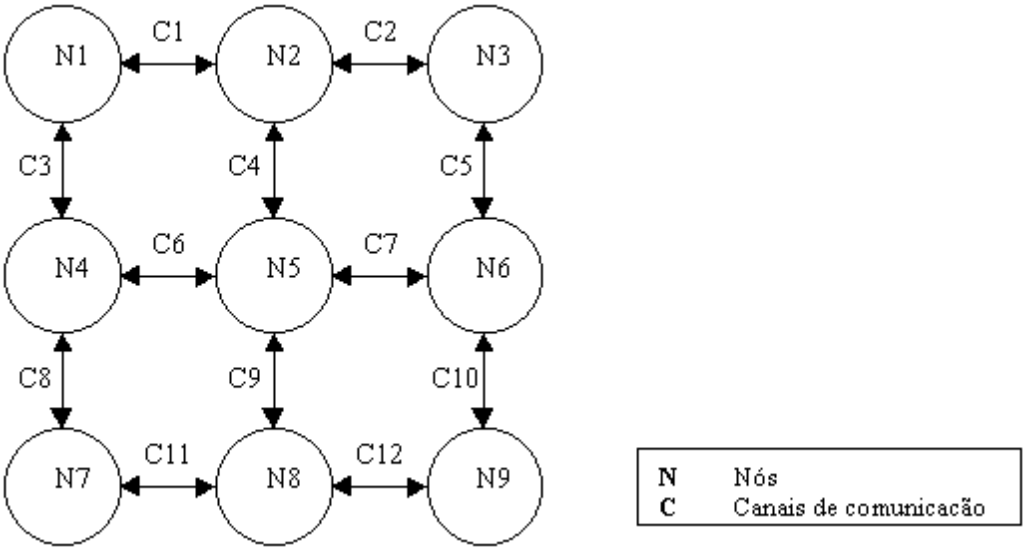

Figura 12: Rede estática em forma de grelha de ordem 3

Torus A rede toroidal é uma variação da grelha. Ela é obtida ao se interligar os nós laterais de cada linha e coluna, assim como os vértices adjacentes. Consegue-se assim uma rede com grau de nó uniforme (igual a 4 para um *torus* bidimensional, por exemplo). O diâmetro da rede também cai, tornado-se igual à sua ordem. Já o custo dessa topologia é maior, visto que, para um *torus* de ordem 4, temos $2y = 2\text{x}4 = 8$ canais de comunicação adicionais em comparação com uma grelha de mesma ordem. Um torus de ordem 3 é descrito esquematicamente na figura 13.

Hipercubo No hipercubo, todos os nós possuem um número uniforme de conexões – em uma rede de ordem 3 (figura 14), todos os nós tem grau 3. Um hipercubo dessa ordem é composto por 8 nós interligados de tal modo que a topologia física da rede se pareça com um cubo. Em um hipercubo de ordem 4, temos 16 nós formando dois cubos distintos, sendo que cada vértice de um cubo deve estar ligado ao seu vértice correspondente no outro cubo. E em um hipercubo de ordem 5, temos 4 cubos (32 nós) com seus vértices interligados.

CCC - Cubo Conectado por Ciclos A topologia CCC é uma variação do hipercubo. Ela é conseguida através da substituição de cada vértice do hipercubo por um ciclo formado por n nós, onde n é igual à ordem da rede. Uma CCC de ordem 3 (figura 15)

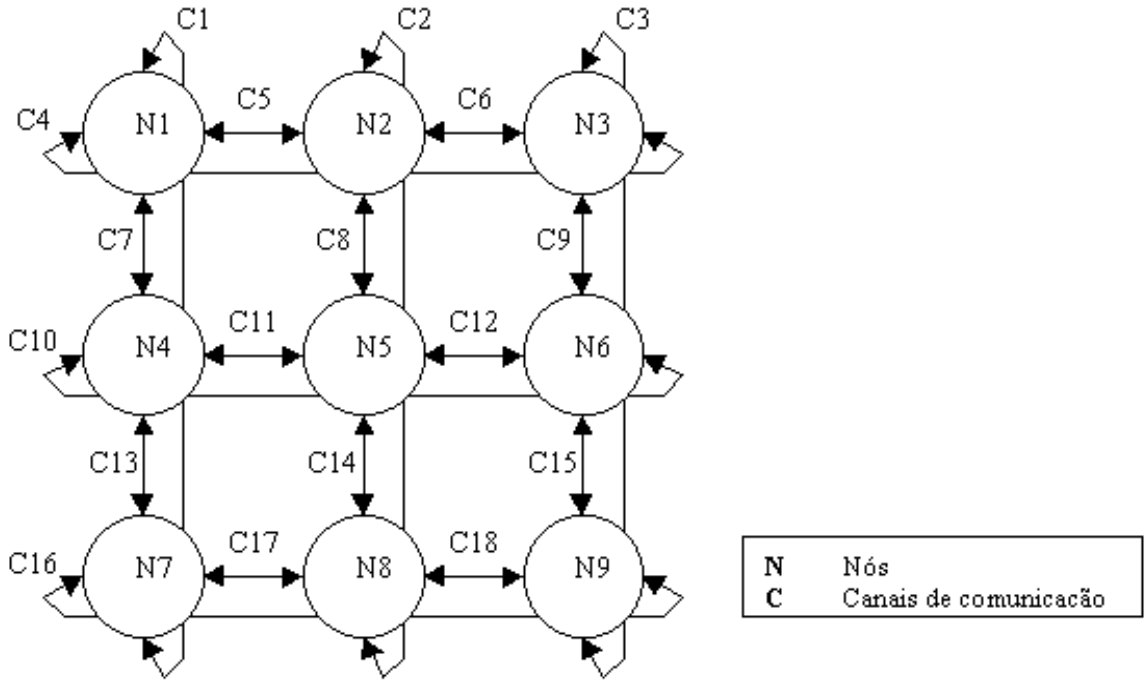

Figura 13: Rede estática em formato toroidal de ordem 3

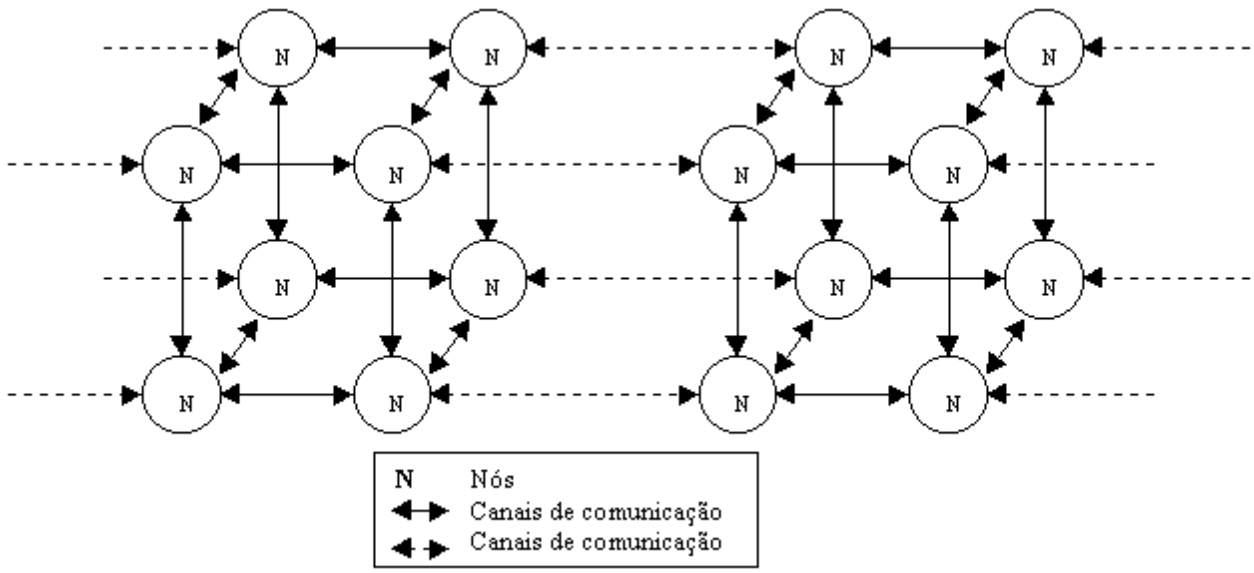

Figura 14: Rede estática em forma de hipercubo de ordem 3

possui, em cada vértice, 3 nós formando um anel. Possui, portanto, 8 anéis e $8.3 = 24$ nós no total.

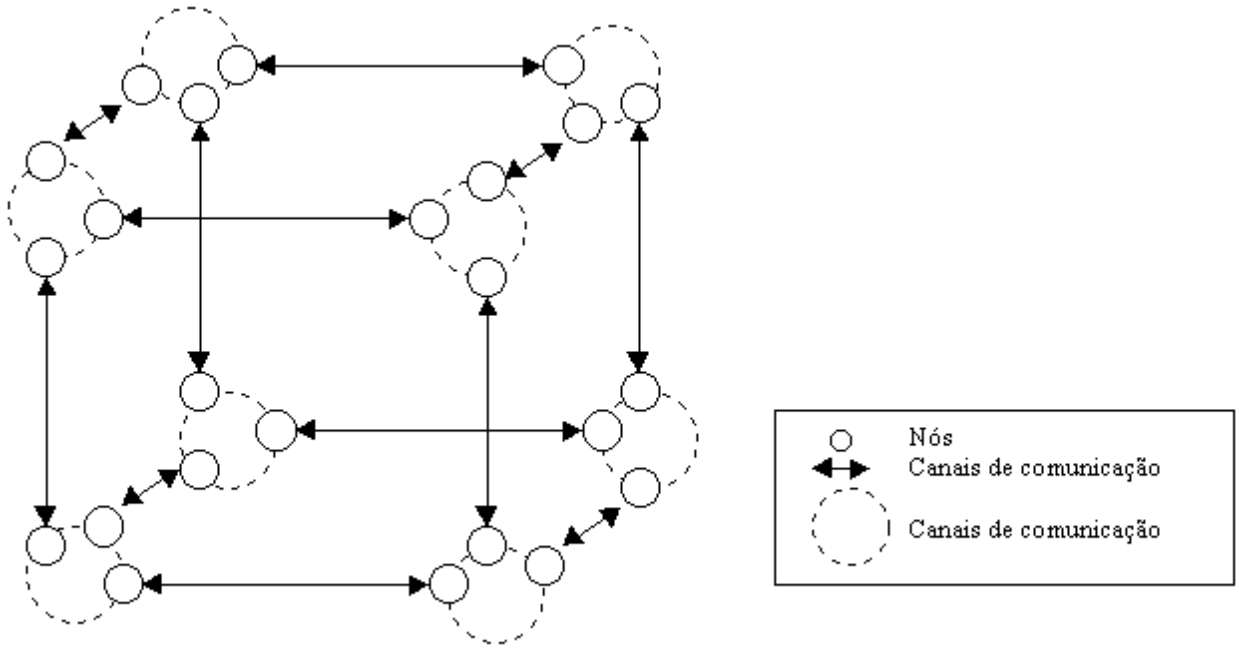

Figura 15: Rede estática com topologia CCC de ordem 3

2.3.2.2 Redes dinâmicas

As redes dinâmicas tem, como característica fundamental, a possibilidade de alterar suas conexões sem necessidade de intervenção física (por exemplo, sem necessidade da reconfiguração do cabeamento que interliga as máquinas da rede). Assim, um nó pode utilizar o mesmo meio de comunicação para se comunicar com vários outros nós.

A utilização de uma rede dinâmica não garante, por si só, a conectividade total simultânea entre os nós. Isso porque redes dinâmicas costumam ter preço elevado, o que pode determinar a existência de restrições . Isso traz de volta a questão do limite do número máximo de conexões de um nó. Ela deve ser vista, no entanto, sob uma ótica ligeiramente diferente. Em uma rede estática, um determinado nó de grau 3 pode se comunicar apenas com os 3 outros nós que estejam fisicamente ligados a ele. Em uma rede dinâmica, um nó de grau 3 pode se comunicar, simultaneamente, com no máximo 3 outros nós; esses nós, no entanto, podem ser diferentes de um momento para outro. Com isso, o roteamento de mensagens raramente é necessário, pois normalmente é possível estabelecer uma conexão direta entre os nós que desejam se comunicar.

Existem, é claro, algumas restrições. Pode ocorrer que, em um determinado momento, todos os canais de comunicação de um nó estejam ocupados, e, assim, será impossível estabelecer uma conexão com nós adicionais. Como solução, pode-se empregar técnicas capazes de tornar possível a transmissão de várias mensagens simultâneas por um mesmo meio (aceitando-se o aumento dos custos da rede e da complexidade do *software*). Uma delas é a utilização de freqüências diferentes para a implementação de canais de comunicação distintos; outra técnica é a multiplexação de mensagens no tempo através da utilização de pacotes de dados.

Existem vários tipos de redes dinâmicas, e alguns dos mais comuns na literatura são apresentados a seguir.

Crossbar Através de um *crossbar* todo nó pode estabelecer dinamicamente uma comunicação direta com qualquer outro nó. É uma rede que apresenta, portanto, conectividade total (figura 16). Mas possui a restrição já comentada de que um nó só pode estar envolvido em uma comunicação de cada vez. Assim, um *crossbar* 10x10, capaz de conectar entre si 10 máquinas de uma rede, possui no máximo, em um dado momento, 10 conexões simultâneas.

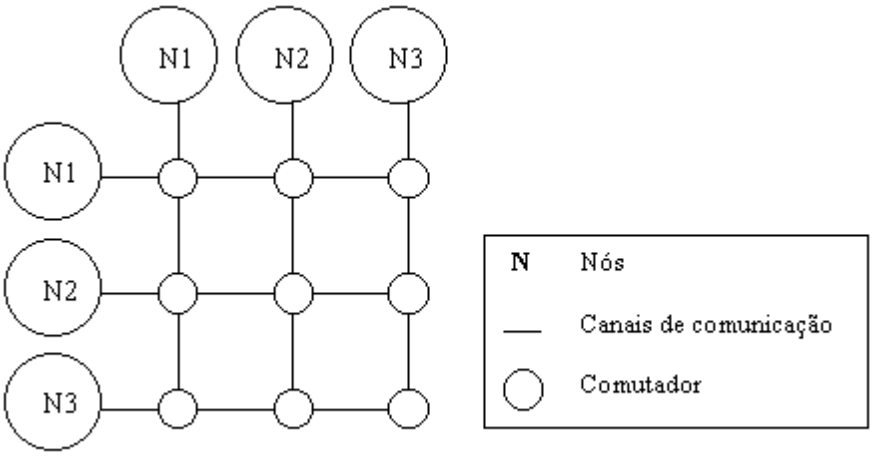

Figura 16: Rede dinâmica do tipo *crossbar*

Multiestágio A conectividade total oferecida pelos *crossbars* pode ter um custo elevado, o que motiva a existência de alternativas. Uma delas é a rede multiestágio, formada por um conjunto de *crossbars* de porte menor do que o de um único *crossbar* capaz de atender à rede toda (figura 17). Uma rede com 8 nós necessitaria de um *crossbar* 8x8 (tendo portanto 8.8 = 64 pontos de comutação). Essa mesma rede pode ser modelada através de 12 *crossbars* 2x2 divididos em três estágios de comutação. Cada

um desses *crossbars* possui 2.2 = 4 pontos de comutação, tendo a rede, portanto, um total de 4.12 = 48 pontos de comutação. A rede multiestágio tem, então, um custo menor do que uma rede *crossbar*.

Outra motivação para a construção de uma rede multiestágio pode ser a impossibilidade de se conseguir um *crossbar* adequado. Para isso, basta que se tente construir uma rede com um número de nós superior ao número de entradas do maior *crossbar* existente no mercado. Nesse caso, ou se constrói um *crossbar* que atenda às exigências da rede, ou utiliza-se uma rede multiestágio.

As redes multiestágio, claro, possuem desvantagens, qual seja a de serem *bloqueantes*: o estabelecimento de uma única conexão entre dois nós impossibilita a ocorrência de diversas outras conexões. Impede até mesmo conexões envolvendo outros nós inteiramente distintos. Na rede 8x8 já exemplificada, para cada nível adicional de comutação além do primeiro, temos 4 conexões bloqueadas. Como temos 2 níveis adicionais (o segundo e o terceiro), temos 2x4 = 8 conexões bloqueadas a cada conexão ativa. Vale lembrar que essas 8 conexões podem ser excludentes entre si: várias delas podem ter os mesmos nós de origem ou destino, o que as impediria de existir simultaneamente mesmo em uma rede *crossbar*.

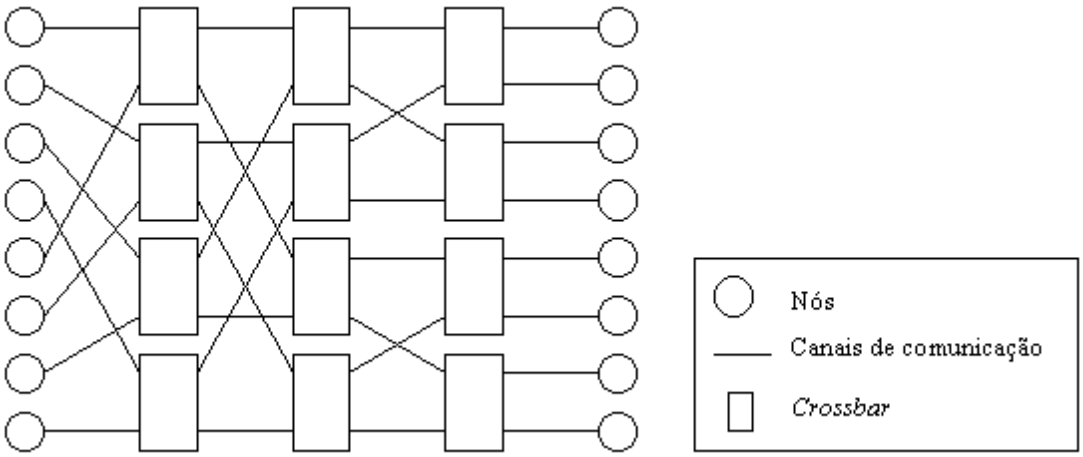

Figura 17: Rede dinâmica do tipo multiestágio

Barramento O barramento é considerado uma rede dinâmica porque, a cada momento, um mesmo canal de comunicação pode ser utilizado para conectar nós diferentes. No barramento, ao realizar uma transmissão, um nó coloca a mensagem no meio de comunicação. Esse meio é compartilhado, e todos os outros nós o monitoram e verificam a quem se destina a mensagem (fugura 18). Apenas um deles vai reconhecê-la e processá-la. Esse tipo de rede possui, portanto, o inconveniente de que apenas

uma mensagem pode ser enviada em um dado momento. Quanto mais máquinas estiverem ligadas ao barramento, maior será a disputa entre elas pela utilização do meio, e menor será a taxa de transmissão devido à alta incidência de colisões. Conseqüentemente, redes baseadas em barramento não possuem a propriedade da escalabilidade. Sendo, porém, as redes de menor custo de todas as apresentadas neste trabalho, não podem nunca deixar de ser uma opção.

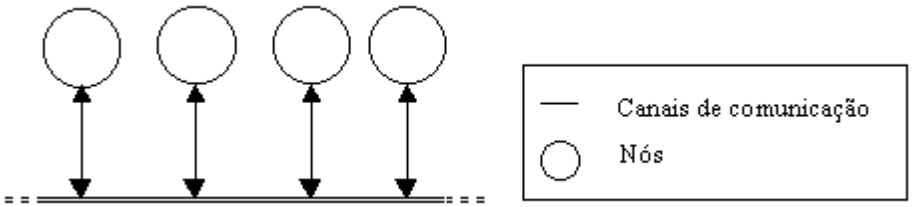

Figura 18: Rede dinâmica utilizando barramento

2.3.3 Clusters

Os *clusters* (ou aglomerados) de computadores são, por assim dizer, os modernos multicomputadores. A eles se aplicam os mesmos princípios gerais já discutidos na seção 2.3.2. Discute-se agora apenas as suas características mais marcantes, capazes de diferenciá-los da categoria dos multicomputadores.

A evolução da tecnologia foi tornando obsoletos os multicomputadores de anos atrás e acabou por retirar muitos de seus fabricantes do mercado. Ao mesmo tempo, tornou disponível, a preços relativamente baixos, máquinas poderosas e ferramentas de comunicação de grande desempenho. Os *clusters* se baseiam na idéia de aproveitar todos esses equipamentos comerciais facilmente encontrados no mercado e reuni-los para formar supercomputadores de baixo custo.

Existem inúmeras outras vantagens nesse tipo de abordagem, além do baixo custo já mencionado. As peças de reposição são facilmente encontradas no mercado, e são fabricadas por diversas empresas. Desse modo, o usuário dessas máquinas nunca fica preso a um único fornecedor, sendo que a concorrência garante preços menores. Existe também muito *software* para essas máquinas, boa parte dele gratuito. Além disso, os usuários já estão acostumados a trabalhar com tais equipamentos e *softwares*; é uma bagagem de conhecimento e experiência que não se deve menosprezar.

Os *clusters*, por serem baseados nas tecnologias de redes utilizadas comercialmente, podem ser ambientes bastante heterogêneos. Nos últimos anos, passou a existir uma

grande interoperabilidade entre equipamentos e sistemas operacionais de diferentes fabricantes. Através da adoção de padrões (protocolos, sistemas de arquivo, etc.), eles tornaram-se capazes de se comunicar e, assim, cooperar, trabalhando em rede. Os *clusters* apenas levam essa idéia adiante, adicionando às redes de computadores um sistema operacional fortemente acoplado capaz de tirar o maior proveito possível do potencial dessas máquinas.

Por serem uma tecnologia relativamente recente, a literatura tradicional não costuma abordar os *clusters* com detalhe. Na *internet*, entretanto, uma rápida busca é capaz de encontrar vários projetos nessa área. Do pioneiro projeto *Beowulf* [1] aos *clusters GAMMA* [9] e *FNN* [22], existe muito material disponível. Também este trabalho, ou melhor, o projeto no qual ele se insere, tem como objetivo a construção de um *cluster* de computadores. Nos capítulos posteriores, a arquitetura do *cluster Crux* será detalhada, assim como as primitivas de comunicação do núcleo de seu sistema operacional.

2.4 Problemas

Por tudo que foi visto no decorrer deste capítulo, pode-se pensar que as arquiteturas paralelas são uma panacéia para a computação. No entanto, no dia-a-dia, é raro deparar-se com uma arquitetura paralela de alto desempenho, algo que não seja uma simples rede de computadores utilizada para compartilhamento de recursos e comunicação. Isso deve-se ao fato de que o uso de arquiteturas paralelas traz consigo alguns problemas de certa complexidade.

Uma das dificuldades é a de que certos problemas simplesmente não são adequados ao processamento paralelo. Não se consegue dividi-los em subproblemas que possam ser resolvidos paralelamente. Aplicações que possuam alto grau de interação com o usuário podem sofrer de tal problema.

Mas a maior dificuldade está no fato de que os *softwares* existentes não podem tirar automaticamente proveito da execução em um ambiente paralelo. Todos eles devem passar por adaptações em sua estrutura, processo esse que pode ser complexo, demorado e, conseqüentemente, caro.

É no momento da adoção a nível comercial de uma tecnologia que ela realmente demonstra todo seu valor, influenciando e melhorando a vida de um número cada vez maior de seus usuários. E são as dificuldades que acabaram de ser expostas que dificultam a utilização comercial, em larga escala, de arquiteturas paralelas. Temos um círculo vicioso que pode ser descrito do seguinte modo: sem *software* adequado, diminui o estímulo para

a construção de arquiteturas paralelas; e com a escassez de tais arquiteturas, não existe mercado para o desenvolvimento do seu *software*. Até o momento, a pesquisa nessa área continua naquele estágio em que é quase totalmente realizada pelo meio acadêmico (como acontece com quase toda nova tecnologia). Certamente chegará o momento em que, com o avanço das pesquisas, o mercado demostrará maior interesse, e teremos o início de um círculo virtuoso: a existência de cada vez mais *software* adequado estimulará a pesquisa na área de arquiteturas paralelas; e haverão cada vez mais arquiteturas paralelas disponíveis, o que estimulará o investimento em *software*.

Podemos encontrar outros pontos que merecem destaque na literatura. Por exemplo, os aspectos que *Tanenbaum* [18] considera mais problemáticos na utilização de arquiteturas paralelas são *software*, comunicação e segurança.

2.5 Resumo

Este capítulo apresentou um breve resumo de temas relacionados à área de arquiteturas paralelas. As máquinas paralelas foram definidas como aquelas que possuem diversas unidades de processamento e um ou mais fluxos de instruções. A motivação para a construção de tais máquinas é tanto a busca por desempenhos cada vez maiores como também a economia.

Foram apresentadas também diversas maneiras de classificar as máquinas paralelas, como as taxonomias de Flynn e de Duncan, bem como quanto ao acoplamento de seus componentes. Uma correlação simplificada entre esses três tipos de classificação foi estabelecida.

Os multiprocessadores e os multicomputadores, algumas das arquiteturas paralelas mais conhecidas e utilizadas, foram apresentadas com maior profundidade. Os multiprocessadores foram definidos como máquinas MIMD fortemente acopladas e classificados de acordo com seu acesso à memória; os multiprocessadores foram definidos como máquinas MIMD fracamente acopladas e classificados quanto ao tipo de rede que utilizam (estática ou dinâmica) e sua topologia. Os *clusters*, um subconjunto dos multicomputadores, foram definidos como aquelas máquinas paralelas que se utilizam de componentes comercialmente disponíveis em sua constituição, alcançando assim alto desempenho a custos reduzidos.

Finalmente, alguns dos problemas inerentes à utilização de arquiteturas paralelas foram comentados. Figuram entre os principais, o problema da adequação das tarefas ao paralelismo e a falta de *software* capaz de executar em tais arquiteturas.

3 *A arquitetura do* cluster *de computadores* Crux

Este capítulo dedica-se a apresentar a arquitetura do *cluster* de computadores *Crux*. A idéia principal desta arquitetura é a simplicidade. Através de uma estreita cooperação entre *hardware* e *software*, ela é capaz de solucionar de forma elegante e eficiente diversos problemas que afligem outros tipos de *clusters*.

Ao aplicarmos, na arquitetura *Crux* aqui apresentada, as classificações estudadas no primeiro capítulo deste trabalho, teremos o seguinte resultado: a taxonomia de *Duncan* a classificaria como uma máquina *MIMD*, portanto assíncrona no sentido da inexistência de uma única referência temporal para todos os processadores do sistema. A taxonomia de *Flynn* também classificaria o *Crux* como uma máquina *MIMD*. Aprofundando essa classificação, percebe-se que o *Crux* é um multicomputador, mais precisamente um *cluster* de computadores; a rede utilizada é dinâmica, do tipo *crossbar*, contando ainda com a ajuda de um barramento de controle. Se levarmos em conta o acoplamento, temos uma máquina com *hardware* fracamente acoplado executando *software* fortemente acoplado.

São expostos a seguir os componentes físicos da arquitetura e o sistema operacional que a controla. Também são discutidas com detalhes as características que tornam o *Crux* uma arquitetura inovadora e de alto desempenho. A principal fonte das seguintes informações é *Corso* [8], mas, a não ser onde expressamente indicado, pode-se considerar que o presente capítulo é um apanhado da seguinte bibliografia: Bogo [2], Boing [3], Campos [4], Cancian [5], Gavilan [10], Merkle [12], Montez [13], Plentz [14], Rech [15], Silva [17] e Zeferino [21].

3.1 Componentes do sistema

Nesta seção, são apresentados os componentes físicos de uma máquina *Crux*, ou seja, o *hardware* que a constitui. As máquinas que compõem o *cluster* de computadores podem ser classificadas em duas categorias: *nós de trabalho* e *nó de controle*. Existem ainda dois meios físicos distintos interligando essas máquinas: a *rede de controle* e a *rede de trabalho*.

Cada um desses elementos, que podem ser observado na figura 19 [8], é descrito a seguir.

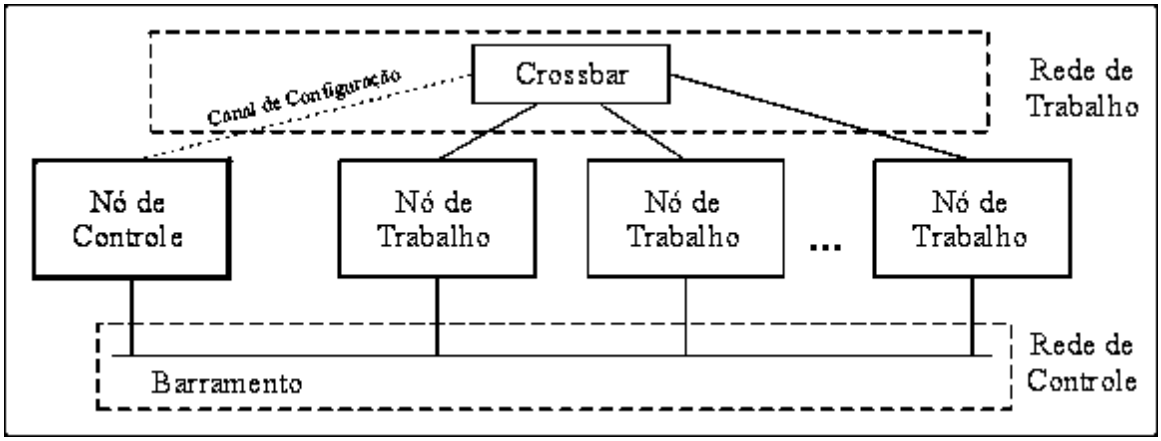

Figura 19: Arquitetura do *Crux*

3.1.1 Nós de trabalho

O nós de trabalho constituem a maioria absoluta das máquinas existentes no *cluster*. Eles são computadores completos, como os encontrados no comércio em geral (afinal, trata-se de um *cluster*). Possuem, é claro, algumas pequenas diferenças, sendo uma delas a ausência quase completa de periféricos: um nó de trabalho não necessita de dispositivos de entrada e saída, como teclado, *mouse*, monitor ou impressora. Ele pode nem mesmo possuir, na maioria das vezes, unidades de armazenamento. Existem, no entanto, nós de trabalho com funções específicas dentro do *cluster*. São aqueles que atuam como *servidores*. Um nó com a função de servidor de arquivos, por exemplo, certamente necessitaria de unidades de armazenamento.

Outra diferença entre um nó de trabalho *Crux* e um computador comum, do tipo *desktop*, é a existência de várias interfaces de rede. Elas conectam-no a duas redes distintas: existe uma interface ligando-o à rede de controle, e uma ou mais que o conectam à rede de trabalho. Tais redes são abordadas em detalhes adiante.

A função dos nós de trabalho do *cluster Crux*, como não poderia deixar de ser, é executar o trabalho pesado. São eles que, cooperativamente, realizam as tarefas passadas para o *cluster*. A coordenação entre eles é intermediada pelo nó de controle, que será visto a seguir.

3.1.2 Nó de controle

O nó de controle é o nó responsável por executar o *servidor de comunicações*. Basicamente, tal servidor é responsável por coordenar as comunicações entre os nós de trabalho do sistema, fazendo com que todas as trocas de mensagens entre eles sejam *diretas*. A comunicação direta, no *Crux*, tem em um sentido mais amplo – ela identifica a comunicação direta não simplesmente entre duas máquinas, mas entre as áreas de memória de dois processos distintos. As vantagens trazidas por essa característica da arquitetura serão explicadas com maiores detalhes posteriormente, ainda neste capítulo.

O nó de controle possui no máximo três interfaces de comunicação. Através de uma delas, sempre presente, ele se conecta à rede de controle. A segunda está presente quando se decide conectar o nó de controle a uma *máquina hospedeira*. A máquina hospedeira possui a tarefa de interagir com o usuário, aliviando o nó de controle de tal tarefa. Quanto tal divisão existe, o nó de controle não necessita de periféricos de interação com o usuário, estando tais periféricos situados unicamente na máquina hospedeira. A terceira interface de comunicação está presente, quando necessário, para a configuração dinâmica do elemento comutador da rede de trabalho.

3.1.3 Rede de trabalho

Todo multicomputador de alto desempenho possui um meio físico de grande velocidade interligando suas máquinas: é esse o papel da rede de trabalho no *Crux*. Ela interliga todos os nós de trabalho do *cluster*, e eles a utilizam para trocar mensagens entre si, enquanto colaboram na resolução de tarefas.

Como já mencionado, o *Crux* utiliza uma rede dinâmica de interconexões. Isso significa que a rede possui um elemento ativo passível de controle, ou seja, o *crossbar*. É ele que, obedecendo aos comandos do nó de controle, vai reconfigurar dinamicamente a topologia da rede, permitindo que os nós de trabalho se comuniquem de acordo com suas necessidades. É através dessa reconfiguração por demanda que o roteamento de mensagens pode ser evitado. As vantagens que a ausência de roteamento acarreta são discutidas em detalhes, ainda neste capítulo.

3.1.4 Rede de controle

A rede de controle interliga os nós de trabalho ao nó de controle. Por ela trafegam mensagens de controle do sistema operacional, todas elas de pequeno tamanho (alguns

poucos *bytes*). Ela não necessita, portanto, ser uma rede de alto desempenho. Um barramento comercial de 100 Mbps é perfeitamente adequado. As mensagens que trafegam por essa rede são descritas a seguir, quando se aborda o sistema operacional *ACrux* e suas primitivas.

3.2 O sistema operacional

Na seção anterior foram descritos os componentes que integram um cluster *Crux*. Mas simplesmente interligar fisicamente tais componentes não é suficiente. É necessário que o sistema operacional executado por cada uma das máquinas pertencentes ao *cluster* saiba tirar proveito das características especiais desse tipo de arquitetura. É portanto necessário a criação de um sistema operacional específico para a arquitetura *Crux*.

A seguir, são apresentados os módulos de *software* que, juntos, implementam o sistema operacional *ACrux*. Ele é um sistema operacional distribuído, baseado no modelo cliente-servidor, e estruturado hierarquicamente. Sendo assim, ele é composto pelo núcleo básico, existente em cada um dos nós do sistema, e também por servidores responsáveis por fornecer serviços aos nós de trabalho. A seguir, são descritas as camadas do sistema operacional, assim como é detalhado também o *servidor de comunicações* executado pelo nó de controle, elemento fundamental do sistema operacional.

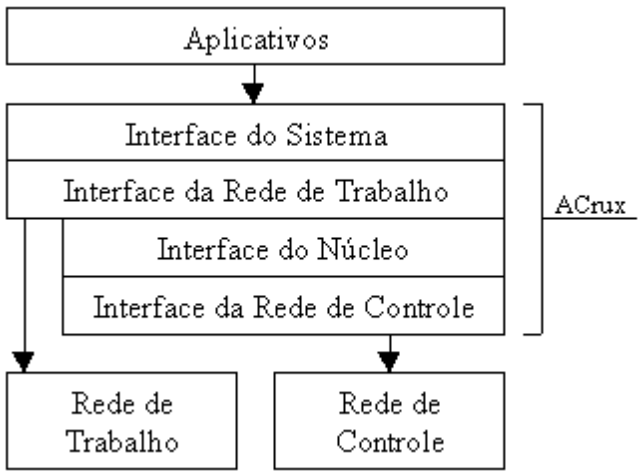

Figura 20: O sistema operacional *ACrux*

3.2.1 O servidor de comunicações

O servidor de comunicações é um dos elementos de maior importância na arquitetura do *Crux*. É através dele que se elimina o roteamento de mensagens, por meio da reconfi-

guração dinâmica dos canais de comunicação entre os nós de trabalho; também é ele que garante a comunicação direta, em todos os casos, entre os nós emissor e receptor.

Essas duas características fundamentais, a comunicação direta e a ausência de roteamento, são os elementos que justificam a existência da arquitetura *Crux*. As vantagens trazidas por tais características são discutidas com detalhe ainda neste capítulo. O modo pela qual elas são possíveis, no entanto, é descrito a seguir.

Sempre que um nó de trabalho deseja se conectar a outro nó, ele deve antes expressar esse desejo ao nó de controle, enviando-lhe uma mensagem pela rede de controle. Essa mensagem é bloqueante, de modo que o nó de trabalho emissor irá esperar a resposta do nó de controle. O nó de controle, ao ser avisado de que um nó de trabalho deseja conectar-se a outro, consulta uma *tabela de conexões* mantida por ele. Se o segundo nó de trabalho ainda não expressou o desejo de se conectar ao primeiro, a resposta ao pedido de conexão não será enviado. O servidor de comunicações voltará a monitorar os pedidos de conexões, e o nó de trabalho que requisitou a conexão continuará suspenso, esperando uma resposta. O segundo nó de trabalho, por fim, enviará uma mensagem ao nó de controle, pedindo uma conexão com o primeiro. Nesse momento, ao consultar a tabela de conexões, o servidor de comunicações constatará que ambos os nós de trabalho já requisitaram sua interligação. Ele irá então enviar a ambos uma mensagem de resposta, indicando que a comunicação pode ser realizada. Antes de enviar a mensagem, no entanto, resta uma tarefa a ser realizada: a reconfiguração da rede dinâmica, de modo a criar um canal de comunicação entre os dois nós de trabalho que desejam se comunicar. Nesse ponto, temos a seguinte situação: os dois nós de trabalho estão interligados por um canal direto (evitando o roteamento de mensagens) e os processos executados por tais nós estão sincronizados, de modo que eles irão começar a escrever/monitorar a rede ao mesmo tempo (garantindo a comunicação direta em seu sentido mais abrangente).

Tanto o servidor de comunicações executado pelo nó de controle quanto os processos pertencentes aos nós de trabalho são implementado através dos recursos que o sistema operacional *ACrux* oferece. Esses recursos, que consistem em chamadas de sistema, estão organizados em uma hierarquia, onde cada camada fornece serviços àquela que se encontra imediatamente acima. As camadas definidas no *ACrux* são detalhadas a seguir.

3.2.2 A interface da rede de controle

A interface da rede de controle [8] é composta pelos seguintes operadores:

- **kc_SendReceive (request, reply)**

- **ks_ReceiveAny (node, request)**
- **ks_Send (node, reply)**

Os três operadores acima são utilizados para implementar a comunicação na rede de controle. Todos os operadores da *interface do núcleo*, que será apresentada a seguir, exigem comunicação entre os nós de trabalho e o nó de controle, e utilizam os operadores da interface da rede de controle para transmitir informações.

O operador *kc_SendReceive* é utilizado pelos nós de trabalho para efetuar uma comunicação com o nó de controle. Ela é bloqueante, ou seja, a mensagem endereçada ao nó de controle é enviada e o processo fica suspenso, aguardando a chegada da resposta. Não existe período definido de *timeout*, pois considera-se que, salvo problemas de programação a nível de usuário, a resposta irá chegar, mais cedo ou mais tarde.

O operador *ks_ReceiveAny* é utilizado pelo nó de controle para receber as mensagens provenientes dos nós de trabalho. Ao ser executado, esse operador não desbloqueia o processo que executou o operador *kc_SendReceive*. O nó de controle pode receber uma mensagem de um nó de trabalho e postergar a resposta indefinidamente.

O operador *ks_Send* é utilizado pelo nó de controle para responder ao nó de trabalho. É a execução desse operador que libera a execução do nó de trabalho que executou o operador *kc_SendReceive*.

3.2.3 A interface do núcleo

A interface do núcleo, como definida em *Corso* [8], é composta pelos seguintes operadores:

- **Connect (node, link)**
- **ConnectAny (node, link)**
- **Disconnect (node)**
- **AllocateAny (node)**
- **Deallocate()**

Os operadores *Connect*, *ConnectAny* e *Disconnect* estão relacionados à criação e liberação de canais de comunicação entre os nós de trabalho. Já os operadores *AllocateAny* e *Deallocate* estão relacionados à alocação e desalocação de nós de trabalho.

O operador *Connect* é utilizado pelos processos pertencentes aos nós de trabalho para requisitarem a configuração de um canal de comunicação direta entre dois desses nós. O processo que realizou uma chamada ao operador *Connect* ficará bloqueado até o momento em que a comunicação direta entre emissor e receptor seja possível.

O operador *ConnectAny* é utilizado pelos processos pertencentes aos nós de trabalho que implementam servidores. Tais processos não sabem com antecedência quais nós irão necessitar de seus serviços, e portanto não podem especificar um nó ao qual querem se conectar. Eles apenas realizam a chamada ao operador *ConnectAny* para indicarem que estão aptos a atenderem pedidos de quaisquer nós do *cluster*. Assim como o operador *Connect*, é uma chamada síncrona e bloqueante. Ela só retornará quando um nó de trabalho requisitar, através do operador *Connect*, uma conexão com o servidor em questão.

O operador *Disconnect* é utilizado pelos processos pertencentes aos nós de trabalho para requisitarem a liberação de um canal de comunicação direto entre dois desses nós. A sincronização provida por esse operador é menor do que aquela oferecida pelos outros operadores dessa camada. O processo que executa uma chamada a *Disconnect* não necessita esperar até que o outro processo ao qual ele está conectado também execute esse mesmo operador, como acontece com os operadores *Connect* e *ConnectAny*. Tão logo um dos processos envolvidos em uma comunicação requisite a desconexão, o servidor de comunicações pode considerar o canal desfeito, e o processo requisitante pode ser liberado para continuar sua execução. O segundo processo envolvido na comunicação não necessita requisitar sua desconexão.

O operador *AllocateAny* é utilizado pelos processos pertencentes aos nós de trabalho para alocarem um novo nó de trabalho. Ela atende a situação em que um processo necessita executar, por exemplo, a chamada de sistema *fork*, que cria um novo processo idêntico ao que realizou a chamada. Como estamos em um *cluster*, e cada um de seus nós executa apenas uma tarefa, deve-se encontrar um nó livre para executar o novo processo que será criado. A chamada *AllocateAny* irá se comunicar com o nó de controle, ou melhor, com o servidor de comunicações nele localizado, que irá, por sua vez, encontrar e alocar um nó disponível.

E o operador *Deallocate*, como já se pode imaginar, irá desocupar um nó do *cluster*. Sempre que um processo encerrar sua execução, ele irá, através desse operador, informar que o nó que ele ocupava está agora livre.

Todos os operadores descritos acima dependem dos serviços fornecidos pela *interface da rede de controle*, abordada na seção anterior.

3.2.4 A interface da rede de trabalho

A interface da rede de trabalho [8] contém os seguintes operadores:

- **s_Send (proc, msg, length)]**
- **s_Receive (proc, msg, length)]**
- **ss_ReceiveAny (proc, msg, length)]**

Esses operadores são utilizados pelos nós de trabalho para efetuar comunicações entre si. Eles utilizam, em sua implementação, os operadores da interface do núcleo e, indiretamente, da rede de trabalho, descritos anteriormente.

O operador *s_Send* é utilizado por um nó de trabalho para enviar informações a outro nó de trabalho. A implementação desse operador exige que, antes da comunicação propriamente dita, seja utilizado o operador *Connect*, da interface do núcleo, para requisitar a conexão e criar o canal físico direto entre os nós. Após a conclusão da comunicação, o operador *Disconnect* deve ser executado para informar ao nó de controle que a conexão não é mais necessária. O operador *s_Send* é bloqueante: enquanto o nó receptor não estiver pronto para receber as informações, o nó emissor ficará suspenso, aguardando.

O operador *s_Receive* é utilizado por um nó de trabalho para receber informações de outro nó de trabalho. De modo semelhante ao operador *s_Send*, ele é implementado utilizando os operadores da interface do núcleo *Connect* e *Disconnect*. É um operador bloqueante, ou seja, enquanto o nó emissor não realizar o *s_Send*, o nó receptor irá suspender sua execução.

O operador *ss_ReceiveAny* é utilizado pelos nós de trabalho que executam processos servidores. Eles devem aceitar conexões de quaisquer nós do *cluster*, e, portanto, não podem utilizar o operador *s_Receive*, que exige a especificação do nó emissor. A implementação do *ss_ReceiveAny* faz uso dos operadores da interface do núcleo *ConnectAny* e *Disconnect*.

Na verdade, como comentado na seção anterior, onde foi abordada a interface do núcleo, apenas um dos lados da comunicação necessita requisitar a desconexão. Isso reduz pela metade o número total de mensagens de desconexão que trafegam pelo sistema na execução de uma aplicação qualquer. Escolheu-se, nessa implementação, retirar a chamada *Disconnect* do operador da rede de trabalho *s_Send*. Sempre que esse operador for executado para realizar uma comunicação, na outra ponta da rede estará sendo exe-

cutado um dos operadores de recebimento de mensagens, *s_Receive* ou *ss_ReceiveAny*; fica portanto a cargo de tais operadores requisitar a desconexão do canal.

Nenhum dos operadores dessa camada, assim como nenhum dos operadores do sistema operacional *ACrux*, tem tempo definido de *timeout*. Considera-se que o meio de comunicação é confiável e que, portanto, cedo ou tarde os operadores serão naturalmente desbloqueados, a não ser em caso de erro de programação a nível de usuário. Nesse caso, cabe ao usuário corrigir seu programa para que sua tarefa possa ser corretamente executada pelo *cluster*.

3.2.5 A interface de sistema

A interface de sistema [8] é equivalente aos operadores oferecidos pelo *Unix* e definidos pelo padrão *Posix*. A única diferença está na implementação de alguns deles.

A maioria dos operadores pode ter uma implementação equivalente ao do sistema operacional *Unix*, sendo executados localmente. Outros, no entanto, devem ter implementação diferenciada por se estar trabalhando com um *cluster* de computadores. O operador *open*, por exemplo, permite que um programa de usuário possa abrir um arquivo para leitura ou escrita. Um nó de trabalho, no entanto, não possui armazenamento local. Toda operação sobre arquivos deve ser feita através de *servidores de arquivo*. O operador *open*, portanto, deve se comunicar com esse tipo de servidor para executar a tarefa que se espera dele. Ele faz isso através dos operadores da interface da rede de trabalho, que permitem que os nós de trabalho se comuniquem. Assim, uma implementação típica do operador *open* faz uso dos operadores *s_Send* e *s_Receive* para se comunicar com um servidor de arquivos, de acordo com um determinado protocolo. A figura 21 apresenta, de forma reduzida, o código do operador *open*, implementado como o cliente de um servidor de arquivos [14].

3.3 Vantagens

Várias propriedades presentes em um *cluster* construído com a arquitetura *Crux* foram salientadas no decorrer deste capítulo. Duas das principais são abordadas agora, separadamente, com mais detalhe, para que fiquem evidentes as vantagens oferecidas por elas.

Uma das propriedades desta rede decorre da escolha de redes dinâmicas na construção do *cluster*. A opção por esse tipo de rede parte da constatação de que a *inexistência de*

```
int open ( const char *name, int flags, ...  )
{
...

    //Preenchendo estrutura com os parametros da chamada
    strcpy ( m.u_call.sopen.path, name );
    m.u_call.sopen.flags = flags;
    m.headr.call = 5;
    m.headr.id = 1;

    ...

    //Enviando mensagem ao servidor de arquivos
    tam_msg = strlen ( m.u_call.sopen.path) +
        sizeof ( m.u_call.sopen.flags) +
        sizeof ( m.u_call.sopen.mode ) +
        sizeof ( m.headr );
    s_Send ( sock_fd, &m, tam_msg );
    //Aguardando a resposta
    tam_msg = sizeof ( m.u_call.m_reply.result ) +
        sizeof ( m.u_call.m_reply.m_errno ) +
        sizeof ( m.headr );
    s_Receive ( sock_fd, &m, tam_msg );

      ...

      return ( m.u_call.m_reply.result );
}
```

Figura 21: Implementação do operador *open*

roteamento, além de simplificar o sistema operacional, torna mais eficiente a comunicação. Sem roteamento, todas as interações entre nós são obrigatoriamente diretas: a mensagem não trafega por nenhum outro nó, além do emissor e do receptor. Isso evita a cópia de mensagens da rede para um computador intermediário, dele novamente para a rede, e assim por diante até a mensagem chegar ao seu destino.

A *comunicação direta* é outra propriedade importante da arquitetura *Crux*. Ela ultrapassa, inclusive, a simples existência de um canal de comunicação direto ligando os nós emissor e receptor; ela objetiva também a transmissão de mensagens diretamente do espaço de endereçamento do processo emissor para o espaço de endereçamento do processo receptor. Não devem existir, nas diversas camadas internas do sistema operacional, cópias da mensagem de um *buffer* a outro, nem no lado emissor, nem no receptor. Essa propriedade é possível devido à sincronização entre nós de trabalho e nó de controle for-

necida pelos operadores bloqueantes *Connect* e *ConnectAny*, necessários na realização de qualquer comunicação pela rede de trabalho. Tais operadores garantem que, ao acontecer uma transmissão, ambos os processos nos nós de trabalho estarão aguardando por sua ocorrência, tornando possível que a mensagem seja transferida diretamente de/para seu espaço de memória. O ganho de desempenho obtido pela inexistência de cópias desnecessárias de mensagem é acompanhado por ganhos no próprio protocolo de transmissão dos dados no meio físico, pois não é necessário particionar os dados em pacotes, eliminando o *overhead* dos cabeçalhos que cada um deles deve sempre possuir, e, conseqüentemente, aumentando a proporção de dados úteis transmitidos.

Pode parecer que comunicação direta e ausência de roteamento estão diretamente ligadas. De fato, a existência da comunicação direta entre os espaços de endereçamento dos processos emissor e receptor depende da inexistência do roteamento de mensagens. O inverso não é verdadeiro. O fato de não se necessitar de roteamento não garante que as comunicações se dêem diretamente do espaço de endereçamento de um processo ao outro. Isso pode ser impossível se a arquitetura do sistema não garantir a perfeita sincronização entre os processos envolvidos na comunicação. Quando isso não ocorre, as mensagens devem ser armazenadas pelo sistema operacional até que o processo receptor esteja em condição de recebê-las, implicando em cópias extras de mensagem, ou seja, *overhead*. A comunicação direta, no entanto, nem sempre é possível, dependendo da aplicação; no capítulo 5, veremos como a arquitetura *Crux* está preparada para ser estendida, podendo trabalhar também, quando necessário, com comunicações não diretas.

3.4 Resumo

Neste capítulo, a arquitetura do *cluster Crux* foi apresentada. Os seus componentes físicos foram enumerados e comentados: os nós de trabalho são os nós que efetivamente executam, cooperativamente, as tarefas passadas ao *cluster*; o nó de controle é responsável por coordenar as interações entre os diversos nós de trabalho; a rede de trabalho, uma rede dinâmica de alta velocidade, interliga os nós de trabalho; e a rede de controle liga nós de trabalho ao nó de controle.

O sistema operacional do *Crux*, o *ACrux*, também foi apresentado. Ele é um sistema operacional hierárquico (composto por diversas camadas) e distribuído, baseado no modelo cliente–servidor. É composto por um núcleo básico existente em todas as máquinas do sistema e processos servidores que oferecem serviços aos nós de trabalho. É dado ênfase na função do servidor de comunicações, responsável tanto pela configuração da rede de

trabalho de acordo com a necessidade dos nós de trabalho, quanto pela sincronização entre o processo emissor e receptor, possibilitando assim a comunicação direta entre eles.

O núcleo do *ACrux* foi explicado em detalhes. Ele é composto por diversas camadas, sendo elas a interface de sistema, baseada no padrão *Posix*; a interface da rede de trabalho, utilizada para a comunicação entre os nós de trabalho; a interface do núcleo, que oferece serviços de alocação e desalocação de nós e canais de comunicação; e a interface da rede de controle, utilizada para a comunicação entre nós de trabalho e o nó de controle.

Por último, as principais características da arquitetura *Crux* são objeto de estudo. Uma das maiores vantagens que ela oferece é a inexistência de roteamento, que torna o sistema operacional e suas rotinas de comunicação mais simples e rápidas. Outra vantagem crucial é a comunicação direta entre os processos emissor e receptor, definida como a cópia da mensagem do emissor diretamente de sua área de memória para o meio físico de comunicação, e desse meio diretamente para a área de memória do processo receptor.

4 *O núcleo do sistema operacional* Linux

Como plataforma de trabalho, o sistema operacional escolhido foi o *Linux*. Deseja-se agora justificar essa escolha e apresentar um breve histórico do sistema. Um pequeno guia de programação no núcleo do *Linux*, muito útil para os que forem dar prosseguimento a este trabalho, também compõe este capítulo. E, como um bônus deste trabalho, apresenta-se uma forma extremamente prática de se alterar, de forma modular, o núcleo do sistema operacional.

As informações contidas neste capítulo são fruto de pesquisas bibliográficas e experiência prática no assunto. A não ser onde expressamente mencionada, a fonte dessas informações é o conjunto da seguinte bibliografia: Comer [7], Projeto Gnu [26], Silberschatz [16] e vários sítios da *internet* [23] [24] [25] [27] [28] [29] [30] [31] [32].

4.1 Escolhendo o sistema operacional

O objetivo deste trabalho é a implementação das primitivas de comunicação do *cluster* de computadores *Crux*. Tais primitivas exigem a execução de rotinas de baixo nível, com acesso direto ao *hardware* do computador. Esse tipo de acesso pode ser conseguido através de sistemas operacionais simplificados, como o *DOS*, que não fazem grandes restrições ao que um programa pode ou não executar. Pode-se conseguir o mesmo efeito buscando uma solução de mais baixo nível ainda (com crescimento proporcional de sua complexidade): a construção completa de um pequeno sistema operacional que, por exemplo, apenas inicie a máquina e seja capaz de executar as primitivas em questão.

Por outro lado, fazendo uso de sistemas operacionais de maior complexidade, pode-se alcançar os objetivos desse trabalho de maneira bem mais simples. Pode-se focar melhor no problema principal sem se perder tanto tempo lutando contra a falta de recursos. Sistemas operacionais mais avançado possuem, no entanto, restrições de segurança. A única maneira de fazer com que determinado código tenha acesso total ao *hardware* do

computador é sua execução em modo privilegiado, ou seja, no núcleo do sistema. Existem dois modos de fazer isso: o primeiro é implementando um *driver* do sistema; o segundo é alterando o núcleo do sistema.

A implementação a partir de *drivers* permite escolher, como plataforma de trabalho, dentre os sistemas operacionais disponíveis comercialmente, como o *Microsoft Windows* ou o *Linux*. Mas se a alternativa for a modificação do núcleo, deve-se utilizar um sistema operacional cujo código-fonte esteja publicamente disponível. Nessa categoria, encontram-se sistemas operacionais acadêmicos como o *Xinu* [7] ou o *Minix* e, novamente, o *Linux*.

A opção da implementação através da alteração do núcleo do sistema é a mais completa por ser a que apresenta maior flexibilidade. Na implementação de um *driver*, existem regras rígidas as serem seguidas. Já alterando o núcleo (adicionando chamadas de sistema, por exemplo) temos acesso total ao sistema operacional e, assim, uma maior liberdade de ação. Optando por essa alternativa, a escolha do sistema operacional a ser utilizado recai naturalmente sobre o *Linux*: é um sistema operacional moderno, muito utilizado atualmente tanto comercial quanto academicamente, com farta documentação disponível e, principalmente, com código-fonte aberto e gratuito.

4.2 O sistema operacional *Linux*

O sistema operacional *Linux* teve sua primeira versão (0.01) liberada em 14 de maio de 1991. Seu autor, o (então) estudante finlandês *Linus Torvald*, também disponibilizou, através da *internet*, o código-fonte do núcleo do sistema. A partir daí, uma rede de colaboradores se formou pelo mundo inteiro, todos ajudando, voluntariamente, a aperfeiçoar esse novo sistema operacional. Em 1994, foi lançada a versão 1.0; em 1996, a versão 2.0.

O *Linux*, desde o início, se inspirou abertamente no sistema operacional *Unix*. A compatibilidade com ele e também com o padrão *Posix* são objetivos do projeto. Isso tornou possível o aproveitamento automático de *software* já existente para plataformas *Unix*. O *Linux*, portanto, já nasceu muito bem servido de aplicativos, utilitários e compiladores, o que certamente ajudou muito em sua aceitação pela comunidade de usuários.

A idéia de desenvolver um sistema operacional semelhante ao *Unix* partiu do projeto *Minix*, um sistema operacional desenvolvido por *Tanenbaum* com objetivos semelhantes, mas voltado ao público acadêmico. *Torvald* era usuário do *Minix*, sendo que as primeiras versões do *Linux* foram compiladas em máquinas executando esse sistema operacional. Também foi no grupo de usuários do *Minix* que *Torvald* anunciou publicamente a existência do *Linux* e conseguiu seus primeiros ajudantes.

O *Linux* acabou por se tornar símbolo da filosofia do *software livre*, que prega, simplificadamente, a distribuição do código-fonte de todo *software* e a liberdade de qualquer usuário para modificar esse código e/ou redistribuir o *software*. O *software* livre, representado pelo projeto GNU [26], tem grande participação no *Linux*: o sistema operacional é composto pelo núcleo do *Linux* (que é um *software* livre) mais uma ampla gama de aplicativos, boa parte deles fazendo parte do projeto GNU ou sendo distribuído através de sua licença.

4.3 Criando chamadas de sistema

Após uma breve introdução ao sistema operacional *Linux*, chega-se agora ao ponto central desse capítulo: como alterar o núcleo do sistema para incluir novas chamadas de sistema. Apresenta-se ainda a comparação do método tradicional de alteração estática do código-fonte do núcleo com o método modular de alteração dinâmica da imagem do núcleo carregada na memória do computador.

Seria interessante, antes de mais nada, mencionar o fato de que trabalha-se nesse projeto com uma distribuição *Conectiva* do sistema operacional *Linux*, mais precisamente o *Conectiva 6*, *kernel* versão 2.2.17. Assim, certos procedimento descritos daqui por diante podem dizer respeito a essa distribuição ou versão de *kernel* em particular. No entanto, de modo geral, o que for relatado aqui se aplica, certamente, a toda e qualquer distribuição do *Linux*. Vale lembrar também que apenas o administrador do sistema pode, em ambas as abordagens, criar novas chamadas de sistema.

4.3.1 Abordagem tradicional

O primeiro passo para adicionar novas chamadas ao núcleo do sistema é incorporar seu código-fonte ao *Linux*. Todos os fontes do *Linux* estão situados em */usr/src/linux*. Nesse local cria-se uma nova pasta (chamada de *ACrux*, por exemplo) e nela coloca-se o código-fonte. A seguir, o arquivo *makefile* principal do *Linux* deve ser alterado (figura 22) para indicar que a nova pasta também deve ser incluída no processo de compilação do núcleo do sistema operacional.

Nesse ponto, ao se recompilar o *kernel*, o novo código já estará presente no sistema operacional, mas ainda não terá sido exportado, ou seja, ainda não existem chamadas de sistema que permitam utilizá-lo. Tais chamadas de sistema são criadas através da alteração dos arquivos *entry.S* (figura 23), na pasta */usr/src/linux/arch/i386/kernel* e *unistd.h* (figura 24), na pasta */usr/src/linux/include/asm*.

```
...
# Include the make variables (CC, etc...)
#

CORE_FILES =kernel/kernel.o mm/mm.o fs/fs.o ipc/ipc.o /
            acrux/acrux.o
FILESYSTEMS =fs/filesystems.a
NETWORKS =net/network.a
DRIVERS =drivers/block/block.a drivers/char/char.o /
           drivers/misc/misc.a
LIBS =$(TOPDIR)/lib/lib.a
SUBDIRS =kernel drivers mm fs net ipc lib acrux
...
fs lib mm ipc acrux kernel drivers net:  dummy
 $(MAKE) $(subst $@, _dir_$@, $@)
...
```

Figura 22: Trecho do arquivo *makefile* do *Linux*

O próximo passo é recompilar o núcleo. Para tanto, basta, estando na pasta */usr/src-/linux*, digitar o comando *make*. O processo de compilação pode se estender por vários minutos mas, após terminado, o novo arquivo contendo o núcleo do sistema estará situado na pasta *usr/src/linux/arch/i386/kernel*, com o nome de *vmlinux*. A tarefa se completa quando se substitui o arquivo *vmlinux* na pasta */boot* com o arquivo *vmlinux* recém criado. Após reinicializar o sistema, o novo núcleo do sistema será carregado na memória, e as chamadas de sistema estarão prontas para o uso. Alternativamente, pode-se manter tanto o núcleo original quanto o núcleo modificado e, utilizando um gerenciador de *boot*, escolher em tempo de inicialização da máquina qual dos núcleos se deseja utilizar.

Falta um último passo para que os programas de usuário possam fazer uso das novas chamadas de sistema. Sempre que um programa executa uma chamada de sistema, a interrupção 0x80 é acionada, e no registrador *EAX* é colocado o número correspondente à chamada que se deseja executar. Tal operação é escondida do usuário pela biblioteca *libc*. Ao utilizá-la, o usuário precisa apenas realizar a chamada à função *open*, por exemplo; a implementação dessa função na *libc* cuida de colocar em *EAX* a constante *SYS_OPEN* (que identifica a posição da chamada *open* na tabela de chamadas), e, nos registadores seguintes (*EBX*, *ECX*, etc.), os vários parâmetros passados pelo usuário para essa chamada em particular. A constante *SYS_OPEN*, e todas as outras constantes que identificam as chamadas de sistema, podem ser encontradas no arquivo *unistd.h*, na pasta */usr/src/linux/include/asm*. O arquivo *unistd.h* já foi alterado para incluir as constantes que identificam as novas chamadas de sistema, mas a *libc* ainda não as conhece. Para

```
...
.long SYMBOL_NAME(sys_sigaltstack)
.long SYMBOL_NAME(sys_sendfile)
.long SYMBOL_NAME(sys_ni_syscall) /* streams1 */
.long SYMBOL_NAME(sys_ni_syscall) /* streams2 */
.long SYMBOL_NAME(sys_vfork) /* 190 */
.long SYMBOL_NAME(sys_s_Send)
.long SYMBOL_NAME(sys_s_Receive)
.long SYMBOL_NAME(sys_ss_ReceiveAny)
.long SYMBOL_NAME(sys_s_GetData)
.long SYMBOL_NAME(sys_ks_Send)
.long SYMBOL_NAME(sys_ks_ReceiveAny)
.long SYMBOL_NAME(sys_kc_SendReceive)

/*
* NOTE!!  This doesn't have to be exact - we just have
* to make sure we have _enough_ of the "sys_ni_syscall"
* entries.  Don't panic if you notice that this hasn't
* been shrunk every time we add a new system call.
*/
.rept NR_syscalls-197
   .long SYMBOL_NAME(sys_ni_syscall)
.endr
```

Figura 23: Trecho do arquivo *entry.S*

resolver esse problema, existe a alternativa de alterar e recompilar a biblioteca; isso, no entanto, torna mais complexo todo o processo que se está expondo aqui. É muito mais simples definir tais chamadas em algum arquivo que o programa de usuário deva incluir para poder utilizá-las. Existem *macros* específicas para a tarefa de declarar novas chamadas no ambiente de usuário. Elas também estão definidas no arquivo *unistd.h*, e são identificadas pelos nomes de *_syscall0*, *_syscall1*, *_syscall2*, *_syscall3*, *_syscall4* e *_syscall5*. Deve-se utilizá-las de acordo com o número de parâmetros da chamada que se deseja declarar (figura 25). Pode-se concluir então que cinco é o número máximo de parâmetros possível para chamada de sistema do sistema operacional *Linux*. Isso acontece porque, como já mencionado, cada parâmetro deve ser atribuído a um registrador quando da chamada da interrupção 0x80. Como o número de registradores é limitado, também o é o número máximo de parâmetros.

O método que acabamos de descrever para implementar novas chamadas de sistema no núcleo do sistema operacional, encontrado na literatura e também disponível na *internet*, tem algumas desvantagens. Em primeiro lugar, exige que o código original do *Linux*,

```
...
#define __NR_sigaltstack     186
#define __NR_sendfile        187
#define __NR_getpmsg         188 /* some people actually
                                    want streams */
#define __NR_putpmsg         189 /* some people actually
                                    want streams */
#define __NR_vfork           190
#define __NR_s_Send          191
#define __NR_s_Receive       192
#define __NR_ss_ReceiveAny   193
#define __NR_s_GetData       194
#define __NR_ks_Send         195
#define __NR_ks_ReceiveAny   196
#define __NR_kc_SendReceive  197
```

Figura 24: Trecho do arquivo *unistd.h*

```
...
_syscall2 (int, s_GetData, int, node, int*, data );
_syscall2 (int, ks_Send, int, node, CTRLBUF*, msg );
_syscall2 (int, ks_ReceiveAny, int*, node, CTRLBUF*, msg );
_syscall3 (int, kc_SendReceive, int, thisProc, CTRLBUF*,
    msg, CTRLBUF*, ret );
```

Figura 25: Mapeando as chamadas de sistema

situado na pasta */usr/src/linux*, seja modificado. Um usuário menos experiente pode se sentir intimidado por tal tarefa. Ela pode ser automatizada através de *scripts* mas, se aplicados sobre um núcleo já modificado, podem ter efeitos inesperados. É necessário recompilar o núcleo e alterar a pasta */boot*, outra tarefa que pode não ser trivial para certos usuários. E, por último, é necessário reiniciar a máquina que estamos alterando, algo que, se não é crítico na maioria dos casos, é certamente inconveniente.

Neste trabalho adotou-se uma metodologia bastante diferente. Ela não possui as desvantagens mencionadas e torna a alteração no núcleo extremamente simples tanto de ser realizada quanto também desfeita. Na próxima seção, aborda-se com detalhes essa nova técnica.

4.3.2 Abordagem modular

A abordagem aqui apresentada pode ser dita modular por empregar o mesmo recurso utilizado para carregar módulos contendo *drivers* do sistema: os *LKM*s – *Loadable Kernel Modules*, ou *módulos carregáveis do núcleo*. Os módulos são arquivos objeto, semelhantes às bibliotecas utilizadas na compilação de programas. O que os diferencia são algumas funções predeterminadas, conhecidas como *pontos de entrada*, que todo módulo deve definir. É através desses pontos de entrada que o núcleo do sistema operacional pode interagir com o módulo para carregá-lo ou removê-lo da memória.

Os módulos são muito utilizados para a instalação de *drivers* do sistema. Existe uma função executada no momento do carregamento do módulo; ela é executada pelo núcleo em modo privilegiado, tendo assim total acesso às estruturas e funções exportadas. No momento do carregamento, seguindo determinado protocolo, o módulo pode então registrar um *driver* do sistema. No entanto, aproveitando o acesso privilegiado ao núcleo do sistema, várias outras operações podem ser executadas: entre elas, a instalação de novas chamadas de sistema.

A operação de adicionar uma chamada de sistema só é possível porque o núcleo do *Linux* exporta uma estrutura conhecida como *tabela das chamadas de sistema* (figura 26), declarado no arquivo *entry.S* na pasta *usr/src/linux/arch/i386/kernel*. Ela possui 255 entradas, sendo que cada entrada contém o endereço correspondente a uma das chamadas de sistema existentes no núcleo do sistema operacional.

```
.data
ENTRY(sys_call_table)
    .long SYMBOL_NAME(sys_ni_syscall) /* 0 - old
        "setup()" system call*/
    .long SYMBOL_NAME(sys_exit)
    .long SYMBOL_NAME(sys_fork)
    .long SYMBOL_NAME(sys_read)
...
```

Figura 26: Declaração da tabela de chamadas de sistema

Tendo acesso à tabela de chamadas de sistema, o objetivo passa a ser adicionar uma nova entrada a ela. Basta descobrir a posição correta para as novas entradas. A posição não pode ser constante, pois o número das entradas originais do sistema pode ser alterado de uma versão do núcleo para outra. Se o módulo utilizar uma posição constante para a inserção das chamadas, estará limitando seu uso a uma determinada versão do núcleo do

Linux. Infelizmente, o número de entradas da tabela de chamadas não é exportado, não sendo possível descobrir seu valor. Sabe-se, entretanto, que toda entrada livre é preenchida com o endereço de uma função responsável por tratar chamadas inválidas de sistema que, ao ser executada, apenas seta a variável de erro do sistema (*errno*, na linguagem *C*) e retorna. Consultando a última entrada da tabela de chamadas de sistema (que certamente estará livre), podemos descobrir o endereço dessa função. Percorrendo a tabela, pode-se encontrar a sua primeira aparição e, portanto, a primeira chamada de sistema livre. A partir daí, preenche-se a tabela com novas chamadas.

Ao se remover o módulo da memória, o processo inverso deve ser executado. Para isso, deve-se sempre armazenar, no carregamento do módulo, a posição inicial das novas chamadas de sistema e sua quantidade. Basta então percorrer a tabela de chamadas de sistema restaurando as entradas que foram modificadas. Restaurá-las significa recuperar seu valor anterior, ou seja, o endereço da função que trata chamadas de sistema inválidas. Esse endereço pode ser novamente consultado a partir da última posição da tabela ou armazenado durante o carregamento do módulo.

Para que os programas de usuário possam fazer uso das novas chamadas de sistema, eles devem declará-las, como já mencionado na seção anterior, através das *macros* da família *_syscall* declaradas no arquivo *unistd.h*. Existe, entretanto, uma diferença: o arquivo *unistd.h* não foi alterado para incluir as constantes que identificam as chamadas de sistema. Basta definir essa constante no espaço de usuário, mas deve-se primeiro descobrir qual o seu valor. Como a inserção das chamadas é um processo dinâmico, a posição delas na tabela de chamadas pode ser diferente de um momento para outro. É necessário então que exista algum modo de fornecer ao usuário essa informação. No momento do carregamento, o módulo pode, por exemplo, imprimir essa informação como uma mensagem de sistema (acessível ao usuário através do comando *dmesg*). Pode também definir uma variável de ambiente ou criar uma área de memória compartilhada. Enfim, existem inúmeros modos de se fazer isso.

De posse da posição das chamadas de sistema na tabela, as constantes que identificam cada chamada de sistema devem ser definidas e o programa deve ser compilado. Em caso de alteração na posição das chamadas de sistema, torna-se necessário uma recompilação do programa. Pode-se, para evitar esse inconveniente, criar algum método de reconhecimento dinâmico das posição das chamadas de sistema. Se a posição estiver armazenada em uma variável de sistema, por exemplo, basta consultá-la e utilizá-la para realizar as chamadas de sistema. Mas as *macros syscall* não poderão ser utilizadas nesse caso. As *macros* são resolvidas em tempo de compilação; ao fazer uso delas, obriga-se o usuário a sempre

recompilar o programa em caso de alterações na tabela de chamadas.

A técnica apresentada nesta seção parte do princípio de que a última entrada da tabela de chamadas de sistema está livre. Se, para um determinado núcleo isso não for verdade, a operação de instalação das chamadas de sistema pode danificar a imagem do núcleo carregada na memória, substituindo a última chamada de sistema original da tabela pela chamada que se deseja carregar. Isso pode ser evitado ao se exigir sempre a existência de duas chamadas de sistema livres. Desse modo, em uma tabela de chamadas completamente preenchida, a última entrada seria considerada livre, mas a penúltima não (pois seu conteúdo dificilmente seria igual ao da última entrada da tabela). Nesse caso, a instalação das novas chamadas simplesmente falharia, e a imagem carregada do núcleo não seria danificada.

Pode-se imaginar que a abordagem modular proposta neste trabalho é mais complicada do que a abordagem tradicional. Essa impressão é falsa, e pode dever-se ao fato de que a abordagem modular foi explicada com grandes detalhes, ao passo que a abordagem tradicional não mereceu tanto destaque por já ser bastante conhecida. E, mesmo que a nova abordagem fosse realmente mais complexa, as vantagens que ela oferece compensariam um pequeno aumento na dificuldade de implementação. As vantagens são várias: o código do *Linux*, situado na pasta */usr/src/linux*, não necessita ser modificado; o núcleo não precisa ser recompilado; as alterações podem ser aplicadas e desfeitas em segundos; a máquina não precisa ser reinicializada em momento algum; e é muito fácil distribuir um *software* baseado nessa técnica.

Essa última vantagem é, certamente, uma das mais importantes. Muitos usuários simplesmente não instalariam um *software* que os obrigasse a alterar e recompilar o código-fonte de seu sistema operacional. Na abordagem modular, no entanto, basta carregar, através de um comando no *shell*, o módulo de interesse do usuário; quando ele não for mais necessário, outro comando remove o módulo da memória e desfaz todas as alterações. Simples e rápido. Os comandos, a propósito, são: *insmod módulo.o* para carregar um módulo; *rmmod módulo* para remover um módulo da memória. Mais informações estão disponíveis através do comando *man insmod*. Reforça-se que tais comandos devem ser executados pelo administrador do sistema; pode-se também inseri-los nos arquivos de inicialização do sistema, executando-os independentemente do usuário que estiver logando.

4.4 Programando no núcleo

Nesta seção apresenta-se um pequeno guia de programação no núcleo do *Linux*. Ele foi incluído neste trabalho porque existem poucas compilações de comandos e técnicas referentes a esse assunto. Os textos nunca estão completos e, além disso, divergem entre si. Por isso, através da experiência prática adquirida na realização deste trabalho, decidiu-se agrupar aqui das informações recolhidas durante o desenvolvimento deste trabalho. Espera-se também que as informações aqui contidas ajudem futuros colaboradores do projeto *Crux*.

Existem alguns pontos principais que dificultam o trabalho dos programadores que resolvem se aventurar nas entranhas do núcleo do *Linux*:

- Há certa falta de documentação. Enquanto a *libc* dispõe das *manpages* (páginas de manual) como fonte de referência, não existe nada semelhante em relação ao núcleo do sistema operacional.
- Não se pode contar com a biblioteca *libc*, o que impede a utilização de diversas funções de grande utilidade. Embora isso possa causar alguns transtornos, existem maneiras de contorná-los. Em último caso, pode-se consultar o código-fonte da biblioteca e reimplementar, no núcleo, a função desejada. Mas antes de se chegar a esse extremo, saiba-se que existem diversas funções equivalentes (ou ao menos semelhantes) às da *libc* definidas dentro do próprio núcleo. O problema é descobrir quais são e onde elas se encontram.
- O núcleo do *Linux* executa em um segmento diferente dos programas de usuário, exigindo tratamento especial na hora da comunicação entre eles.

O presente trabalho tem como objetivo a construção de chamadas de sistema no núcleo do *Linux*. Também são essas chamadas o principal meio de comunicação entre programas de usuário e o núcleo do sistema operacional. É, portanto, através das chamadas de sistema que serão explicitadas algumas das particularidades da programação no núcleo do sistema operacional.

Os parâmetros recebidos pelas chamadas de sistema são os primeiros elementos que devem receber atenção especial. Se tais parâmetros são ponteiros, eles apontam para endereços dentro do segmento utilizado pelo processo de usuário que está realizando a chamada de sistema. Portanto, tal ponteiro não pode ser usado diretamente pelo *kernel*, que está sendo executado em um segmento diferente. A conversão é feita pelos *procedimentos utilitários* [18] *copy_to_user* e *copy_from_user*. O primeiro copia informações do

segmento utilizado pelo núcleo para o segmento do processo de usuário; o segundo faz o inverso – copia informações de um segmento de usuário para o segmento do núcleo. Pode ser necessário, ao se copiar informações do segmento do usuário, alocar dinamicamente memória no segmento do núcleo. A gerência da alocação e desalocação de memória do núcleo é realizada pelas funções *kmalloc* e *kfree*, semelhantes aos operadores da *libc malloc* e *free*. Um trecho de código que faz uso dos operadores *copy_from_user*, *kmalloc* e *kfree* pode ser observado na figura 27.

```
asmlinkage int sys_s_Send ( int thisProc, int proc,
void *msg, int length )
{
  int ret = -1;
  int *kbuf = 0;

  kbuf = (int*)kmalloc ( length );
  copy_from_user ( kbuf, msg, length );

  ...

  kfree ( kbuf );
  return ret;
}
```

Figura 27: Trecho de código exemplificando o uso dos operadores *copy_from_user*, *kmalloc* e *kfree*

Ao se criar uma nova chamada de sistema, pode ser necessário realizar, dentro dela, uma outra chamada de sistema: uma *chamada de sistema aninhada*, pode-se dizer. Na abordagem estática, tradicional, de alteração do núcleo, todas as chamadas de sistema executadas de dentro do próprio núcleo devem receber o prefixo *sys_* (é uma convenção; toda chamada de sistema é declarada internamente com o prefixo *sys_*, mas exportada sem ele). Tome-se a chamada *open*, por exemplo. Fora do núcleo, em um programa de usuário, as referências a ela são realizadas através do nome *open*. Já no núcleo, ela pode ser utilizada através do comando *sys_open*. A abordagem modular de alteração do núcleo obriga a um procedimento um pouco diferente para que se possa utilizar chamadas de sistema; ele será descrito logo adiante. A figura 28 compara dois trechos de código utilizando os operadores *open* e *close*, sendo o primeiro uma função usual, e o segundo uma função situada no interior do núcleo.

Outro cuidado que se deve ter na utilização de chamadas de sistema aninhadas é com a passagem de parâmetros como ponteiros. As chamadas nativas estão preparadas

```
//Funcao comum
int lerArquivo ( char *nomeArq )
{
  int ret = -1;
  int fd = 0;

  fd = open ( nomeArq )

  ...

  close ( fd );
  return ret;
}

//Funcao situada no nucleo
int lerArquivo ( char *nomeArq )
{
  int ret = -1;
  int fd = 0;

  fd = sys_open ( nomeArq )

  ...

  sys_close ( fd );
  return ret;
}
```

Figura 28: Trecho de código exemplificando o uso, no interior do núcleo, de chamadas de sistema

para receber ponteiros endereçando segmentos de usuário, e, portanto, fazem uso dos já mencionados procedimentos utilitários *copy_to_user* e *copy_from_user*. Mas se tais chamadas já estão sendo realizadas de dentro do núcleo, os ponteiros sendo passados como parâmetro para elas podem, em certas ocasiões, estar apontando para dados já situados no segmento do núcleo. Para tratar essa situação existe mais um procedimento utilitário, denominado *set_fs*, que é utilizado para indicar às funções de cópia o segmento válido para os ponteiros passados para elas. Assim, antes de uma chamada de sistema nativa, consulta-se o segmento de dados ativo através de uma chamada à função *get_fs*. Executa-se então um *set_fs (KERNEL_DS)* para indicar que já se está trabalhando em um segmento do núcleo, e, assim, os ponteiros podem ser utilizados diretamente. Após o término da chamada, executa-se *set_fs*, para voltar ao estado anterior (aquele retornado por *get_fs*). Ou, se existir a certeza de que se deve retornar ao segmento do

usuário, pode-se simplesmente executar *set_fs (USER_DS)*. A figura 29 apresenta uma função que encapsula todo o procedimento acima, fazendo com que o uso das chamadas de sistema aninhadas seja um pouco mais amigável. Esse exemplo apresenta o mapeamento da chamada de sistema *close*.

```
int _crux_close(int fd)
{
  int ret = 0;
  mm_segment_t oldFs = get_fs();

  set_fs ( KERNEL_DS );
  ret = sys_close ( fd );
  set_fs ( oldFs );

  return ( ret );
}
```

Figura 29: Trecho de código exemplificando o uso dos operadores *set_fs* e *get_fs*

Vale lembrar que grande parte das fontes de informação sobre programação no núcleo do *Linux* menciona a impossibilidade da utilização de chamadas de sistema aninhadas. Isso provavelmente deve-se ao desconhecimento da utilização correta das funções *copy_to_user* e *copy_from_user* em conjunção com as funções *set_fs* e *get_fs*. Essas mesmas fontes (e algumas outras mais) insistem em apontar que também os módulos não podem fazer uso de chamadas de sistema. Mais uma vez, é uma informação incorreta. De fato, a maioria das chamadas de sistema, identificadas pelo prefixo *sys_* , não é exportada pelo núcleo, não sendo portanto diretamente acessível para código situado no interior de módulos. Como mencionado anteriormente, o uso de chamadas de sistema por módulos é possível, desde que se utilize a técnica correta. Novamente, a tabela de chamadas de sistema é a solução. Essa tabela é exportada, sendo acessível aos módulos. E, como já mencionado neste trabalho, ela contém o endereço de todas chamadas de sistema existentes no núcleo do sistema operacional. Basta definir um ponteiro com a declaração exata da chamada que se deseja utilizar e copiar a entrada correspondente da tabela de chamadas de sistema para esse ponteiro. Tem-se assim o tão desejado acesso às chamadas de sistema originais do sistema. A figura 30 apresenta um trecho de código que consulta a tabela de chamadas de sistema, copiando vários de seus endereços para ponteiros de funções. Reapresenta também o código da figura 29, agora utilizando tais ponteiros.

Por último, temos outro procedimento utilitário de grande utilidade: o *printk*. Com essa chamada, pode-se escrever na saída padrão do sistema, que pode ser consultada

através do comando *dmesg* em qualquer *shell* do *Linux*. Essa ferramenta é simplesmente indispensável quando se torna necessário depurar código situado no núcleo do sistema operacional. Um exemplo de seu uso pode ser encontrado na figura 31.

Certamente existem várias outras questões referentes à programação no espaço do *kernel* que não são abordadas aqui. Um guia completo do assunto seria, por si só, material para um livro. Foram apresentadas somente as informações de maior importantância para este projeto em particular. De posse delas, espera-se que outros interessados possam iniciar com mais tranqüilidade a exploração do núcleo do *Linux*.

4.5 Resumo

O tema deste capítulo foi o sistema operacional *Linux*. O motivo que levou o *Linux* a ser utilizado neste projeto foi o fato de ser um *software* livre, portanto gratuito e com código aberto. Também foi apresentado um pequeno histórico desse sistema operacional, onde se colocou que *Linus Torvald*, então um estudante, iniciou em 1991 o seu desenvolvimento, baseando-o no *Minix* de *Tanenbaum*.

Dois métodos utilizados para a criação de chamadas de sistema no núcleo do sistema operacional *Linux* foram comentados. O primeiro, mais difundido, necessita que o código-fonte do *Linux* seja alterado e recompilado. O segundo, fazendo uso de módulos carregáveis, é capaz de alterar (ou restaurar) dinamicamente a imagem do núcleo do sistema operacional carregada na memória, sem necessidade de recompilação ou alteração de seu código-fonte.

Um pequeno guia de programação no modo privilegiado do núcleo do *Linux* também foi elaborado. Nele, foram expostos os cuidados necessários na passagem de parâmetros à chamadas de sistema, como a utilização das funções *copy_to_user* e *copy_from_user*; a utilização de chamadas de sistema aninhadas também mereceu destaque, sendo demonstrado como a utilização correta dos operadores *set_fs* e *get_fs* permite seu uso; foram apresentados também alguns dos operadores existentes no núcleo, capazes de substituir funções da *libc* inacessíveis em modo privilegiado, como *kmalloc*, *kfree* e *printk*.

```
asmlinkage int (*_socketcall)(int call, unsigned
                                          long *args);
asmlinkage int (*_close)(int fd);
asmlinkage int (*_fcntl)(int fd, int cmd, long arg);
asmlinkage int (*_nanosleep)(const struct timespec *req,
    struct timespec *rem);
asmlinkage int (*_yield)(void);
asmlinkage int (*_getpid)(void);
asmlinkage int (*_gettimeofday)(struct timeval *tv,
              struct timezone *tz);

void setarChamadas()
{
  _socketcall = sys_call_table[__NR_socketcall];
  _close = sys_call_table[__NR_close];
  _fcntl = sys_call_table[__NR_fcntl];
  _nanosleep = sys_call_table[__NR_nanosleep];
  _yield = sys_call_table[__NR_sched_yield];
  _getpid = sys_call_table[__NR_getpid];
  _gettimeofday = sys_call_table[__NR_gettimeofday];
}

...

int _crux_close(int fd)
{
  int ret = 0;
  mm_segment_t oldFs = get_fs();

  set_fs ( KERNEL_DS );
  ret = _close ( fd );
  set_fs ( oldFs );

  return ( ret );
}
```

Figura 30: Trecho de código exemplificando o uso de chamadas de sistema a partir de módulos

```
int lerArquivo ( char *nomeArq )
{
  int ret = -1;
  int fd = 0;

  fd = sys_open ( nomeArq )

  ...

  sys_close ( fd );
  printk ( "lerArquivo retornou %d", ret );
  return ret;
}
```

Figura 31: Trecho de código exemplificando o uso do operador *printk*

5 *Implementação das primitivas de comunicação no núcleo do* Linux

Nos capítulos anteriores, fez-se uma revisão de todos os conhecimentos necessáros para o desenvolvimento do presente trabalho. Neste capítulo, pretende-se abordar o processo em si, a implementação do sistema operacional *ACrux* como uma extensão do núcleo básico do *Linux*. Apresenta-se desde modificações no *hardware* e no *software* da arquitetura *Crux*, para redução de custos e portabilidade, quanto detalhes de implementação do sistema operacional que o preparam para futuros trabalhos.

5.1 Implementação

A implementação em si consiste em seguir as especificações contidas em *Corso* [8], e torná-las realidade na plataforma *Linux*. Para isso, no entanto, era necessário um conhecimento mais aprofundado desse sistema operacional, principalmente do código-fonte de seu núcleo. Todo o trabalho de pesquisa necessário foi descrito nos capítulos anteriores. De posse das informações necessárias, a implementação seguiu seu curso natural, de modo relativamente rápido.

O código-fonte resultante desse trabalho está presente como um apêndice deste trabalho. Seria didático, porém cansativo, repetir neste capítulo, todo esse código. As principais explicações sobre ele já foram dadas no capítulo 3, que trata de sua especificação. Cita-se aqui apenas um trecho de código (figura 32), interessante por reunir e exemplificar algumas das características já discutidas sobre a arquitetura *Crux*.

O código apresentado refere-se ao operador *s_Send*, da interface da rede de trabalho. Os pontos de interesse que ele apresenta são:

Interface do núcleo O operador *s_Send* utiliza os operadores do núcleo *Connect* e *Disconnect* para requisitar ao servidor de comunicações o estabelecimento de um canal de comunicação, na rede de trabalho, ligando os nós identificados pelas variáveis

```
asmlinkage int sys_s_Send ( int thisProc, int proc,
 void *msg, int length )
{
  int link = 0;
  int L = sizeof ( length );
  int ret = -1;
  int *kbuf = 0;

  Compute ( thisProc, S_SEND, length );

  kbuf = (int*)kmalloc ( length );
  copy_from_user ( kbuf, msg, length );

  Connect ( thisProc, proc, '&link );
  if ( link >= 0 )
  {
    _send ( link, L, &length );
    _send ( link, length, kbuf );

    //Apenas um dos lados necessita requisitar a desconexao
    //Disconnect ( thisProc, proc );
    ret = 0;
  }

  kfree ( kbuf );

  Compute ( thisProc, S_SEND, length );
  return ret;
}
```

Figura 32: O operador *s_ Send*, da interface da rede de trabalho

thisProc e *proc*. É a implementação da *comunicação direta*, uma das principais características da arquitetura *Crux*. Pode-se perceber também que, na verdade, o operador *Disconnect* não é realmente utilizado; seu lugar apenas está demarcado por um comentário, indicando que seu uso é opcional. Como já comentado (na seção 3.2.4), apenas um dos lados da comunicação necessita requisitar sua desconexão. Nessa implementação, o operador *s_ Send* não pede a desconexão; isso fica a cargo dos operadores complementares *s_ Receive* e *ss_ ReceiveAny*.

Interface de acesso ao meio físico Essa camada, que será discutida em detalhes na seção 5.3, oferece acesso ao meio físico da rede de trabalho. Ela é representada no código pelas chamadas chamadas ao operador *_ send*. Perceba-se que a primeira chamada é utilizada para enviar ao destinatário o tamanho da mensagem seguinte.

Todas as mensagens que transitam pelo sistema devem ter tamanho conhecido; nada impede, no entanto, que esse tamanho seja definido dinamicamente, algo possível a partir da utilização desse pequeno protocolo.

Coleta de estatísticas O código apresenta chamadas ao operador *Compute*. Essa chamada destina-se a recolher informações sobre o uso do operador *s_Send*; as informações coletadas são o número de vezes que um operador foi chamado, o tempo gasto em sua execução e o número de *bytes* que ele transmitiu ou recebeu. Todos os operadores da interface da rede de trabalho coletam tais estatísticas. A interface da rede de controle não coleta, nessa implementação, tais estatísticas, pois é uma camada subordinada à interface da rede de trabalho; a maioria das informações pode ser deduzida a partir dos dados coletados na camada superior. A coleta de estatísticas foi implementada de modo que possa ser ativada e desativada dinamicamente, mesmo durante a execução de tarefas. O mecanismo de ativação e desativação é acionado através da chamada de sistema auxiliar *s_GetData*.

Utilização de operadores do núcleo O operador *s_Send* é implementado como uma chamada de sistema. Por executar em modo privilegiado, no mesmo segmento do núcleo, ele faz uso de operadores do núcleo que o auxiliam a copiar dados do segmento do usuário (*copy_from_user*) e a alocar memória (*kmalloc*).

Após esses breves comentários sobre o código gerado a partir das especificações do sistema operacional *ACrux*, pode-se partir para a discussão de outros pontos de interesse. Algumas situações não estavam previstas na especificação original do *Crux*, e as alterações resultantes são descritas nas seções seguintes.

5.2 Plataforma de desenvolvimento

Não existe, no momento, um *cluster* de computadores construído de acordo com as especificações do *Crux*. Basicamente, o que falta é o elemento comutador da rede de trabalho, sendo ele o elemento determinante dos custos dessa arquitetura. Foi proposto assim um modelo alternativo, capaz de fornecer grande desempenho a custos menores. Ele propõe substituir a rede de trabalho baseada em comutação, instalando em seu lugar uma rede do tipo *Firewire* (IEEE 1394). A rede de controle, tirando proveito do alto desempenho que ele oferece, seria mapeada nesse mesmo meio físico. A figura 33 descreve esquematicamente duas topologias (dentre várias) possíveis para o *Crux* sobre *FireWire*. Os problemas de roteamente são tratados a nível de *hardware*, não sendo necessário, para isso, alterar o sistema operacional *ACrux*.

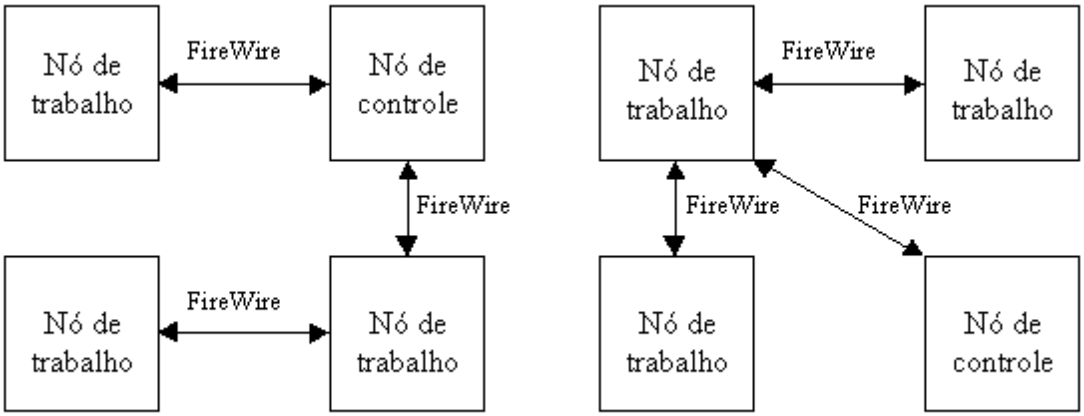

Figura 33: Duas topologias possíveis para o *Crux* sobre *FireWire*

O *FireWire* é uma rede ponto-a-ponto *serial* de alta velocidade, capaz de atingir, na sua versão atual (IEEE 1394a), velocidades de 100, 200 e 400 Mbps ([33] [34]). Especificações mais recentes (IEEE 1394b), no entanto, prometem aumentar o desempenho dessa tecnologia, oferecendo velocidades adicionais de 800 Mbps, 1.6 e 3.2 Gbps, mantendo a compatibilidade com as especificações anteriores. A evolução de desempenho é muito grande, considerando-se que a especificação original previa velocidade máxima de 50 Mbps.

O nome *FireWire* é marca registrada da Apple Computers, empresa que propôs pela primeira vez, em 1986, tal tecnologia. Após ser transformada, em 1995, em um padrão da IEEE, sob a norma 1394, outras empresas desenvolveram produtos baseados nessa tecnologia (a Sony, por exemplo, comercializa o *FireWire*, no Japão, sob o nome de *iLink*). Os computadores da Apple sempre tiveram grande aceitação no meio de produção gráfica, devido ao seu ótimo desempenho e aos *softwares* disponíveis nessa plataforma. Nesse tipo de atividade, costuma-se lidar com grande quantidade de dados; pequenas quantidades de imagens e vídeos podem, atualmente, consumir várias centenas de *megabytes*. A situação era a mesma, guardadas as proporções, nos idos de 1986, quando a Apple decidiu apresentar o *FireWire* como sua solução para a transferência, de um dispositivo para outro, de grande quantidade de dados a altas velocidades. Por ter sido criado com vistas ao mercado de videografismo, é nesse meio em que ele é mais difundido atualmente. Câmeras de vídeo profissionais tem interfaces *FireWire*, que permitem a transferência direta de seus dados para outro dispositivo (um computador, por exemplo), em velocidades superiores ao tempo real do vídeo. Na verdade, nem é necessário a existência de um computador em uma das pontas da comunicação; uma câmera pode até mesmo se conectar, via *FireWire*, diretamente a um monitor.

Um barramento *FireWire* pode ser utilizado para conectar até 63 dispositivos ao mesmo tempo. A topologia física da rede, no entanto, deve ser sempre acíclica. O cabo utilizado no *FireWire* 1394a (utilizado no presente trablho) é blindado, e pode ter 4 ou 6 pinos; no de 4 pinos, utilizando um conector RJ-45, eles são entrelaçados dois a dois, um par para envio, o outro para recebimento; no de 6 pinos, temos dois pinos adicionais utilizados para a energização de dispositivos que não possuam alimentação própria. A distância máxima entre os dispositivos pode variar entre 4,5 e 14 metros, inversamente proporcional à velocidade de comunicação utilizada. O *FireWire* 1394b, no entanto, promete distâncias máximas de 100 metros ao se utilizar fibra ótica plástica; distâncias ainda maiores poderão ser alcançadas com fibras óticas de melhor qualidade.

5.3 Camada de acesso ao meio físico

Como mencionado na seção anterior, tanto a interface da rede de trabalho quanto a interface da rede de controle utilizam o mesmo meio físico, o *FireWire*. Ambas, na verdade, necessitavam do mesmo serviço: o *acesso ao meio físico*. Surgiu naturalmente a idéia de criar uma nova camada que oferecesse tal serviço, encapsulando dentro dela todos os detalhes de implementação do acesso a um determinado meio de comunicação.

A estrutura lógica do sistema operacional permaneceu idêntica, com a mesma divisão hierárquica de camadas (interface de sistema, interface da rede de trabalho, interface do núcleo e interface da rede de controle). A única alteração foi a inclusão da nova camada, e o fato de que tanto a interface da rede de trabalho quanto a interface da rede de controle passaram a fazer uso dos serviços oferecidos pela camada de acesso ao meio físico (figura 34). Os serviços dessa camada são oferecidos através dos seguintes operadores:

_send (link, size, buffer)

_receive (link, size, buffer)

_receiveAny (link, size, buffer)

O operador *_send* é utilizado pelo operador da rede de trabalho *s_Send* para enviar mensagens de um nó de trabalho a outro, através do *FireWire*. O operador *ks_Send*, da rede de controle, também o utiliza para responder a uma requisição feita ao nó de controle por um nó de trabalho, requisições estas enviadas pelo operador *kc_SendReceive*, que também utiliza o operador *_send*.

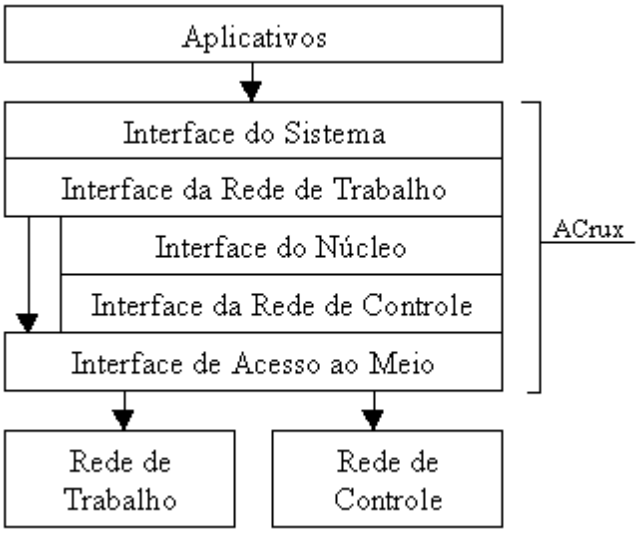

Figura 34: A camada de acesso ao meio físico

O operador *_receive* é utilizado pelo operador da rede de trabalho *s_Receive* para receber, pelo *FireWire*, uma mensagem enviada de outro nó de trabalho. O operador da rede de controle kc_SendReceive também o utiliza para aguardar a resposta do nó de controle a um pedido realizado por um nó de trabalho.

O operador *_receiveAny* é utilizado por aqueles operadores que necessitam receber mensagens de quaisquer nós integrantes do *cluster*. É o caso do operador da rede de controle *ks_ReceiveAny*, que o utiliza para que o nó de controle possa receber requisições de quaisquer nós de trabalho que necessitem de seus serviços.

Com a introdução dessa nova camada, torna-se simples portar o código do sistema operacional *ACrux* de um meio físico para outro, de maneira que as camadas superiores não necessitem de alterações. Ela também revelou-se a solução de um problema causado pela existência de trabalhos paralelos em andamento: enquanto este trabalho se ocupava da implementação das camadas do sistema operacional, o estudo da tecnologia *FireWire* ficou a cargo de outro trabalho. A camada de acesso ao meio físico foi a solução ideal para resolver o problema da divisão de tarefas, servindo de interface entre os dois trabalhos. Neste, foram definidos os operadores necessários à camada de acesso ao meio físico; o outro ficou com a tarefa de implementar tais operadores.

Em implementações futuras do *Crux* está prevista a volta às especificações originais, onde a rede de trabalho utiliza uma rede dinâmica comutada e a rede de serviço utiliza um barramento. Acredita-se que a existência da camada de acesso ao meio físico também irá facilitar essa transição. Bastaria implementar, nessa camada, o suporte a diferentes protocolos, e elaborar um método pelo qual as camadas superiores possam informar qual

protocolo desejam utilizar. Ou, alternativamente, implementar várias camadas físicas, e ligar estaticamente as camadas superiores a um determinado meio físico. Na verdade, demonstra-se que qualquer meio físico que suporte a implementação das chamadas *_send*, *_receive* e *_receiveAny*, mantendo sua semântica bloqueante, pode ser utilizado na arquitetura *Crux*.

5.4 *ACrux* sobre TCP/IP

Como mencionado na seção anterior, haviam projetos paralelos em andamento. Enquanto neste se trabalhava na implementação das camadas superiores do *ACrux*, outro implementava a camada de acesso ao meio físico. Por isso, essa camada não esteve disponível durante a maior parte do desenvolvimento deste projeto. Para suprir sua inexistência durante os trabalhos iniciais de implementação das camadas superiores, foi construído um simulador através da utilização de *sockets*. Pode-se considerar, no entanto, que não foi implementado somente um simulador, mas sim uma outra camada de acesso ao meio físico, que possibilita a utilização de redes TCP/IP.

Essa camada de acesso ao TCP/IP, além de haver auxiliado nos trabalhos de desenvolvimento do sistema operacional *ACrux*, permitiu montar uma máquina *Crux* completa, com nós de trabalho, de controle, e servidor de comunicações, tudo isso sobre uma rede de computadores comuns interligados por TCP/IP. A rede TCP/IP substitui tanto a rede de controle quanto a rede de trabalho, como pode-se observar na figura 35. É uma alternativa viável, na falta de uma máquina *Crux* interligada por redes de alto desempenho. E mesmo que tal máquina esteja disponível, o *ACrux* sobre TCP/IP pode ser uma ferramenta destinada a ajudar na implementação e depuração de programas paralelos. Pode-se validar o programa em um ambiente real de execução, apenas de menor desempenho. Após validado, o programa pode ser levado para a máquina *Crux* de alto desempenho, possuindo-se já a certeza de que ele está correto. Pode-se, por fim, utilizá-lo como base para comparar o desempenho obtido com a utilização de outros meios físicos. O *ACrux* sobre TCP/IP pode ser considerada um dos subprodutos deste trabalho.

5.5 Suporte a novos protocolos de comunicação

Existe, no *ACrux*, a necessidade de suportar operadores de comunicação adicionais. Todos os operadores mencionados até aqui são de uso restrito, destinados à comunicação entre os processos de usuário e o sistema operacional. A interface da rede de controle,

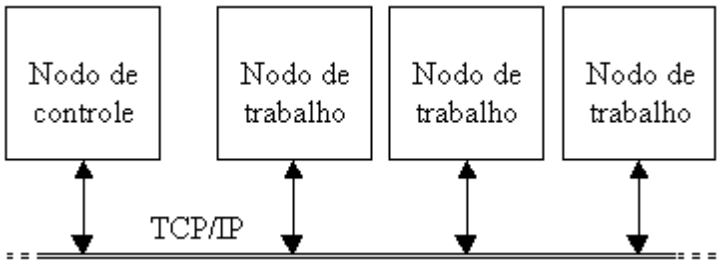

Figura 35: Crux sobre um barramento TCP/IP

por exemplo, permite a comunicação dos processos que executam na rede de trabalho com o nó de controle. E a interface da rede de trabalho permite a comunicação entre nós de trabalho, mas ela é destinada a implementar a interface de sistema do *ACrux*, e tem características que a tornam inadequada à grande parte dos programas paralelos que se deseja executar em uma máquina *Crux*.

A principal característica da interface da rede de trabalho é a de que ela não cria conexões duradouras entre os nós de trabalho. Pode-se constatar isso observando a implementação do operador da rede de trabalho *s_Send* (apresentada anteriormente na figura 32): a primeira tarefa por ele realizada é o estabelecimento de uma conexão através do operador de núcleo *Connect*; a última tarefa é o rompimento dessa conexão, através do operador do núcleo *Disconnect*. Isso pode ser adequado na comunicação de processos com servidores do sistema, onde se deseja a comunicação direta e é necessário, portanto, sincronizar os processos a cada mensagem enviada. Pode não servir, no entanto, para uma situação em que processos paralelos cooperam na realização de alguma tarefa que exija intensa comunicação entre eles. Boa parte do tráfego na rede seria constituído por pedidos de conexão e desconexão, e o desempenho da aplicação seria comprometido.

É por isso que o *ACrux* deve prever mecanismos que possibilitem a extensão dos operadores de comunicação por ele suportados. Estão previstos a implementação de operadores de comunicação como os utilizados pela linguagem *SuperPascal* [35], e pelas bibliotecas *PVM* e *MPI*, mais adequados à construção de programas paralelos do que os operadores da rede de trabalho do *Crux*. O tratamento de outros protocolos de comunicação está previsto, neste trabalho, através de modificações na interface da rede de controle e no formato da mensagem que trafega por essa rede, sem esquecer do servidor de comunicações. Basicamente, a alteração consiste em adicionar um campo à mensagem enviada entre os nós de trabalho e o nó de controle, campo esse capaz de indicar ao servidor de comunicações a qual protocolo a mensagem se refere. O nó de controle, ao receber a mensagem, apenas verifica o campo referente ao protocolo, ignorando o resto da mensagem (mesmo

porque ele ainda não conhece seu formato). Ao identificar o protocolo a que pertence a mensagem, o servidor de comunicações aciona a função tratadora adequada, repassando a ela a mensagem. Cada tratador conhece o formato de mensagem de apenas um dos protocolos, sendo ele o único capaz de decodificá-la e tratá-la corretamente.

Partiu-se, portanto, do princípio, que as futuras extensões do *ACrux* serão implementadas como outras camadas do sistema operacional, e utilizarão, para se comunicarem com o nó de controle, a mesma interface da rede de controle utilizada pela já existente interface do núcleo. A figura 36 descreve esquematicamente a interação das extensões do sistema com o restante do sistema operacional.

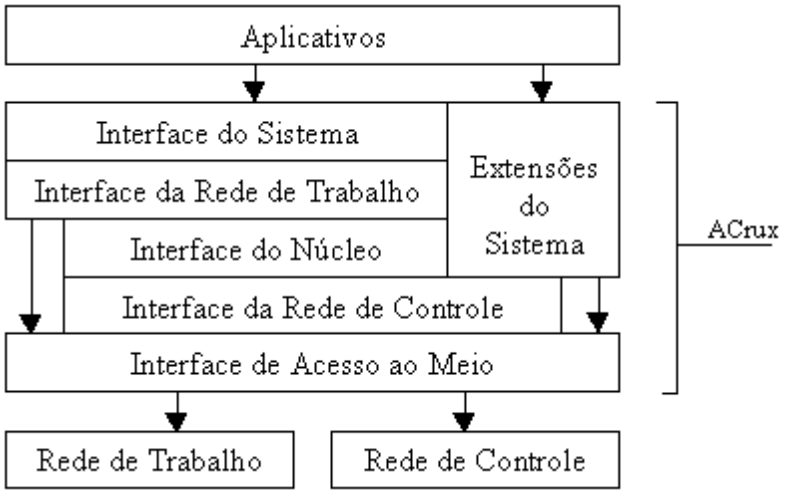

Figura 36: Prevendo extensões ao sistema operacional *ACrux*

5.6 Tratamento de sinais

Outro ponto levado em consideração na implementação do núcleo do sistema operacional *ACrux* foi a troca de *sinais* entre os processos do sistema. Os sinais, no ambiente *Unix*, são um mecanismo de comunicação entre processos, através do qual eles podem enviar mensagens uns aos outros, com fins de, por exemplo, sincronização.

No ambiente *Crux*, a troca de sinais entre os processos é um pouco mais complexa, por estarmos tratando com um sistema operacional distribuído. Os processos que desejam se comunicar estão situados em máquinas diferentes, e os sinais devem ser transmitidos pelo meio de comunicação que os interliga. Mas a real dificuldade é o fato dos sinais serem eventos externos ao fluxo de execução dos programas que os recebem. A comunicação direta exigida pelas mensagens de sistema do *ACrux*, que exige a prévia concordância de emissor e receptor antes da comunicação ter lugar, impede que um processo envie uma

mensagem diretamente a a outro avisando que está lhe sinalizando; ambos os processos devem antes se comunicar com o nó de controle. O emissor irá pedir ao nó de controle permissão para se comunicar com o receptor, mas o receptor não tem como saber o momento em que alguém deseja lhe enviar um sinal, não podendo então requisitar uma conexão.

A solução encontrada é fazer uso dos servidores do sistema para que os processos possam trocar sinais entre si. O nó emissor, ao executar, por exemplo, uma chamada *kill*, capaz de enviar um sinal a outro processo, irá se comunicar com um dos servidores do sistema, que irá armazenar o sinal e a quem ele se destina. Na próxima vez que o processo a que se destina o sinal se comunicar com esse servidor, ele receberá, juntamente com a resposta à sua requisição, o sinal enviado a ele. Isso significa que, em toda mensagem trocada entre um nó de trabalho e um servidor do sistema, deve haver um local reservado para os sinais. Como toda interação entre nó e servidor utiliza a interface da rede de trabalho, é nessa camada que foi reservado um espaço para o transporte dos sinais. A implementação em si do mecanismo de troca de sinais ficará a cargo de trabalhos futuros.

5.7 Resumo

Este capítulo se dedicou a comentar todo o trabalho final de implementação do núcleo do sistema operacional *ACrux* e seu servidor de comunicações. Apenas um exemplo do código produzido, o operador da interface da rede de trabalho *s_Send*, foi explicado em detalhes. Os pontos de interesse que foram ressaltados em sua implementação foram a utilização da interface de acesso ao núcleo e da interface de acesso ao meio físico; a coleta de estatísticas, dinamicamente ativada ou desativada; e a utilização de operadores do núcleo do *Linux*.

A plataforma utilizada para a implementação da arquitetura *Crux* também foi detalhada. Na ausência, por motivos financeiros, de um *crossbar*, a rede de trabalho foi implementada sobre uma rede do tipo *firewire*, capaz de oferecer taxas de transferência de 400 Mbps e tratar a nível de *hardware* o roteamento de mensagens. A rede de controle também foi mapeada nesse mesmo meio físico, para aproveitar o grande desempenho por ele oferecido.

Outra alteração na especificação original da arquitetura *Crux* foi a implementação de uma camada extra no núcleo do sistema operacional *ACrux*. Foi criada a *camada de acesso ao meio físico*, capaz de oferecer, em um exemplo prático, acesso à rede *FireWire* tanto à interface da rede de controle quanto à interface da rede de trabalho. Ela possibilita

também maior facilidade na hora de portar o *Crux* para outros meios físicos, sendo que as alterações necessárias no sistema operacional se concentrariam nessa camada mais inferior, sem se propagar para o restante do *software*.

Durante o desenvolvimento do trabalho, a camada de acesso ao meio físico do *FireWire* ainda não havia sido terminada. Uma camada de acesso ao meio físico substituta foi elaborada, utilizando os protocolos TCP/IP. Ou seja, com ela, pode-se montar uma máquina *Crux* sobre qualquer rede TCP/IP, utilizando-a para, por exemplo, validar programas paralelos ou realizar medições de desempenho.

Finalmente, o *ACrux* foi preparado para receber extensões a seus operadores atuais. Tais operadores são adequados às necessidades do sistema operacional, mas podem não ser às necessidades de certas aplicações a nível de usuário. Por esse motivo, a implementação do sistema operacional *ACrux* resultante desse trabalho já foi modelada levando em conta as futuras extensões que a ele serão adicionadas. Também foi previsto a necessidade de enviar sinais entre os processos do sistema; a solução adotada foi reservar, na interface da rede de trabalho, um espaço, para que toda mensagem trocada entre os nós por essa interface possa servir também para que os processos recebam sinais a eles destinados.

6 Conclusão

O presente trabalho, no âmbito do projeto *Crux*, se propunha a implementar o núcleo de um sistema operacional hierárquico e distribuído, denominado *ACrux*. Os passos necessários para sua implementação foram uma revisão da bibliografia na área de arquiteturas paralelas, um estudo da arquitetura *Crux* e o aprofundamento dos conhecimentos sobre o núcleo do sistema operacional *Linux*, todos eles narrados nos diversos capítulos desta dissertação.

A partir desse estudo e dos trabalhos de implementação, o produto resultante desse projeto foi um módulo carregável do *Linux*, capaz de, quando instalado, adicionar ao núcleo do sistema operacional carregado na memória, automaticamente e de forma transparente, todas as camadas que compõem o *ACrux*. Essas camadas estão acessíveis aos programas de usuário na forma de chamadas de sistema, que eles podem utilizar para se intercomunicar.

Outro importante componente da arquitetura *Crux* implementado no decorrer deste projeto foi o servidor de comunicações. Executando no nó de controle, ele é o responsável por coordenar todos os outros nós do *cluster*. Foi implementado como um processo de usuário que, utilizando as chamadas de sistema do núcleo do *ACrux*, é capaz de se comunicar com os nós de trabalho e atender às suas requisições.

Um terceiro subproduto deste trabalho foi a implementação de uma camada de acesso ao meio físico capaz de fazer com que a arquitetura *Crux* possa executar utilizando como meio de comunicação qualquer rede TCP/IP. Ela foi elaborada para auxiliar na tarefa de implementação do núcleo do *ACrux* e do servidor de comunicações, mas pode encontrar utilidade mesmo após terminado esse ciclo de desenvolvimento. Pode ser utilizada para exemplificar o funcionamento da arquitetura *Crux* em uma rede comum, de baixo desempenho, ou ainda para validar programas paralelos antes de sua execução em um ambiente de alto desempenho.

E um último subproduto que se pode considerar importante é a pequena compilação de técnicas de programação no núcleo do *Linux* apresentada neste trabalho. É o tipo

de informação difícil de se encontrar nessa forma, agrupada e validada; o mais comum é encontrá-la dispersa na *internet*, e ainda com baixo grau de confiabilidade. Espera-se que tais informações sejam de grande utilizada para futuros membros do projeto *Crux*, ou mesmo para outros leitores interessados no assunto.

Considerando todos esses resultados, conclui-se que os objetivos do projeto foram alcançados e, pode-se dizer, superados. Ao fim dos trabalhos, temos condições totais de executar o sistema operacional *ACrux*, com seus servidor de comunicações, sobre uma rede TCP/IP. Com o estudo do *FireWire*, tema de outra dissertação, logo teremos condições de executar o *Crux* sobre uma rede de grande desempenho. Ainda existe, no entanto, muito trabalho a ser feito.

Para trabalhos futuros o próprio núcleo do sistema operacional *ACrux* oferece oportunidades. A interface do núcleo teve implementados seus operadores relativos à conexão e desconexão de canais da rede de trabalho; existem ainda os operadores relacionados à alocação e desalocação de nós de trabalho.

Pode-se citar também a remoção do sistema operacional *Linux* dos nós de trabalho. Eles não são máquinas de uso geral, e não necessitam de um sistema operacional completo e complexo como o *Linux* para executarem suas tarefas. Os únicos dispositivos de entrada e saída de que eles necessitam são suas interfaces de rede; nem mesmo teclados ou monitores são necessários. A construção de um pequeno sistema operacional traria benefícios como o aumento do desempenho dos nós; tendo em vista que seu único uso será no ambiente *Crux*, isso abre espaço para a possibilidade de otimizações.

Outros trabalhos futuros são as já mencionadas extensões aos operadores de comunicação do *ACrux*. Os operadores implementados neste trabalho são adequados ao sistema, mas aplicações específicas podem requerer comportamentos diferentes. Extensões que possibilitem a utilização de bibliotecas como *PVM* e *MPI*, ou ainda de linguagens como *SuperPascal* e *Joyce*, por exemplo, são algumas das possibilidades existentes.

Outra fonte inesgotável de trabalho pode ser a recém-criada camada de acesso ao meio físico. Uma das primeiras tarefas pode ser sua adaptação para o meio físico de um *crossbar*, voltando às especificações originais do *Crux*. Mas, existindo o interesse, ela pode ser mapeada para outros meios físicos, como, por exemplo, *SCSI*, *USB*, *Fibre Channel*, entre outros.

Finalmente, existem diversos *softwares* desenvolvidos com vistas a serem executados na plataforma *Crux*. Tais *softwares* foram testados em simuladores da arquitetura, e já podem agora ser portados para a máquina real. Entre os *softwares* existentes, destacam-se

o servidor de arquivos e compiladores e interpretadores da linguagem *SuperPascal*.

APÊNDICE A – Camada de acesso à rede de trabalho

```
////////////////////////////////////////////////////////
//
// Karlos H. Budag
// Maio / 2002
// Interface da rede de trabalho (WrkNet.c)
//
////////////////////////////////////////////////////////

/* Declare what kind of code we want from the header files */
#define __KERNEL__          /* We're part of the kernel */ #define
MODULE              /* Not a permanent part, though. */

//__NR_...
#include <asm/unistd.h>

/* Standard headers for LKMs */
#include <linux/modversions.h>
#include <linux/module.h>

#define _LOOSE_KERNEL_NAMES
    /* With some combinations of Linux and gcc, tty.h will not compile if
       you don't define _LOOSE_KERNEL_NAMES.  It's a bug somewhere.
    */
#include <linux/tty.h>      /* console_print() interface */

////////////////////////////////////

#include "global.h"
#include "WrkNet.h"
#include "KrnlInt.h"
#include "CtrlNet.h"

//////////////////////////////////////

extern void *sys_call_table[]; asmlinkage int
(*original_call)(const char *, int, int); int _crux_chamadas = -1;

//////////////////////////////////////////////////////

void setarChamadas() {
  _socketcall = sys_call_table[__NR_socketcall];
```

```
  _close = sys_call_table[__NR_close];
  _fcntl = sys_call_table[__NR_fcntl];
  _nanosleep = sys_call_table[__NR_nanosleep];
  _yield = sys_call_table[__NR_sched_yield];
}

int encontreChamadaLivre ( int num ) {
  int i = 255;
  void *ultimaChamada = 0;
  int indiceChamadaLivre = -1;

  ultimaChamada = sys_call_table[i];

  for ( i--; i >= 0; i-- )
  {
    if ( sys_call_table[i] != ultimaChamada )
    {
      indiceChamadaLivre = i+1;
      break;
    }
  }

  // Nao foram encontradas chamadas livres suficientes
  if ( (256 - num) < indiceChamadaLivre )
    return -1;

  return indiceChamadaLivre;
}


/* Initialize the module */ int init_module() {
  int i = 0;

  setarChamadas();

  i = encontreChamadaLivre ( 3 );
  if ( i < 0 )
  {
    printk ( "WrkCrux: ERRO - Uma chamada de sistema livre não foi encontrada\n" );
    return -1;
  }

  _crux_chamadas = i;
  original_call = sys_call_table[_crux_chamadas];
  sys_call_table[_crux_chamadas]   = sys_s_Send;
  sys_call_table[_crux_chamadas+1] = sys_s_Receive;
  sys_call_table[_crux_chamadas+2] = sys_ss_ReceiveAny;

  printk ( "WrkCrux: As novas chamadas foram instaladas a partir da posição %d\n",
    _crux_chamadas );

  /* If we return a non zero value, it means that
```

```
   * init_module failed and the kernel module
   * can't be loaded */
  return 0;
}


/* Cleanup - undid whatever init_module did */ void
cleanup_module() {
  if ( _crux_chamadas < 0 )
  {
    printk (
      "WrkCrux: ERRO - As chamadas de sistema não foram encontradas na tabela\n" );
    return;
  }

  sys_call_table[_crux_chamadas]   = original_call;
  sys_call_table[_crux_chamadas+1] = original_call;
  sys_call_table[_crux_chamadas+2] = original_call;

  printk ( "WrkCrux: As chamadas de sistema foram desinstaladas\n" );
}
```

```
////////////////////////////////////////////////
//
// Karlos H. Budag
// Maio / 2002
// Interface da rede de trabalho (WrkNet.h)
//
////////////////////////////////////////////////

int _wrk_links[_MAX_NODES];
int _wrk_pid[_MAX_NODES];

asmlinkage int sys_s_Send ( int thisProc, int proc, void *msg,
 int length );
asmlinkage int sys_s_Receive ( int thisProc, int proc, void *msg,
 int *length );
asmlinkage int sys_ss_ReceiveAny ( int thisProc, int *proc, void *msg,
 int *length );
asmlinkage int sys_s_GetData ( int node, int *data );
asmlinkage int sys_s_GetId ( void );

void InitWrkNet ( int node );
void ResetWrkNet ( int node );
```

APÊNDICE B – Camada do núcleo

```
////////////////////////////////////////
//
// Karlos H. Budag
// Maio / 2002
// Interface do núcleo (KrnlInt.c)
//
////////////////////////////////////////

/* Declare what kind of code we want from the header files */
#define __KERNEL__          /* We're part of the kernel */ #define
MODULE              /* Not a permanent part, though. */

#include <linux/modversions.h>
#include <linux/tty.h>      /*
console_print() interface */

////////////////////////////////////////////

#include "global.h"
#include "CtrlNet.h"
#include "KrnlInt.h"

//////////////////////////////

int _initKrnlInt_flag = 0;

void InitKrnlInt ( int node ) {
  int i = 0;
  int ii = 0;

  if ( ! _initKrnlInt_flag )
  {
    _initKrnlInt_flag = 1;

    for ( ; i < _MAX_NODES; i++ )
    {
      for ( ii = 0; ii < _MAX_NODES; ii++ )
        _krnl_link[i][ii] = -1;
    }
  }

  InitCtrlLink ( node );

  _print2 ( "InitKrnlInt: Inicializando a interface do nucleo do nodo %d\n", node );
}
```

```
void ResetKrnlInt ( int node ) {
  int i = 0;

  if ( ! _initKrnlInt_flag )
    return;

  for ( ; i < _MAX_NODES; i++ )
    _krnl_link[node][i] = -1;
  for ( i = 0; i < _MAX_NODES; i++ )
    _krnl_link[i][node] = -1;

  _print2 ( "ResetKrnlInt: Finalizando a interface do nucleo do nodo %d\n", node );

  ResetCtrlLink ( node );
}

//////////////////////////////

int Connect ( int thisProc, int proc, int *link ) {
  int op = 0;
  int res = 0;
  int srvProc = 0;
  CTRLBUF msg, ret;

  //Zera a conexao
  if ( _krnl_link[thisProc][proc] != -1 )
    _krnl_link[thisProc][proc] = -1;

  *link = -1;
  MakeKcMsg ( &msg, kc_CONNECT, thisProc, proc );
  res = sys_kc_SendReceive ( thisProc, &msg, &ret );
  if ( res < 0 )
  {
    *link = -1;
    return -1;
  }

  UnmakeKcMsg ( &ret, &op, &srvProc, link );
  _print3 ( "Connect: Conexao entre nodos %d e %d estabelecida pelo link %d\n",
    thisProc, proc, *link );

  _krnl_link[thisProc][proc] = *link;
  return 0;
}

int ConnectAny ( int thisProc, int *proc, int *link ) {
  int op = 0;
  int res = 0;
  int param1 = 0;
  int param2 = 0;
  CTRLBUF msg, ret;
```

```
  *proc = 0;
  *link = -1;

  MakeKcMsg ( &msg, kc_CONNECT_ANY, thisProc, 0 );
  res = sys_kc_SendReceive ( thisProc, &msg, &ret );
  if ( ret < 0 )
  {
    *proc = -1;
    *link = -1;
    return -1;
  }

  UnmakeKcMsg ( &ret, &op, &param1, &param2 );
  if ( param2 < 0 )
    return -1;

  *proc = param2;
  res = Connect ( thisProc, *proc, link );

  return res;
}

int Disconnect ( int thisProc, int proc ) {
  int res = 0;
  CTRLBUF msg;
  CTRLBUF ret;

  //Jah esta desconectado
  if ( _krnl_link[thisProc][proc] == -1 )
    return 0;

  MakeKcMsg ( &msg, kc_DISCONNECT, thisProc, proc );
  res = sys_kc_SendReceive ( thisProc, &msg, &ret );

  _krnl_link[thisProc][proc] = -1;
  return res;
}
```

```
//////////////////////////////////////
//
// Karlos H. Budag
// Maio / 2002
// Interface do núcleo (KrnlInt.h)
//
//////////////////////////////////////

int _krnl_link[_MAX_NODES][_MAX_NODES];

void InitKrnlInt ( int node );
void ResetKrnlInt ( int node );

int Connect ( int thisProc, int proc, int *link );
int ConnectAny ( int thisProc, int *proc, int *link );
int Disconnect ( int thisProc, int proc );
```

APÊNDICE C – Camada de acesso à rede de controle

```
////////////////////////////////////////////////////
//
// Karlos H. Budag
// Maio / 2002
// Interface da rede de controle (CtrlNet.c)
//
////////////////////////////////////////////////////

/* Declare what kind of code we want from the header files */
#define __KERNEL__          /* We're part of the kernel */ #define
MODULE              /* Not a permanent part, though. */

//get_fs(), copy_from_user()...
#include <asm/uaccess.h>

//GFP_KERNEL
#include <linux/slab.h>

#include <linux/modversions.h>
#include <linux/tty.h>      /*
console_print() interface */

//////////////////////////////////////////

#include "global.h"
#ifdef _meiofisico_tcp_
#include
"crux_sockets.h"
#else
#include <sync1394/sync1394_mod.h>
#endif
#include "CtrlNet.h"

//////////////////////////////////////

int _srvpid = -1; int _initCtrlNet = 1; int _srv_initCtrlNet = 1;
//int _serverAddr = 0;

////////////////////////////////

//Descobre a que protocolo se refere uma mensagem
void UnmakeProt ( CTRLBUF *msg, int *prot ) {
  int i = 0;
```

```
  int op, param1, param2;
  int len = sizeof ( *prot );
  _memcpy ( prot, &(*msg)[i], len );

  //Soh pra debugar o valor do buffer
  UnmakeKcMsg ( msg, &op, &param1, &param2 );
}

//Monta um buffer com uma mensagem do sistema
void MakeKcMsg ( CTRLBUF *msg, int op, int param1, int param2 ) {
  int i = 0;
  int prot = kc_PROT_SYSTEM;
  int len = sizeof ( prot );

  printk (
    "<<<<<<<<<<<<<>>>>>>>>>>>> MakeKcMsg: prot %d, op %d, param1 %d, param2 %d\n",
    prot, op, param1, param2 );
  _memcpy ( &(*msg)[i], &prot, len );

  len = sizeof ( op );
  i = i + len;
  _memcpy ( &(*msg)[i], &op, len );

  len = sizeof ( param1 );
  i = i + len;
  _memcpy ( &(*msg)[i], &param1, len );

  len = sizeof ( param2 );
  i = i + len;
  _memcpy ( &(*msg)[i], &param2, len );
}

void UnmakeKcMsg ( CTRLBUF *msg, int *op, int *param1, int *param2
) {
  int i = 0;
  int prot = 0;
  int len = sizeof ( prot );
  _memcpy ( &prot, &(*msg)[i], len );

  if ( prot != kc_PROT_SYSTEM )
    return;

  len = sizeof ( *op );
  i = i + len;
  _memcpy ( op, &(*msg)[i], len );

  len = sizeof ( *param1 );
  i = i + len;
  _memcpy ( param1, &(*msg)[i], len );

  len = sizeof ( *param2 );
  i = i + len;
```

```
  _memcpy ( param2, &(*msg)[i], len );

  printk (
    "<<<<<<<<<<>>>>>>>>>> UnmakeKcMsg: prot %d, op %d, param1 %d, param2 %d\n",
    prot, *op, *param1, *param2 );
}

//Monta um buffer com uma mensagem do SuperPascal
void MakeSpMsg ( CTRLBUF *msg ) {
  int i = 0;
  int prot = kc_PROT_SUPERPASCAL;
  int len = sizeof ( prot );
  _memcpy ( &(*msg)[i], &prot, len );
}

void UnmakeSpMsg ( CTRLBUF *msg ) {
  int i = 0;
  int prot = 0;
  int len = sizeof ( prot );
  _memcpy ( &prot, &(*msg)[i], len );

  if ( prot != kc_PROT_SUPERPASCAL )
    return;
}

//Monta um buffer com uma mensagem do PVM
void MakePvmMsg ( CTRLBUF *msg ) {
  int i = 0;
  int prot = kc_PROT_PVM;
  int len = sizeof ( prot );
  _memcpy ( &(*msg)[i], &prot, len );
}

void UnmakePvmMsg ( CTRLBUF *msg ) {
  int i = 0;
  int prot = 0;
  int len = sizeof ( prot );
  _memcpy ( &prot, &(*msg)[i], len );

  if ( prot != kc_PROT_PVM )
    return;
}

//Monta um buffer com uma mensagem do MPI
void MakeMpiMsg ( CTRLBUF *msg ) {
  int i = 0;
  int prot = kc_PROT_MPI;
  int len = sizeof ( prot );
  _memcpy ( &(*msg)[i], &prot, len );
}

void UnmakeMpiMsg ( CTRLBUF *msg ) {
```

```
  int i = 0;
  int prot = 0;
  int len = sizeof ( prot );
  _memcpy ( &prot, &(*msg)[i], len );

  if ( prot != kc_PROT_MPI )
    return;
}

////////////////////////////////////////////////////

void InitCtrlLink ( int node ) {
  int i = 0;
  int novoSocket = 0;

  //Primeira inicializacao, ao carregar o modulo
  if ( _initCtrlNet )
  {
    _initCtrlNet = 0;
    for ( ; i < _MAX_NODES; i++ )
      _ctrl_links[node].outLink = -1;
  }

  //Conectando o processo ao servidor
#ifdef _meiofisico_tcp_
  novoSocket = conectarSocket ( _serverAddr, SRVNODE_PORT+node, 0 );
  _ctrl_links[node].outLink = novoSocket;
#else
  sync1394_reset_1394();
  _ctrl_links[node].outLink = sync1394_get_control_1394id();
#endif

  _print3 ( "InitCtrlLink: Iniciando a rede de controle do nodo %d, outLink %d\n",
    node, _ctrl_links[node].outLink );
}

void ResetCtrlLink ( int node ) {
  if ( _initCtrlNet )
    return;

  if ( _ctrl_links[node].outLink != -1 )
  {
#ifdef _meiofisico_tcp_
    _crux_close ( _ctrl_links[node].outLink );
#endif
    _ctrl_links[node].outLink = -1;
  }

  _print2 ( "ResetCtrlLink: Finalizando a rede de controle do nodo %d\n", node );
}

void SrvInitCtrlNet() {
```

```
  int i = 0;

  _srv_initCtrlNet = 0;
#ifndef _meiofisico_tcp_
  sync1394_reset_1394();
#endif
  _srvpid = _crux_getpid();

  for ( i = 1; i < _MAX_NODES; i++ )
  {
#ifdef _meiofisico_tcp_
    _ctrl_links[i].sockAcc = criarSocket ( SRVNODE_PORT + i, 1, 0 );
#else
    _ctrl_links[i].sockAcc = -1;
#endif
    _ctrl_links[i].inLink = -1;
    _ctrl_links[i].addr = -1;

    _print4 (
      "SrvInitCtrlNet: iniciando rede ctrl - nodo %d, sockAcc %d, _srvpid %d\n",
      i, _ctrl_links[i].sockAcc, _srvpid );
  }
}

void SrvResetCtrlNet() {
  int i = 0;

  _srv_initCtrlNet = 1;
  _srvpid = -1;

  for ( ; i < _MAX_NODES; i++ )
  {
    if ( _ctrl_links[i].inLink != -1 )
    {
#ifdef _meiofisico_tcp_
      _crux_close ( _ctrl_links[i].inLink );
#endif
      _ctrl_links[i].inLink = -1;
    }

    if ( _ctrl_links[i].sockAcc != -1 )
    {
#ifdef _meiofisico_tcp_
      _crux_close ( _ctrl_links[i].sockAcc );
#endif
      _ctrl_links[i].sockAcc = -1;
      _ctrl_links[i].addr = -1;
    }
  }

  _print1 ( "SrvResetCtrlLink: Nodo de controle finalizando a rede de controle\n" );
}
```

```
////////////////////////////////////////////////

asmlinkage int sys_ks_Send ( int node, CTRLBUF *msg ) {
  int link = -1;
  int length = sizeof ( *msg );
  CTRLBUF kbuf;

  if ( ! _srv_initCtrlNet )
  {
    if ( _srvpid != _crux_getpid() )
      SrvResetCtrlNet();
  }

  if ( _srv_initCtrlNet )
    SrvInitCtrlNet();

  if ( _ctrl_links[node].inLink == -1 )
    return -1;

  copy_from_user ( kbuf, msg, sizeof(CTRLBUF) );
  link = _ctrl_links[node].inLink;
#ifdef _meiofisico_tcp_
  tcp_send ( link, length, msg );
#else
  printk ( "<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>> ks_Send: antes sync1394_send %d,
    %d\n", length, link );
  sync1394_send ( msg, length, link );
  printk ( "<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>> ks_Send: depois sync1394_send\n" );
#endif

  return 0;
}

asmlinkage int sys_ks_ReceiveAny ( int *node, CTRLBUF *msg ) {
  int i = 0;
  int ret = 0;
  int link = -1;
  int addr = 0;
  int novoSocket = 0;
  int length = sizeof ( CTRLBUF );
  CTRLBUF kbuf;
  CTRLBUF *pkbuf = &kbuf;
  int knode;
  int *pKnode = &knode;

  if ( ! _srv_initCtrlNet )
  {
    if ( _srvpid != _crux_getpid() )
      SrvResetCtrlNet();
  }
```

```
  if ( _srv_initCtrlNet )
    SrvInitCtrlNet();

#ifdef _meiofisico_tcp_
  while ( ++ i )
  {
    if ( i >= _MAX_NODES )
    {
      i = 1;

      ret = _crux_yield();

      //0.2 seg, tempo minimo para que o processo perca a CPU
//      ret = _crux_nanosleep ( 2000000L );
      if ( ret < 0 )
      {
        _print2 ( "ks_ReceiveAny: sinal interrompendo a execucao - %d\n", ret );
        return ret;
      }
    }

    //Esse nodo jah se conectou
    if ( _ctrl_links[i].inLink != -1 )
    {
      knode = _ctrl_links[i].addr;
      link = _ctrl_links[i].inLink;
      ret = tcp_receiveany ( link, length, kbuf );
      if ( ret > 0 )
        break;

      //Essa conexao foi fechada
      if ( ret < 0 )
      {
        _crux_close ( link );
        _ctrl_links[i].inLink = -1;
      }
    }

    //Esse nodo ainda nao se conectou
    else if ( _ctrl_links[i].sockAcc != -1 )
    {
      novoSocket = acceptSocket ( _ctrl_links[i].sockAcc, &addr );

      if ( novoSocket >= 0 )
      {
        _ctrl_links[i].inLink = novoSocket;
        _ctrl_links[i].addr = addr;
      }
    }
  }
#else
  printk (
```

```
    "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ks_ReceiveAny: antes sync1394_receive_any %d\n",
    sizeof(CTRLBUF) );
  ret = sync1394_receive_any ( (void*)pkbuf, sizeof(CTRLBUF), &pKnode );
  printk ( "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< conteudo pkbuf %d\n", *(int*)pkbuf );
  printk (
    "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< ks_ReceiveAny: depois sync1394_receive_any %d\n",
    knode );
  _ctrl_links[knode].inLink = knode;
  _ctrl_links[knode].addr = knode;
#endif

  copy_to_user ( node, &knode, sizeof(knode) );
  copy_to_user ( msg, &kbuf, sizeof(kbuf) );
  return 0;
}

asmlinkage int sys_kc_SendReceive ( int thisProc, CTRLBUF *msg,
CTRLBUF *ret ) {
  int length = sizeof ( *msg );
  int link = -1;

  if ( _ctrl_links[thisProc].outLink == -1 )
    return -1;

  link = _ctrl_links[thisProc].outLink;
#ifdef _meiofisico_tcp_
  tcp_send ( link, length, *msg );
#else
  printk ( "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< kc_SendReceive: antes sync1394_send %d,
    %d\n", length, link );
  sync1394_send ( msg, length, link );
  printk ( "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< kc_SendReceive: depois sync1394_send\n" );
#endif

  link = _ctrl_links[thisProc].outLink;
#ifdef _meiofisico_tcp_
  tcp_receive ( link, length, ret );
#else
  printk ( "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< kc_SendReceive: antes sync1394_receive %d,
    %d\n", sizeof(CTRLBUF), link );
  sync1394_receive ( (void*)ret, sizeof(CTRLBUF), link );
  printk (
    "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< kc_SendReceive: depois sync1394_receive\n" );
#endif


  return 0;
}
```

```
////////////////////////////////////////////////
//
// Karlos H. Budag
// Maio / 2002
// Interface da rede de controle (CtrlNet.h)
//
////////////////////////////////////////////////

#define kc_RETURN 0
#define kc_CONNECT 1
#define kc_CONNECT_ANY 2
#define kc_DISCONNECT 3

#define kc_PROT_SYSTEM 0
#define kc_PROT_SUPERPASCAL 1
#define kc_PROT_PVM 2
#define kc_PROT_MPI 3

typedef struct
{
  int inLink;
  int outLink;
  int sockAcc;
  int addr;
} _CTRL_LINKS;

_CTRL_LINKS _ctrl_links[_MAX_NODES];

typedef char CTRLBUF[16];

asmlinkage int sys_ks_Send ( int node, CTRLBUF *msg );
asmlinkage int sys_ks_ReceiveAny ( int *node, CTRLBUF *msg );
asmlinkage int sys_kc_SendReceive ( int thisProc, CTRLBUF *msg, CTRLBUF *ret );

void UnmakeProt ( CTRLBUF *msg, int *prot );
void MakeKcMsg ( CTRLBUF *msg, int op, int param1, int param2 );
void UnmakeKcMsg ( CTRLBUF *msg, int *op, int *param1, int *param2 );
void MakeSpMsg ( CTRLBUF *msg );
void UnmakeSpMsg ( CTRLBUF *msg );
void MakePvmMsg ( CTRLBUF *msg );
void UnmakePvmMsg ( CTRLBUF *msg );
void MakeMpiMsg ( CTRLBUF *msg );
void UnmakeMpiMsg ( CTRLBUF *msg );

void InitCtrlLink ( int node );
void ResetCtrlLink ( int node );
void SrvInitCtrlNet();
void SrvResetCtrlNet();
```

APÊNDICE D – Camada de acesso ao meio físico (TCP/IP)

```
//////////////////////////////////////////////
//
// Karlos H. Budag
// Maio / 2002
// Acesso ao meio fisico (crux_sockets.c)
//
//////////////////////////////////////////////

/* Declare what kind of code we want from the header files */
#define __KERNEL__          /* We're part of the kernel */ #define
MODULE              /* Not a permanent part, though. */

#include <linux/modversions.h>
#include <linux/tty.h>      /*
console_print() interface */

//////////////////////////////////////////////

#include "global.h"
#include "crux_sockets.h"

//////////////////////////////////

int _getId() {
  return ( -1 );
}

void tcp_send ( int link, int size, void *buffer ) {
  //Enviando os dados
  _crux_send ( link, buffer, size, 0 );
  _print3 ( "_out %d, %d\n", link, size );
}

void tcp_receive ( int link, int size, void *buffer ) {
  //Recebendo os dados
  size = _crux_recv ( link, buffer, size, 0 );
  _print3 ( "_in %d, %d\n", link, size );
}

int tcp_receiveany ( int link, int size, void *buffer ) {
  //Recebendo ( nao bloqueante )
  size = _crux_recv ( link, buffer, size, MSG_DONTWAIT );
```

```
  //Nao ha nada para ler
  if ( size < 0 )
    return 0;

  //O socket jah foi fechado
  if ( size == 0 )
    return -1;

  _print3 ( "_anyIn %d, %d\n", link, size );
  return 1;
}
```

```
////////////////////////////////////////////
//
// Karlos H. Budag
// Maio / 2002
// Acesso ao meio fisico (crux_sockets.h)
//
////////////////////////////////////////////

int _getId();
void tcp_send ( int link, int size, void *buffer );
void tcp_receive ( int link, int size, void *buffer );
int tcp_receiveany ( int link, int size, void *buffer );
```

APÊNDICE E – Implementação do módulo

```
////////////////////////////////////////
//
// Karlos H. Budag
// Maio / 2002
// Modulo (MainCrux.c)
//
////////////////////////////////////

/* Declare what kind of code we want from the header files */
#define __KERNEL__          /* We're part of the kernel */
#define MODULE              /* Not a permanent part, though. */

//__NR_...
#include <asm/unistd.h>

/* Standard headers for LKMs */
#include <linux/modversions.h>
#include <linux/module.h>

#define _LOOSE_KERNEL_NAMES
    /* With some combinations of Linux and gcc, tty.h will not compile if
       you don't define _LOOSE_KERNEL_NAMES.  It's a bug somewhere.
    */
#include <linux/tty.h>      /* console_print() interface */

////////////////////////////////////

#include "global.h"
#include "WrkNet.h"
#include "KrnlInt.h"
#include "CtrlNet.h"

////////////////////////////////////////

extern void *sys_call_table[];
asmlinkage int (*original_call)(const char *, int, int);
int _crux_chamadas = -1;

////////////////////////////////////////////////////

void ResetLinks()
{
  int i = 1;
```

```
  for ( ; i < _MAX_NODES; i++ )
    ResetWrkNet ( i );

  SrvResetCtrlNet();
}

void setarChamadas()
{
  _socketcall = sys_call_table[__NR_socketcall];
  _close = sys_call_table[__NR_close];
  _fcntl = sys_call_table[__NR_fcntl];
  _nanosleep = sys_call_table[__NR_nanosleep];
  _yield = sys_call_table[__NR_sched_yield];
  _getpid = sys_call_table[__NR_getpid];
  _gettimeofday = sys_call_table[__NR_gettimeofday];
}

int encontreChamadaLivre ( int num )
{
  int i = 255;
  void *ultimaChamada = 0;
  int indiceChamadaLivre = -1;

  ultimaChamada = sys_call_table[i];

  for ( i--; i >= 0; i-- )
  {
    if ( sys_call_table[i] != ultimaChamada )
    {
      indiceChamadaLivre = i+1;
      break;
    }
  }

  // Nao foram encontradas chamadas livres suficientes
  if ( (256 - num) < indiceChamadaLivre )
    return -1;

  return indiceChamadaLivre;
}


#ifdef _meiofisico_tcp_
char *addr;
MODULE_PARM(addr, "s");
#endif

int ValidarEndereco ( char *strAddr )
{
#ifdef _meiofisico_tcp_
  char *octeto1;
```

```
  char *octeto2;
  char *octeto3;
  char *octeto4;

  octeto1 = (char*)strAddr;
  if ( ! octeto1 )
  {
    printk ( "crux: ERRO - Parâmetro 'addr' não definido\n" );
    return 0;
  }

  octeto2 = (char*)strchr ( octeto1, '.' );
  if ( ! octeto2 )
  {
    printk ( "crux: ERRO - Parâmetro 'addr' inválido. Ex.: addr=127.0.0.1\n" );
    return 0;
  }

  octeto2 ++;
  octeto3 = (char*)strchr ( octeto2, '.' );
  if ( ! octeto3 )
  {
    printk ( "crux: ERRO - Parâmetro 'addr' inválido. Ex.: addr=127.0.0.1\n" );
    return 0;
  }

  octeto3 ++;
  octeto4 = (char*)strchr ( octeto3, '.' );
  if ( ! octeto4 )
  {
    printk ( "crux: ERRO - Parâmetro 'addr' inválido. Ex.: addr=127.0.0.1\n" );
    return 0;
  }

  octeto4 ++;
  if ( octeto4[0] == '\0' ) //O ultimo octeto veio vazio
  {
    printk ( "crux: ERRO - Parâmetro 'addr' inválido. Ex.: addr=127.0.0.1\n" );
    return 0;
  }

  _serverAddr = MakeAddr ( atoi(octeto1), atoi(octeto2), atoi(octeto3),
    atoi(octeto4) );
#endif

  return 1;
}

/* Initialize the module */
int init_module()
{
  int i = 0;
```

```
  printk ( "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< init_module(1)\n" );

#ifdef _meiofisico_tcp_
  if ( ! ValidarEndereco ( addr ) )
    return -1;
#endif

  setarChamadas();

  printk ( "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< init_module(2)\n" );

  i = encontreChamadaLivre ( 3 );
  if ( i < 0 )
  {
    printk ( "crux: ERRO - Uma chamada de sistema livre não foi encontrada\n" );
    return -1;
  }

  printk ( "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< init_module(3)\n" );

  _crux_chamadas = i;
  original_call = sys_call_table[_crux_chamadas];
  sys_call_table[_crux_chamadas]   = sys_s_Send;
  sys_call_table[_crux_chamadas+1] = sys_s_Receive;
  sys_call_table[_crux_chamadas+2] = sys_ss_ReceiveAny;
  sys_call_table[_crux_chamadas+3] = sys_s_GetData;
  sys_call_table[_crux_chamadas+4] = sys_s_GetId;
  sys_call_table[_crux_chamadas+5] = sys_ks_Send;
  sys_call_table[_crux_chamadas+6] = sys_ks_ReceiveAny;
  sys_call_table[_crux_chamadas+7] = sys_kc_SendReceive;

  printk ( "crux: As novas chamadas foram instaladas a partir da posição %d\n",
    _crux_chamadas );

  /* If we return a non zero value, it means that
   * init_module failed and the kernel module
   * can't be loaded */
  return 0;
}


/* Cleanup - undid whatever init_module did */
void cleanup_module()
{
  printk ( "<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<< cleanup_module(1)\n" );

  if ( _crux_chamadas < 0 )
  {
    printk (
      "crux: ERRO - As chamadas de sistema não foram encontradas na tabela\n" );
    return;
```

```
  }

  printk ( "<<<<<<<<<<<<<<<>>>>>>>>>>>>>>> cleanup_module(2)\n" );

  ResetLinks();

  printk ( "<<<<<<<<<<<<<<<>>>>>>>>>>>>>>> cleanup_module(3)\n" );

  sys_call_table[_crux_chamadas]   = original_call;
  sys_call_table[_crux_chamadas+1] = original_call;
  sys_call_table[_crux_chamadas+2] = original_call;
  sys_call_table[_crux_chamadas+3] = original_call;
  sys_call_table[_crux_chamadas+4] = original_call;
  sys_call_table[_crux_chamadas+5] = original_call;
  sys_call_table[_crux_chamadas+6] = original_call;
  sys_call_table[_crux_chamadas+7] = original_call;

  printk ( "crux: As chamadas de sistema foram desinstaladas\n" );
}
```

```
////////////////////////////////////////
//
// Karlos H. Budag
// Maio / 2002
// Modulo (MainCrux.h)
//
////////////////////////////////////

#ifndef __MAINCRUX_H
#define __MAINCRUX_H

#define _MAX_NODES 5

#define kc_RETURN 0
#define kc_CONNECT 1
#define kc_CONNECT_ANY 2
#define kc_DISCONNECT 3

#define kc_PROT_SYSTEM 0
#define kc_PROT_SUPERPASCAL 1
#define kc_PROT_PVM 2
#define kc_PROT_MPI 3

typedef char CTRLBUF[16];

//Descobre a que protocolo se refere uma mensagem
void UnmakeKcMsg ( CTRLBUF *msg, int *op, int *param1, int *param2 );
void UnmakeProt ( CTRLBUF *msg, int *prot )
{
  int i = 0;
  int op, param1, param2;
  int len = sizeof ( *prot );
  memcpy ( prot, &(*msg)[i], len );

  //Soh pra debugar o valor do buffer
  UnmakeKcMsg ( msg, &op, &param1, &param2 );
}

//Monta um buffer com uma mensagem do sistema
void MakeKcMsg ( CTRLBUF *msg, int op, int param1, int param2 )
{
  int i = 0;
  int prot = kc_PROT_SYSTEM;
  int len = sizeof ( prot );

  printf ( "<<<<<<<<<<<<<<<>>>>>>>>>>>>>>> MakeKcMsg: prot %d, op %d, param1 %d,
    param2 %d\n", prot, op, param1, param2 );
  memcpy ( &(*msg)[i], &prot, len );

  len = sizeof ( op );
  i = i + len;
  memcpy ( &(*msg)[i], &op, len );
```

```
  len = sizeof ( param1 );
  i = i + len;
  memcpy ( &(*msg)[i], &param1, len );

  len = sizeof ( param2 );
  i = i + len;
  memcpy ( &(*msg)[i], &param2, len );
}

void UnmakeKcMsg ( CTRLBUF *msg, int *op, int *param1, int *param2 )
{
  int i = 0;
  int prot = 0;
  int len = sizeof ( prot );
  memcpy ( &prot, &(*msg)[i], len );

  if ( prot != kc_PROT_SYSTEM )
    return;

  len = sizeof ( *op );
  i = i + len;
  memcpy ( op, &(*msg)[i], len );

  len = sizeof ( *param1 );
  i = i + len;
  memcpy ( param1, &(*msg)[i], len );

  len = sizeof ( *param2 );
  i = i + len;
  memcpy ( param2, &(*msg)[i], len );

  printf ( "<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>> UnmakeKcMsg: prot %d, op %d, param1 %d,
    param2 %d\n", prot, *op, *param1, *param2 );
}

//Monta um buffer com uma mensagem do SuperPascal
void MakeSpMsg ( CTRLBUF *msg )
{
  int i = 0;
  int prot = kc_PROT_SUPERPASCAL;
  int len = sizeof ( prot );
  memcpy ( &(*msg)[i], &prot, len );
}

void UnmakeSpMsg ( CTRLBUF *msg )
{
  int i = 0;
  int prot = 0;
  int len = sizeof ( prot );
  memcpy ( &prot, &(*msg)[i], len );
```

```
  if ( prot != kc_PROT_SUPERPASCAL )
    return;
}

//Monta um buffer com uma mensagem do PVM
void MakePvmMsg ( CTRLBUF *msg )
{
  int i = 0;
  int prot = kc_PROT_PVM;
  int len = sizeof ( prot );
  memcpy ( &(*msg)[i], &prot, len );
}

void UnmakePvmMsg ( CTRLBUF *msg )
{
  int i = 0;
  int prot = 0;
  int len = sizeof ( prot );
  memcpy ( &prot, &(*msg)[i], len );

  if ( prot != kc_PROT_PVM )
    return;
}

//Monta um buffer com uma mensagem do MPI
void MakeMpiMsg ( CTRLBUF *msg )
{
  int i = 0;
  int prot = kc_PROT_MPI;
  int len = sizeof ( prot );
  memcpy ( &(*msg)[i], &prot, len );
}

void UnmakeMpiMsg ( CTRLBUF *msg )
{
  int i = 0;
  int prot = 0;
  int len = sizeof ( prot );
  memcpy ( &prot, &(*msg)[i], len );

  if ( prot != kc_PROT_MPI )
    return;
}

////////////////////////////////////////////

#include <linux/unistd.h>

#define _pos_NRs            191
#define __NR_s_Send         _pos_NRs
#define __NR_s_Receive      _pos_NRs+1
#define __NR_ss_ReceiveAny  _pos_NRs+2
```

```
#define __NR_s_GetData      _pos_NRs+3
#define __NR_s_GetId        _pos_NRs+4
#define __NR_ks_Send        _pos_NRs+5
#define __NR_ks_ReceiveAny  _pos_NRs+6
#define __NR_kc_SendReceive _pos_NRs+7

_syscall4 (int, s_Send,         int,  thisProc, int,      proc, void*,    msg,
  int,  length );
_syscall4 (int, s_Receive,      int,  thisProc, int,      proc, void*,    msg,
  int*, length );
_syscall4 (int, ss_ReceiveAny,  int,  thisProc, int*,     proc, void*,    msg,
  int*, length );
_syscall2 (int, s_GetData,      int,  node,     int*,     data               );
_syscall0 (int, s_GetId                                                      );
_syscall2 (int, ks_Send,        int,  node,     CTRLBUF*, msg                );
_syscall2 (int, ks_ReceiveAny,  int*, node,     CTRLBUF*, msg                );
_syscall3 (int, kc_SendReceive, int,  thisProc, CTRLBUF*, msg,  CTRLBUF*, ret);

#endif //__MAINCRUX_H
```

APÊNDICE F – Rotinas globais

```
//////////////////////////////////////
//
// Karlos H. Budag
// Maio / 2002
// Funcoes genericas (global.c)
//
//////////////////////////////////////

/* Declare what kind of code we want from the header files */
#define __KERNEL__          /* We're part of the kernel */
#define MODULE              /* Not a permanent part, though. */

//get_fs(), copy_from_user()...
#include <asm/uaccess.h>

//__NR_...
#include <asm/unistd.h>

//INADDR_ANY
#include <linux/in.h>

//prefetch
#ifndef _kernel_antigo_
#undef _LINUX_PREFETCH_H
#undef ARCH_HAS_PREFETCH
#include <linux/prefetch.h>
#endif

#ifndef _kernel_antigo_
#include <linux/module.h>
#endif

#include <linux/modversions.h>
#include <linux/tty.h>      /* console_print() interface */

//////////////////////////////////////////

#include "global.h"

//////////////////////////////////////////

#ifdef _meiofisico_tcp_
int MakeAddr ( int byte1, int byte2, int byte3, int byte4 )
{
  int addr = 0;
```

```
  char *pAddr = (char*)&addr;

  pAddr[0] = byte1;
  pAddr[1] = byte2;
  pAddr[2] = byte3;
  pAddr[3] = byte4;

  return addr;
}

int criarSocket ( int port, int bNonBlocking, int bDgram )
{
  int optval = 0;
  int umSocket = 0;
  struct sockaddr_in sock_ser;
  int retval;

  //Criando o socket
  umSocket = _crux_socket ( AF_INET, ( bDgram ? SOCK_DGRAM : SOCK_STREAM ), 0 );
  if ( umSocket < 0 )
  {
    printk ( "criarSocket: erro socket() %d\n", umSocket );
    return ( -1 );
  }

  //Setando modo nao bloqueante
  if ( bNonBlocking )
  {
    retval = _crux_fcntl ( umSocket, F_SETFL, O_NONBLOCK );
    if ( retval != 0 )
      printk ( "criarSocket: erro fcntl(): %d, %d\n", umSocket, retval );
  }

  //Reutilizar a porta...
  optval = 1;
  retval = _crux_setsockopt ( umSocket, SOL_SOCKET, SO_REUSEADDR, &optval,
    sizeof ( optval ) );
  if ( retval < 0 )
  {
    printk ( "criarSocket: erro setsockopt() %d, %d\n", umSocket, retval );
    _crux_close ( umSocket );
    return ( -1 );
  }

  //Nomeando o socket
  _memset ( &sock_ser, 0, sizeof ( sock_ser ) );
  sock_ser.sin_family = AF_INET;
  sock_ser.sin_addr.s_addr = htonl ( INADDR_ANY );
  sock_ser.sin_port = htons ( port );
  retval = _crux_bind( umSocket, (struct sockaddr *) &sock_ser,
    sizeof(sock_ser));
  if ( retval < 0 )
```

```
  {
    printk ( "criarSocket: erro bind() %d, %d\n", umSocket, retval );
    _crux_close ( umSocket );
    return ( -1 );
  }

  //Indicando que aceita conexoes...
  retval = _crux_listen ( umSocket, 1 );
  if ( retval < 0 )
  {
    printk ( "criarSocket: erro listen() %d, %d\n", umSocket, retval );
    _crux_close ( umSocket );
    return ( -1 );
  }

  return ( umSocket );
}

int conectarSocket ( int addr, int port, int bDgram )
{
  struct sockaddr_in serv_sock;
  int umSocket = 0;
  int retval = 0;

  //Criando o socket
  umSocket = _crux_socket ( AF_INET, ( bDgram ? SOCK_DGRAM : SOCK_STREAM ), 0 );
  if ( umSocket < 0 )
  {
    printk ( "conectarSocket: erro socket() %d\n", umSocket );
    return ( -1 );
  }

  //Conectando
  _memset ( &serv_sock, 0, sizeof(serv_sock) );
  serv_sock.sin_family = AF_INET;
  serv_sock.sin_port = htons ( port );
  _memcpy ( &serv_sock.sin_addr, &addr, sizeof(serv_sock.sin_addr) );
  retval = _crux_connect ( umSocket, (struct sockaddr *)&serv_sock,
    sizeof(serv_sock) );
  if ( retval < 0)
  {
    printk ( "conectarSocket: erro connect() %d, %d\n", umSocket, retval );
    _crux_close ( umSocket );
    return ( -1 );
  }

  return ( umSocket );
}

int acceptSocket ( int umSocket, int *addr )
{
  int fromlen = 0;
```

```
  int novoSocket = 0;
  struct sockaddr_in sock_cli;

  //A conexao eh nao bloqueante
  fromlen = sizeof(sock_cli);
  novoSocket = _crux_accept ( umSocket, (struct sockaddr *)&sock_cli, &fromlen );

  //Eh um socket nao bloqueante
  if ( novoSocket >= 0 )
  {
    _memcpy ( addr, &sock_cli.sin_addr.s_addr, sizeof ( *addr ) );
    _print3 ( "acceptSocket %d, %d\n", novoSocket, *addr );
  }

  return ( novoSocket );
}
#endif /*_meiofisico_tcp*/

////////////////////////////////////////////////

int _crux_getpid(void)
{
  int ret = 0;
  mm_segment_t oldFs = get_fs();

  set_fs ( KERNEL_DS );
  ret = _getpid();
  set_fs ( oldFs );

  return ( ret );
}

int _crux_gettime ( struct timeval *tv )
{
  int ret = 0;
  mm_segment_t oldFs = get_fs();

  set_fs ( KERNEL_DS );
  ret = _gettimeofday(tv, 0);
  set_fs ( oldFs );

  return ( ret );
}

//Retirado de /usr/src/linux/include/linux/sched.h
int _signal_pending(struct task_struct *p)
{
  return (p->sigpending != 0);
}

int _crux_yield(void)
{
```

```
  int ret = 0;
  mm_segment_t oldFs = get_fs();

  //Testando a ocorrencia de sinais
  //para que um processo nao fique preso no polling
  if ( _signal_pending(current) )
    return -1;

  set_fs ( KERNEL_DS );
  ret = _yield();
  set_fs ( oldFs );

  //Chama o escalonador
  schedule();

  return ( ret );
}

int _crux_nanosleep ( int tempo )
{
  int ret = 0;
  struct timespec tsreq;
  struct timespec tsrem;
  mm_segment_t oldFs = get_fs();

  tsreq.tv_sec = 0;
  tsreq.tv_nsec = tempo;

  tsrem.tv_sec = 0;
  tsrem.tv_nsec = 0;

  set_fs ( KERNEL_DS );
  ret = _nanosleep ( &tsreq, &tsrem );
  set_fs ( oldFs );

  return ( ret );
}

int _crux_fcntl(int fd, int cmd, long arg)
{
  int ret = 0;
  mm_segment_t oldFs = get_fs();

  set_fs ( KERNEL_DS );
  ret = _fcntl ( fd, cmd, arg );
  set_fs ( oldFs );

  return ( ret );
}

int _crux_close(int fd)
{
```

```
  int ret = 0;
  mm_segment_t oldFs = get_fs();

  set_fs ( KERNEL_DS );
  ret = _close ( fd );
  set_fs ( oldFs );

  return ( ret );
}

int _crux_socket(int family, int type, int protocol)
{
  int ret = 0;
  unsigned long a[6];
  mm_segment_t oldFs = get_fs();

  a[0] = family;
  a[1] = type;
  a[2] = protocol;

  set_fs ( KERNEL_DS );
  ret = _socketcall ( SYS_SOCKET, &a[0] );
  set_fs ( oldFs );

  return ( ret );
}

int _crux_connect(int fd, struct sockaddr *uservaddr, int addrlen)
{
  int ret = 0;
  unsigned long a[6];
  mm_segment_t oldFs = get_fs();

  a[0] = fd;
  a[1] = (unsigned long)uservaddr;
  a[2] = addrlen;

  set_fs ( KERNEL_DS );
  ret = _socketcall ( SYS_CONNECT, &a[0] );
  set_fs ( oldFs );

  return ( ret );
}

int _crux_bind(int fd, struct sockaddr *umyaddr, int addrlen)
{
  int ret = 0;
  unsigned long a[6];
  mm_segment_t oldFs = get_fs();

  a[0] = fd;
  a[1] = (unsigned long)umyaddr;
```

```
  a[2] = addrlen;

  set_fs ( KERNEL_DS );
  ret = _socketcall ( SYS_BIND, &a[0] );
  set_fs ( oldFs );

  return ( ret );
}

int _crux_listen(int fd, int backlog)
{
  int ret = 0;
  unsigned long a[6];
  mm_segment_t oldFs = get_fs();

  a[0] = fd;
  a[1] = backlog;

  set_fs ( KERNEL_DS );
  ret = _socketcall ( SYS_LISTEN, &a[0] );
  set_fs ( oldFs );

  return ( ret );
}

int _crux_accept(int fd, struct sockaddr *upeer_sockaddr, int *upeer_addrlen)
{
  int ret = 0;
  unsigned long a[6];
  mm_segment_t oldFs = get_fs();

  a[0] = fd;
  a[1] = (unsigned long)upeer_sockaddr;
  a[2] = (unsigned long)upeer_addrlen;

  set_fs ( KERNEL_DS );
  ret = _socketcall ( SYS_ACCEPT, &a[0] );
  set_fs ( oldFs );

  return ( ret );
}

int _crux_send (int s, const void *msg, int len, unsigned int flags)
{
  int ret = 0;
  unsigned long a[6];
  mm_segment_t oldFs = get_fs();

  a[0] = s;
  a[1] = (unsigned long)msg;
  a[2] = len;
  a[3] = flags;
```

```
  set_fs ( KERNEL_DS );
  ret = _socketcall ( SYS_SEND, &a[0] );
  set_fs ( oldFs );

  return ( ret );
}

int _crux_recv (int s, void *buf, int len, unsigned int flags)
{
  int ret = 0;
  unsigned long a[6];
  mm_segment_t oldFs = get_fs();

  a[0] = s;
  a[1] = (unsigned long)buf;
  a[2] = len;
  a[3] = flags;

  set_fs ( KERNEL_DS );
  ret = _socketcall ( SYS_RECV, &a[0] );
  set_fs ( oldFs );

  return ( ret );
}

int _crux_setsockopt(int s, int level, int optname, const void *optval,
  int optlen)
{
  int ret = 0;
  unsigned long a[6];
  mm_segment_t oldFs = get_fs();

  a[0] = s;
  a[1] = level;
  a[2] = optname;
  a[3] = (unsigned long)optval;
  a[4] = optlen;

  set_fs ( KERNEL_DS );
  ret = _socketcall ( SYS_SETSOCKOPT, &a[0] );
  set_fs ( oldFs );

  return ( ret );
}

////////////////////////////////////////////

void _memset ( void *buffer, char val, unsigned int size )
{
  int i = 0;
  char *cBuf = buffer;
```

```
  for ( ; i < size; i++ )
    cBuf[i] = val;
}

void _memcpy ( void *dest, void *source, unsigned int size )
{
  int i = 0;
  char *cDest = (char*)dest;
  char *cSource = (char*)source;

  for ( ; i < size; i++ )
    cDest[i] = cSource[i];
}

unsigned int _strlen ( char *buffer )
{
  int i = 0;

  while ( buffer[i] != '\0' )
    i++;

  return i;
}

//////////////////////////////////////////////////////

#ifndef _kernel_antigo_
MODULE_LICENSE("GPL");
MODULE_AUTHOR("Karlos H. Budag");
MODULE_DESCRIPTION("Primitivas de comunicacao do ACrux");
#endif
```

```
//////////////////////////////////////
//
// Karlos H. Budag
// Maio / 2002
// Funcoes genericas (global.h)
//
////////////////////////////////////

#define _MAX_NODES 5 #define SRVNODE_PORT 9400

//Utilize _print diretamente, para mensagens de erro
//independentes de _DEBUG
#define _print printk

#define _DEBUG
#ifdef _DEBUG
  #define _print1 _print
  #define _print2 _print
  #define _print3 _print
  #define _print4 _print
  #define _print5 _print
  #define _print6 _print
#else
  #define _print1(par1)
  #define _print2(par1,par2)
  #define _print3(par1,par2,par3)
  #define _print4(par1,par2,par3,par4)
  #define _print5(par1,par2,par3,par4,par5)
  #define _print6(par1,par2,par3,par4,par5,par6)
#endif

////////////////////////////////////////////////

//Escalonador
extern struct task_struct *current;
extern inline int signal_pending(struct task_struct *p);
extern asmlinkage void schedule(void);

//Sockets
#ifdef _meiofisico_tcp_
int MakeAddr ( int byte1, int byte2, int byte3, int byte4 );
int criarSocket ( int port, int bNonBlocking, int bDgram );
int conectarSocket ( int addr, int port, int bDgram );
int acceptSocket ( int umSocket, int *addr );
int _serverAddr;
#endif

//Chamadas de sistema
asmlinkage int (*_socketcall)(int call, unsigned long *args);
asmlinkage int (*_close)(int fd);
asmlinkage int (*_fcntl)(int fd, int cmd, long arg);
asmlinkage int (*_nanosleep)(const struct timespec *req, struct timespec *rem);
```

```
asmlinkage int (*_yield)(void);
asmlinkage int (*_getpid)(void);
asmlinkage int (*_gettimeofday)(struct timeval *tv, struct timezone *tz);

int _crux_gettime ( struct timeval *tv );
int _crux_getpid(void);
int _crux_yield(void);
int _crux_nanosleep ( int tempo );
int _crux_fcntl(int fd, int cmd, long arg);
int _crux_close(int fd);
int _crux_socket(int family, int type, int protocol);
int _crux_connect(int fd, struct sockaddr *uservaddr, int addrlen);
int _crux_bind(int fd, struct sockaddr *umyaddr, int addrlen);
int _crux_listen(int fd, int backlog);
int _crux_accept(int fd, struct sockaddr *upeer_sockaddr, int *upeer_addrlen);
int _crux_send (int s, const void *msg, int len, unsigned int flags);
int _crux_recv (int s, void *buf, int len, unsigned int flags);
int _crux_setsockopt(int s, int level, int optname, const void *optval,
  int optlen);

//Funcoes utilitarias
void _memset ( void *buffer, char val, unsigned int size );
void _memcpy ( void *dest, void *source, unsigned int size );
unsigned int _strlen ( char *buffer );
```

APÊNDICE G – Servidor de controle

```
//////////////////////////////////////
//
// Karlos H. Budag
// Maio / 2002
// Servidor de controle (SrvNode.c)
//
//////////////////////////////////////

#include <signal.h>
#include "MainCrux.h"

////////////////////////////////

#define _MAX_LINKS 8
#define _ANYONE -1
#define _EMPTY -2

//Mensagens de debug
#define _print printf

#define _DEBUG
#ifdef _DEBUG
  #define _print1 _print
  #define _print2 _print
  #define _print3 _print
  #define _print4 _print
  #define _print5 _print
  #define _print6 _print
#else
  #define _print1(par1)
  #define _print2(par1,par2)
  #define _print3(par1,par2,par3)
  #define _print4(par1,par2,par3,par4)
  #define _print5(par1,par2,par3,par4,par5)
  #define _print6(par1,par2,par3,par4,par5,par6)
#endif

typedef struct
{
  int Connected;
  int ReceiveAny;
  int queue[_MAX_NODES];
  int addr;
}  _LINKS;
```

```
_LINKS _links[_MAX_LINKS];

typedef struct
{
  int usage[_MAX_NODES];
  int link[_MAX_NODES];
} _WRK_LINKS;

_WRK_LINKS _wrk_links[_MAX_NODES];

int _thisProc = 0;
int _serverAddr = 0;

////////////////////////////////////

void SetUpLink ( int node1, int node2, int *link )
{
  if ( ! _wrk_links[node1].link[node2] )
  {
    _wrk_links[node1].link[node2] = _links[node2].addr;

    _print4 ( "SetupLink: Criando conexao entre %d e %d pelo link %d\n", node1,
      node2, _wrk_links[node1].link[node2] );

    _wrk_links[node2].link[node1] = _links[node1].addr;
  }

  _wrk_links[node1].usage[node2] ++;
  _wrk_links[node2].usage[node1] ++;

  *link = _wrk_links[node1].link[node2];
}

void UnsetLink ( int node1, int node2 )
{
  _wrk_links[node1].usage[node2] --;
  _wrk_links[node2].usage[node1] --;

  if ( ! _wrk_links[node1].usage[node2] )
  {
    _wrk_links[node1].link[node2] = 0;
    _wrk_links[node2].link[node1] = 0;

    _print1 ( "UnsetLink: Desfazendo conexao entre %d e %d\n", node1, node2 );
  }
}

////////////////////////////////////////

void AddQueue ( int node1, int node2 )
{
  int i = 0;
```

```
  for ( ; i < _MAX_NODES; i++ )
  {
    if ( _links[node1].queue[i] == _EMPTY )
    {
      _links[node1].queue[i] = node2;
      return;
    }
  }
}

int InQueue ( int node1, int node2 )
{
  int i = 0;

  if ( node2 == _ANYONE )
    return ( _links[node1].queue[0] != _EMPTY ? 1 : 0 );

  for ( ; i < _MAX_NODES; i++ )
  {
    if ( _links[node1].queue[i] == node2 )
      return 1;
  }

  return 0;
}

int RemoveQueue ( int node1, int node2 )
{
  int i = 0;
  int iRem = -1;
  int node = 0;

  if ( node2 == _ANYONE )
    node2 = _links[node1].queue[0];

  for ( ; i < _MAX_NODES; i++ )
  {
    if ( _links[node1].queue[i] == node2 )
    {
      iRem = i;
      node = _links[node1].queue[i];
      _links[node1].queue[i] = _EMPTY;

      break;
    }
  }

  if ( iRem >= 0 )
  {
    for ( i = iRem+1; i < _MAX_NODES; i++ )
    {
```

```
      _links[node1].queue[i-1] = _links[node1].queue[i];

      if ( _links[node1].queue[i] == _EMPTY )
        break;
    }
  }

  return node;
}

////////////////////////////////////////////////////

void InitLinks()
{
  int i = 0;
  int ii = 0;
  int link = 0;

  for ( ; i < _MAX_LINKS; i++ )
  {
    _links[i].Connected = 0;
    _links[i].ReceiveAny = 0;

    for ( ii = 0; ii < _MAX_NODES; ii++ )
      _links[i].queue[ii] = _EMPTY;
  }

  for ( i = 0; i < _MAX_NODES; i++ )
  {
    for ( ii = 0; ii < _MAX_NODES; ii++ )
    {
      _wrk_links[i].usage[ii] = 0;
      _wrk_links[i].link[ii] = 0;
    }
  }
}

void ResetLinks()
{
}

////////////////////////////////////////////////////

void sighandler_kill ( int val )
{
  //apenas para que o processo nao morra
}

typedef void (*sighandler_t)(int);
sighandler_t oldsig;

void prot_system ( int link, CTRLBUF *msg )
```

```
{
  int op = 0;
  int param1 = 0;
  int param2 = 0;

    UnmakeKcMsg ( msg, &op, &param1, &param2 );

    //endereco IP
    _links[param1].addr = link;
    _print5 ( "prot_system: operacao %d, param1 %d, param2 %d, link %d\n", op,
      param1, param2, link );

    switch ( op )
    {
      case kc_CONNECT:

        _print1 ( "prot_system: executando kc_CONNECT\n" );

        //Zera a conexao anterior
        if ( _links[param1].Connected )
          _links[param1].Connected = 0;

        //O outro canal quer se comunicar com este?
        if ( InQueue ( param1, param2 ) )
        {
          param2 = RemoveQueue ( param1, param2 );

          _links[param1].Connected = param2;
          _links[param2].Connected = param1;

          SetUpLink ( param1, param2, &link );
          _print2 ( "prot_system: conectando %d com %d atraves do link %d\n",
            param1, param2, link );
          MakeKcMsg ( msg, kc_RETURN, _thisProc, link );
          ks_Send ( param1, msg );

          SetUpLink ( param2, param1, &link );
          _print2 ( "prot_system: agora, conectando %d com %d atraves de %d\n",
            param2, param1, link );
          MakeKcMsg ( msg, kc_RETURN, _thisProc, link );
          ks_Send ( param2, msg );
        }

        //O outro canal aceita qualquer conexao
        else if ( _links[param2].ReceiveAny )
        {
          AddQueue ( param2, param1 );
          _links[param2].ReceiveAny = 0;

          MakeKcMsg ( msg, kc_RETURN, _thisProc, param1 );
          ks_Send ( param2, msg );
        }
```

```
  //Esta conexao deve ser colocada na fila de espera
  else
    AddQueue ( param2, param1 );

  break;

case kc_CONNECT_ANY:

  _print1 ( "prot_system: executando kc_CONNECT_ANY\n" );

  //Zera a conexao anterior
  if ( _links[param1].Connected )
    _links[param1].Connected = 0;

  //Algum canal quer se comunicar com este?
  if ( InQueue ( param1, _ANYONE ) )
  {
    param2 = RemoveQueue ( param1, _ANYONE );
    AddQueue ( param1, param2 );
    _print2 ( "prot_system: kc_CONNECT_ANY retornou nodo %d\n",
      param2 );

MakeKcMsg ( msg, kc_RETURN, _thisProc, param2 );
    ks_Send ( param1, msg );
  }

  //Marca esse canal
  else
    _links[param1].ReceiveAny = 1;

  break;

case kc_DISCONNECT:

  _print1 ( "prot_system: executando kc_DISCONNECT\n" );

  //Este nodo estah conectado
  if ( _links[param1].Connected )
  {
    //Apenas um dos nodos requisita a desconexao
    param2 = _links[param1].Connected;
    _links[param1].Connected = 0;
    _links[param2].Connected = 0;

    UnsetLink ( param1, param2 );
    UnsetLink ( param2, param1 );
    MakeKcMsg ( msg, kc_RETURN, _thisProc, 0 );
    ks_Send ( param1, msg );
  }

  //Este nodo nao estah conectado
```

```
        else
        {
          _print1 ( "prot_system: Erro kc_DISCONNECT - nodo %d nao conectado\n",
            param1 );

          MakeKcMsg ( msg, kc_RETURN, _thisProc, -1 );
          ks_Send ( param1, msg );
        }

        break;

      default:
        _print1 ( "prot_system: erro op %d inexistente\n", op );
    }
}

void prot_superpascal ( int link, CTRLBUF *msg )
{
}

void prot_pvm ( int link, CTRLBUF *msg )
{
}

void prot_mpi ( int link, CTRLBUF *msg )
{
}

void serverNode()
{
  CTRLBUF msg;
  int link = 0;
  int prot = 0;
  int ret = 0;

  _print ( "serverNode: servidor iniciando\n" );

  errno = 0;
  oldsig = signal(SIGINT, sighandler_kill);
  if ( oldsig == SIG_ERR )
    _print3 ( "serverNode: signal %d, %d\n", oldsig, errno );

  while ( 1 )
  {
    ret = ks_ReceiveAny ( &link, &msg );
    if ( ret < 0 )
    {
      _print2 ( "serverNode: sinal interrompeu ks_ReceiveAny - %d\n", ret );
      break;  //Interrompido por um sinal
    }

    UnmakeProt ( &msg, &prot );
```

```
    printf ( "<<<<<<<<<<<<<<<<<>>>>>>>>>>>>>>>>> UnmakeProt %d\n", prot );

    switch ( prot )
    {
      case kc_PROT_SYSTEM:
        prot_system ( link, &msg );
        break;

      case kc_PROT_SUPERPASCAL:
        prot_superpascal ( link, &msg );
        break;

      case kc_PROT_PVM:
        prot_pvm ( link, &msg );
        break;

      case kc_PROT_MPI:
        prot_mpi ( link, &msg );
        break;

      default:
        _print1 ( "serverNode: protocolo prot %d nao definido\n", prot );
    }
  }

  _print ( "serverNode: servidor terminando\n" );
}

int main(int argc, char* argv[])
{
  InitLinks();
  serverNode();
  ResetLinks();

  return 0;
}
```

APÊNDICE H – Makefile

```
SrvNode:    SrvNode.c MainCrux.h
    gcc SrvNode.c -o SrvNode

# incluir -D_kernel_antigo_ para compilar em kernels anteriores ao 2.4.x
# retirar -D_kernel_antigo_ para compilar em kernels mais recentes
# incluir _meiofisico_tcp_ para utilizar ACrux sobre tcp/ip (sockets)
# Os arquivos crux_sockets.c e .h contem a interface de acesso ao meio fisico
#   tcp/ip (via sockets)
# Os arquivos crux_1394_mod.c e .h contem a interface de acesso ao meio fisico
#   firewire

MainCrux.o: crux_sockets.c crux_sockets.h CtrlNet.c CtrlNet.h WrkNet.c WrkNet.h /
KrnlInt.c KrnlInt.h global.c global.h MainCrux.c
    gcc -D_kernel_antigo_ -D_meiofisico_tcp_ -I/usr/src/linux/include -c /
      crux_sockets.c CtrlNet.c WrkNet.c KrnlInt.c global.c MainCrux.c
    ld -r crux_sockets.o CtrlNet.o WrkNet.o KrnlInt.o global.o MainCrux.o /
      /usr/lib/libc.a -o MainCrux.aux.o
    rm -f crux_sockets.o
    rm -f CtrlNet.o
    rm -f WrkNet.o
    rm -f KrnlInt.o
    rm -f global.o
    rm -f MainCrux.o
    mv MainCrux.aux.o MainCrux.o
    insmod -p MainCrux.o
```

Referências Bibliográficas

1 BEOWULF project. Disponível em: <http://www.beowulf.org>. Acesso em: agosto de 2002.

2 BOGO, Madianita. *Interface da rede de controle do cluster Clux*. Dissertação (Mestrado em Ciência da Computação) — CPGCC / UFSC, Florianópolis, 2002.

3 BOING, Amilcar. *Um simulador para multicomputador implementado como núcleo de sistema operacional multiprogramado*. Dissertação (Mestrado em Ciência da Computação) — CPGCC / UFSC, Florianópolis, 1996.

4 CAMPOS, Rodrigo. *Um sistema operacional fundamentado no modelo cliente-servidor e um simulador multiprogramado de multicomputador*. Dissertação (Mestrado em Ciência da Computação) — CPGCC / UFSC, Florianópolis, 1995.

5 CANCIAN, Rafael. *Avaliação de desempenho de algoritmos de escalonamento de tempo real para o ambiente do multicomputador Crux*. Dissertação (Mestrado em Ciência da Computação) — CPGCC / UFSC, Florianópolis, 2001.

6 CLUSTER Computing Info Centre. Disponível em: <http://www.cs.mu.oz.au/ raj/cluster/>. Acesso em: agosto de 2002.

7 COMER DOUGLAS; FOSSUM, Timothy V. *Operating system design vol. I: the Xinu approach (PC edition)*. Englewood Cliffs: Prentice-Hall, 1988.

8 CORSO, Thadeu B. *Crux: ambiente multicomputador reconfigurável por demanda*. Tese (Doutorado em Engenharia Elétrica) — CPGEE / UFSC, Florianópolis, 1999.

9 GAMMA Project. Disponível em: <http://www.disi.unige.it/project/gamma/>. Acesso em: agosto de 2002.

10 GAVILAN, Júlio. *Síntese em alto nível de uma rede de interconexão dinâmica para multicomputador*. Dissertação (Mestrado em Ciência da Computação) — CPGCC / UFSC, Florianópolis, 1996.

11 INTERNET Parallel Computing Archive. Disponível em: <http://wotug.ukc.ac.uk/parallel/>. Acesso em: agosto de 2002.

12 MERKLE, Carla. *Ambiente para execução de programas paralelos escritos na linguagem Superpascal em um multicomputador com rede de interconexão dinâmica*. Dissertação (Mestrado em Ciência da Computação) — CPGCC / UFSC, Florianópolis, 1996.

13 MONTEZ, Carlos. *Um sistema operacional com micronúcleo distribuído e um simulador multiprogramado de multicomputador*. Dissertação (Mestrado em Ciência da Computação) — CPGCC / UFSC, Florianópolis, 1995.

14 PLENTZ, Patricia. *Um servidor de arquivos para um cluster de computadores.* Dissertação (Mestrado em Ciência da Computação) — CPGCC / UFSC, Florianópolis, 2002.

15 RECH, Luciana. *Interface da rede de trabalho do cluster Clux.* Dissertação (Mestrado em Ciência da Computação) — CPGCC / UFSC, Florianópolis, 2002.

16 SILBERSCHATZ ABRAHAM; GALVIN, Peter B. *Operating System Concepts.* 5 ed. New York: John Wiley & Sons, 1999.

17 SILVA, Valéria Alves da. *Multicomputador Nó//: implementação de primitivas básicas de comunicação e avaliação de desempenho.* Dissertação (Mestrado em Ciência da Computação) — CPGCC / UFSC, Florianópolis, 1996.

18 TANENBAUM, Andrew S. *Modern Operating Systems.* Englewood Cliffs: Prentice-Hall, 1992.

19 TANENBAUM, Andrew S. *Redes de Computadores.* 3 ed. Rio de Janeiro: Campus, 1997.

20 IEEE Task Force on Cluster Computing (TFCC). Disponível em: <http://www.ieeetfcc.org/>. Acesso em: agosto de 2002.

21 ZEFERINO, Cesar. *Projeto do sistema de comunicação de um multicomputador.* Dissertação (Mestrado em Ciência da Computação) — CPGCC / UFSC, Florianópolis, 1996.

22 FNN: Flat Neighborhood Networks. Disponível em: <http://aggregate.org/FNN>. Acesso em: agosto de 2002.

23 CONCRETE Architecture of the Linux Kernel. Disponível em: <http://plg.uwaterloo.ca/ itbowman/CS746G/a2/>. Acesso em: julho de 2002.

24 CONCEPTUAL Architecture of the Linux Kernel. Disponível em: <http://plg.uwaterloo.ca/ itbowman/CS746G/a1/>. Acesso em: julho de 2002.

25 THE GNU C Library. Disponível em: <http://www.gnu.org/manual/glibc-2.2.3/>. Acesso em: julho de 2002.

26 THE GNU Project and the Free Software Foundation (FSF). Disponível em: <http://www.gnu.org>. Acesso em: julho de 2002.

27 HISTORY of Linux. Disponível em: <http://ragib.hypermart.net/linux/>. Acesso em: setembro de 2002.

28 LINUX Kernel Hackers' Guide. Disponível em: <http://www.linuxdoc.org/LDP/khg/HyperNews/get/>. Acesso em: agosto de 2002.

29 THE Linux Kernel API. Disponível em: <http://www.kernelnewbies.org/documents/kdoc/kernel-api/linuxkernelapi.html>. Acesso em: outubro de 2002.

30 LINUX Online. Disponível em: <http://www.linux.org>. Acesso em: agosto de 2002.

31 THE Linux Programmer's Guide. Disponível em: <http://www.linuxhq.com/guides/LPG/lpg.html>. Acesso em: agosto de 2002.

32 (NEARLY) Complete Linux Loadable Kernel Modules. Disponível em: <http://packetstorm.linuxsecurity.com/docs/hack/LKM_HACKING.html>. Acesso em: julho de 2002.

33 IEEE 1394 "Firewire" Overview. Disponível em: <http://www.quatech.com/Application_Objects/FAQs/comm-over-1394.htm>>. Acesso em: setembro de 2002.

34 1394 Trade Association. Disponível em: <http://www.1394ta.org/Technology/About/faq.htm>. Acesso em: setembro de 2002.

35 HANSEN, Per Brinch. Superpascal – a publication language for parallel scientific computing. *Concurrency: Practice and Experience*, v. 6, n. 5, p. 461–483, aug. 1994.