

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

**UM SISTEMA OPERACIONAL COM MICRÔNÚCLEO  
DISTRIBUÍDO E UM SIMULADOR MULTIPROGRAMADO DE  
MULTICOMPUTADOR**

por  
Carlos Barros Montez

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

Prof. Thadeu Botteri Corso  
Orientador

Prof. Simão Sirineo Toscani  
Co-Orientador

Florianópolis, maio de 1995

**UM SISTEMA OPERACIONAL COM MICRONÚCLEO DISTRIBUÍDO E UM  
SIMULADOR MULTIPROGRAMADO DE MULTICOMPUTADOR**

CARLOS BARROS MONTEZ

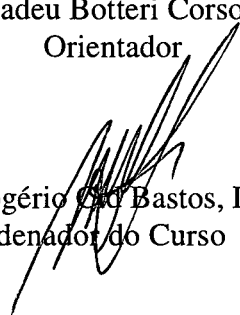
ESTA DISSERTAÇÃO FOI JULGADA ADEQUADA PARA OBTENÇÃO DO TÍTULO DE

**MESTRE EM CIÊNCIA DA COMPUTAÇÃO**

ESPECIALIDADE SISTEMAS DE COMPUTAÇÃO E APROVADA EM SUA FORMA  
FINAL PELO PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO



Prof. Thadeu Botteri Corso, M.Sc.  
Orientador

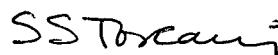


Prof. Rogério Bastos, Dr.  
Coordenador do Curso

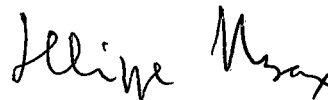
**BANCA EXAMINADORA**



Prof. Thadeu Botteri Corso, M.Sc. (Presidente)



Prof. Simão Sirineu Toscani, Dr. (Co-Orientador)



Prof. Philippe Olivier Alexandre Navaux, Dr.



Prof. Elizabeth Sueli Specialski, M.Sc.

À minha mãe Izolda e aos meus irmãos Cláudio, Luiz e Ângela.

## AGRADECIMENTOS

Este trabalho só se tornou possível através das pessoas que me ajudaram e colaboraram direta ou indiretamente para que alcançasse esse objetivo. Agradeço a todos e em especial ao meu orientador Thadeu pela sua simplicidade e seus valiosos conselhos, e ao colega Rodrigo pelas longas e esclarecedoras discussões.

À Maria pelo carinho e atenção.

Aos amigos Vicente e Brasília que me mostraram o caminho (e pelo constante bom-humor).

Ao David pelas horas de lanche.

À Carla pela amizade e simpatia.

À Verinha sempre atenciosa.

A todos os colegas que propiciaram uma companhia agradável e alegre durante as noites e dias passados no laboratório.

Não poderia deixar de agradecer ao brasileiro batalhador que, graças ao pagamento de seus impostos, permitiu a CAPES financiar a realização deste trabalho.

"Desconfiai do mais trivial,  
na aparência singelo.  
E examinai, sobretudo, o que parece habitual.  
Suplicamos expressamente:  
não aceiteis o que é de hábito  
como coisa natural."

Bertold Brecht

## Sumário

<b>Lista de Figuras.....</b>	<b>iv</b>
<b>Lista de Tabelas.....</b>	<b>vi</b>
<b>Lista de Abreviaturas .....</b>	<b>vii</b>
<b>Resumo .....</b>	<b>viii</b>
<b>Abstract.....</b>	<b>ix</b>
<b>1. Introdução.....</b>	<b>1</b>
<b>2. Estado da Arte.....</b>	<b>4</b>
2.1. Programação Concorrente.....	4
2.2. Sistemas com Múltiplos Processadores .....	7
2.2.1. Multiprocessadores .....	7
2.2.2. Redes de Computadores .....	8
2.2.3. Multicomputadores .....	9
2.2.4. Redes Estáticas e Redes Dinâmicas .....	11
2.3. Sistemas Operacionais Distribuídos .....	16
2.3.1. O Enfoque Micronúcleo .....	17
2.3.2. Comunicação entre Processos .....	22
2.3.3. Chamada de Procedimento Remoto .....	24
<b>3. Arquitetura do Multicomputador Nó // .....</b>	<b>27</b>
3.1. Os Nós.....	28
3.2. O Comutador de Conexões.....	29
3.3. O Controlador de Nós e Conexões (CNC).....	29

3.4. O Barramento de Serviços .....	30
3.5. O Nó de Comunicação Externa (NCE) .....	32
3.6. O Nó // e outros Multicomputadores .....	33
<b>4. Simulador do Multicomputador Nó // .....</b>	<b>34</b>
4.1. Necessidade de um Simulador .....	34
4.2. Sistema de Desenvolvimento .....	35
4.3. Implementação .....	36
4.3.1. Alterações no XINU .....	39
<b>5. Sistema Operacional do Multicomputador Nó // .....</b>	<b>42</b>
5.1. Descrição .....	42
5.2. Processos .....	43
5.3. Micronúcleo CRUX .....	45
5.3.1. Gerência do Barramento de Serviços .....	47
5.3.2. Gerência de Nós e Conexões .....	47
5.3.3. Gerência de Processos .....	49
5.3.4. Comunicação entre Processos .....	50
5.4. Biblioteca CRUX .....	53
5.4.1. Chamadas Resolvidas Localmente .....	54
5.4.2. Chamadas Resolvidas pelo Servidor CRUX .....	55
5.4.3. Chamadas Especiais .....	56
5.4.3.1. Fork .....	56
5.4.3.2. Exec .....	60

5.4.3.3. Exit.....	62
5.4.3.4. Wait.....	63
5.5. Servidor CRUX.....	64
5.5.1 Implementação do Servidor CRUX.....	65
5.6. Interpretador de Comandos.....	67
5.7. Carga Inicial do Sistema Operacional .....	70
<b>6. Conclusões.....</b>	<b>73</b>
6.1. Contribuições .....	74
6.2. Perspectivas.....	75
<b>Apêndice A - O Sistema XINU .....</b>	<b>78</b>
<b>Referências Bibliográficas .....</b>	<b>81</b>

## Lista de Figuras

Figura 2-1. Níveis de processamento.....	5
Figura 2-2. Multiprocessador: um sistema fortemente acoplado. ....	8
Figura 2-3. Rede de computadores: um sistema muito fracamente acoplado. ....	8
Figura 2-4. Multicomputador: um sistema fracamente acoplado. ....	10
Figura 2-5. Rede completamente conectada, anel, grelha e hipercubo. ....	12
Figura 2-6. Comutador de conexões do tipo crossbar. ....	15
Figura 2-7. A estrutura em camadas do sistema operacional THE. ....	17
Figura 2-8. O micronúcleo do Mach e seus sub-sistemas. ....	18
Figura 2-9. O modelo cliente-servidor em um sistema distribuído. ....	19
Figura 2-10. Uma chamada de procedimento remoto. ....	25
Figura 3-1. A arquitetura do Nó //. ....	27
Figura 3-2. A estrutura interna de um nó. ....	28
Figura 4-1. Nó // conectado a uma estação de trabalho.....	35
Figura 4-2. Trecho de código que cria processos XINU.....	37
Figura 4-3. Trecho de código que simula um nó.....	38
Figura 4-4. O XINU utilizado como base para o simulador do Nó //. ....	39
Figura 5-1. Um processo colocado na memória de um nó. ....	44



Figura 5-2. O micronúcleo embutido no código do processo. ....	46
Figura 5-3. As camadas do micronúcleo.....	46
Figura 5-4. Comunicação cliente-servidor utilizando serviços do micronúcleo.....	51
Figura 5-5. Comunicação entre um processo cliente e um servidor. ....	52
Figura 5-6. A biblioteca CRUX. ....	54
Figura 5-7. Algoritmo da chamada de sistema close. ....	56
Figura 5-8. Um programa típico utilizando fork.....	57
Figura 5-9. Etapas da chamada de sistema fork. ....	58
Figura 5-10. Etapas da chamada de sistema exec.....	61
Figura 5-11. Algoritmo de exit.....	63
Figura 5-12. Algoritmo de wait.....	64
Figura 5-13. Utilização de um servidor de arquivos externo ao Nó //. ....	66
Figura 5-14. Um trecho do programa shell.....	68
Figura 5-15. Execução do shell.....	69
Figura 5-16. Etapas da carga inicial do sistema operacional CRUX. ....	71
Figura A-1. Camadas do sistema XINU .....	78

## Lista de Tabelas

Tabela 5-1. Função da camada Gerência do Barramento de Serviços.....	47
Tabela 5-2. Funções da camada Gerência de Nós e Conexões .....	48
Tabela 5-3. Funções da camada de Gerência de Processos.....	49
Tabela 5-4. Funções da camada de Comunicação entre Processos.....	51

## Lista de Abreviaturas

4.3 BSD	Sistema Operacional da Berkeley Software Distributions compatível com UNIX.
CPGCC	Curso de Pós-graduação em Ciência da Computação.
CNC	Controlador de Nós e Conexões.
COFF	Common Object File Format – Formato Comum de Arquivo Objeto.
DMA	Direct Memory Access – Acesso Direto à Memória.
E/S	Entrada e Saída.
Kbyte	Kilobyte – 1024 bytes.
Mbyte	Megabyte – 1048576 bytes.
NCE	Nó de Comunicação Externa.
OSF/1	Sistema Operacional da Open Software Foundation compatível com UNIX.
OS/2	Sistema Operacional da IBM.
POSIX.1	IEEE Portable Operating System Interface for Computing Environments – Sistema Operacional Portável – Documento que padroniza a interface de programação UNIX.
RAM	Random Access Memory – Memória de Acesso Direto.
ROM	Read Only Memory – Memória Apenas de Leitura.
RPC	Remote Procedure Call – Chamada de Procedimento Remoto.
UFSC	Universidade Federal de Santa Catarina.

## Resumo

O desenvolvimento de máquinas formadas por vários processadores é objeto de inúmeras pesquisas visando obter maior poder de processamento. Os grupos de pesquisa em Arquitetura de Computadores e Sistemas Operacionais do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina, inserem-se nessa linha de pesquisa propondo um projeto que visa a construção de um multicomputador com rede de interconexão dinâmica, através do qual pretende-se resolver alguns problemas inerentes das arquiteturas paralelas de forma simples e inovadora.

O presente trabalho faz parte desse projeto, propondo: (a) um simulador para o multicomputador, concebido como um ambiente multiprogramado, para servir como base de desenvolvimento de *software* para a máquina real, e (b) um sistema operacional com micronúcleo distribuído, cujos serviços, compatíveis a nível de programação com os do sistema UNIX, são suportados por um conjunto de processos servidores.

## **Abstract**

Nowadays there are countless researches on parallel computer architectures which gather several processing elements in a single machine to increase processing power. The research teams in Computers Architecture and in Operating Systems from the Pos-Graduation Course in Computer Science of the Universidade Federal de Santa Catarina has spent a lot of effort in order to develop a dynamic interconnection network multicomputer which the main goal is to support the implementation of parallel algorithms in a simple and innovative way.

This work describes proposal for two specific subject in this project: (a) a simulator of the multicomputer, conceived like a multiprogramming environment, providing to the real machine a software development base, and (b) a operational system with distributed microkernel, which services, compatible at programming level with the UNIX systems are supported by a set of servers processes.

## 1. Introdução

Devido ao barateamento de preço dos processadores, existem várias pesquisas em andamento sobre novas arquiteturas de computadores formados por mais de um processador. Objetiva-se com tais arquiteturas o aumento da produtividade através do paralelismo real de processamentos.

Essa linha de pesquisa se distingue da tradicional na qual o desenvolvimento de processadores mais potentes é o objetivo principal. Observa-se, no atual estado da arte, que para se conseguir um pequeno ganho na capacidade de processamento destes, é necessário um grande investimento em pesquisas.

Por outro lado, a construção de computadores formados por vários processadores existentes no mercado pode viabilizar um produto com capacidade de processamento muito maior do que a de um computador monoprocessador tradicional.

Entretanto, a utilização dessas máquinas paralelas implica na confecção de uma nova classe de sistemas operacionais, denominados sistemas operacionais distribuídos, que se preocupam em tornar transparente aos programas de aplicação a arquitetura empregada.

Devido aos requisitos criados por essa nova classe de sistemas operacionais, novos paradigmas são empregados e, dentre eles, se sobressai a utilização de pequenos núcleos, denominados micronúcleos, em contrapartida aos complexos e monolíticos núcleos existentes nos sistemas convencionais.

Os grupos de pesquisa em Arquitetura de Computadores e Sistemas Operacionais do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina, estão construindo uma máquina paralela – o Nó // (lê-se nó paralelo). Esse computador tem por objetivo apresentar soluções relativamente simples para diversos problemas enfrentados por outras arquiteturas.

Este trabalho enquadra-se dentro do projeto Nó // e possui dois objetivos principais:

- a) a construção de um simulador do multicomputador Nó //;
- b) a confecção de um sistema operacional para essa arquitetura, formado por micronúcleo distribuído, com interface de programação compatível com o UNIX.

Colocam-se diversas justificativas a essas tarefas:

- A utilização do simulador permitirá o desenvolvimento de *software* (sistema operacional e aplicações) para o multicomputador antes do término de sua construção.
- O simulador possibilitará a análise da carga e das interações entre os componentes do multicomputador.
- A confecção do sistema operacional comprovará a adequação do multicomputador para sistemas que exploram o paralelismo em máquinas de memória distribuída.
- O fato do sistema operacional ser compatível, a nível de programação, com UNIX, permitirá a importação de uma vasta gama de aplicações existentes.

- O sistema operacional distribuído oferecerá um ambiente de execução paralela que permitirá o andamento de diversas pesquisas que buscam explorar o paralelismo.

Este texto é constituído de seis capítulos, os quais são resumidos a seguir.

- O capítulo 2 introduz o estado da arte na Arquitetura de Computadores e Sistemas Operacionais. Tenta-se caracterizar a necessidade e conveniência de se utilizar máquinas paralelas, classificando-as segundo o grau de acoplamento de seus componentes. Aborda-se o conceito de sistemas operacionais distribuídos e o enfoque de se confeccionar tais sistemas com micronúcleo distribuído pelos seus processadores.
- O capítulo 3 apresenta o multicomputador Nó // sendo feita uma comparação de sua arquitetura com outras tradicionais.
- O capítulo 4 discute a necessidade de um simulador, apresenta suas características e a sua implementação.
- O capítulo 5 introduz o sistema operacional CRUX com micronúcleo distribuído para o Nó //. O sistema operacional é explicado subdivido em suas peças principais: o micronúcleo, a biblioteca e o servidor CRUX. São descritas também as etapas necessárias para a carga inicial do sistema operacional.
- O capítulo 6 contém as conclusões do trabalho desenvolvido. São apresentadas propostas de extensões do sistema operacional e discutidas as contribuições oferecidas.



## **2. Estado da Arte**

A construção de computadores digitais é um processo que se iniciou há quarenta anos, no final da segunda grande guerra mundial. Essas máquinas utilizavam um modelo proposto por John von Neumann, que consistia de um único processador conectado à memória e a outros periféricos através de um barramento.

O desenvolvimento dos computadores costuma ser classificado em gerações sucessivas, conforme a tecnologia aplicada para a construção dos seus processadores. A primeira geração de computadores, caracterizada pelo uso de válvulas, a segunda geração, pelo de transistores, e a terceira geração, pelo de circuitos integrados, não apresentaram muitas propostas de variação do modelo de von Neumann. Entretanto, com o advento da quarta geração de computadores e dos circuitos de larga integração, o preço dos processadores caiu vertiginosamente, permitindo o desenvolvimento de arquiteturas alternativas de computadores constituídos por mais de um processador.

### **2.1. Programação Concorrente**

A concorrência da multiprogramação é caracterizada pela simulação do paralelismo ou, em outras palavras, pelo paralelismo lógico. Dois processos são ditos concorrentes se suas execuções se sobrepõem no tempo, sendo irrelevante se as operações individuais de cada processo são realmente paralelas ou entrelaçadas no tempo. Toda vez que a primeira operação de um processo é iniciada antes que a última operação de outro seja completada, os dois processos são concorrentes [HAN73 - pág. 57].

O aumento do *throughput* possibilitado pela multiprogramação é obtido em detrimento do desempenho dos processamentos individuais, o que respondia na época da introdução desse conceito à expectativa de compartilhamento de um recurso dispendioso – o processador. Porém, o surgimento de computadores com vários processadores possibilitou a existência do paralelismo real, o que viabilizou os novos critérios de maximização do desempenho de aplicações específicas.

A necessidade atual da maximização do desempenho das aplicações é devida, principalmente, à tendência dos sistemas no sentido ascendente de sofisticação e complexidade. A maior parte da computação existente é confinada a níveis de processamento de dados e informação. A tendência é de promover o uso de computadores na direção do processamento de conhecimento e inteligência onde é consensual que o grau de paralelismo explorável é muito maior [HWA85 - pág. 5].

A figura 2-1 esboça as relações entre níveis de processamento.

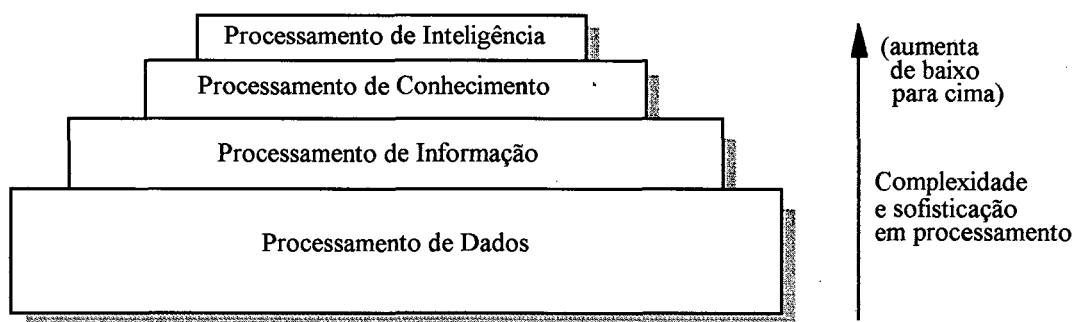


Figura 2-1. Níveis de processamento.

Brinch Hansen já observava em 1973 que os principais obstáculos para a utilização da concorrência em instalações de computadores eram a economia e imaginação humana. Afirmava que, ainda que os aperfeiçoamentos da tecnologia pudessem eventualmente tornar possível um alto grau de concorrência em nossas computações, devemos sempre tentar particionar nossos problemas conceitualmente em um número moderado de atividades seqüenciais que possam ser programadas separadamente e então conectadas livremente para execução concorrente [HAN73 - pág. 31].

Atualmente, a economia não é uma restrição grave à utilização da concorrência, porém a construção de grandes peças de *software* constituídas de múltiplas atividades seqüenciais é muito difícil, principalmente porque é necessária a preocupação com questões como exclusão mútua, condições de competição (*race conditions*), impasses (*deadlocks*) e inanição (*starvation*).

Uma tendência crescente na construção de sistemas é a utilização do paradigma de programação orientada a objetos, no qual os programas são organizados como coleções de objetos cooperantes. Segundo os defensores desse paradigma, ele pode aliviar os problemas da concorrência ocultando-a dentro de abstrações reutilizáveis [BOO91 - pág. 66].

## 2.2. Sistemas com Múltiplos Processadores

Sistemas com vários processadores podem ser classificados, a grosso modo, em função do grau de integração e acoplamento dos seus componentes: os *multiprocessadores* caracterizam-se por possuírem um espaço de endereçamento compartilhado por um número reduzido de processadores; nos *multicomputadores* cada processador possui sua memória privativa e canais de comunicação ligados através de redes de interconexão compostas por múltiplos canais bipontuais; as *redes de computadores* são formadas por computadores independentes interconectados através de redes concebidas como canais de comunicação compartilhados [AUS91].

Embora este trabalho oriente-se à exploração de uma implementação específica de um multicomputador, serão vistas, a fim de comparação, as principais características dos multiprocessadores e das redes de computadores.

### 2.2.1. Multiprocessadores

Os multiprocessadores (figura 2-2) são considerados sistemas fortemente acoplados devido à alta taxa de transferência de dados. Nesse tipo de arquitetura, dados são transferidos à velocidade de acesso à memória. Trocas de mensagens entre processos implicam, simplesmente, na cópia de dados de um trecho de memória para outro.

Esse tipo de arquitetura é eficiente mas não pode ser expandida para um número muito grande de processadores, por causa da dificuldade de construção do *hardware* e também pelo aumento da competição pelo barramento de memória [BEN90 - pág. 156].

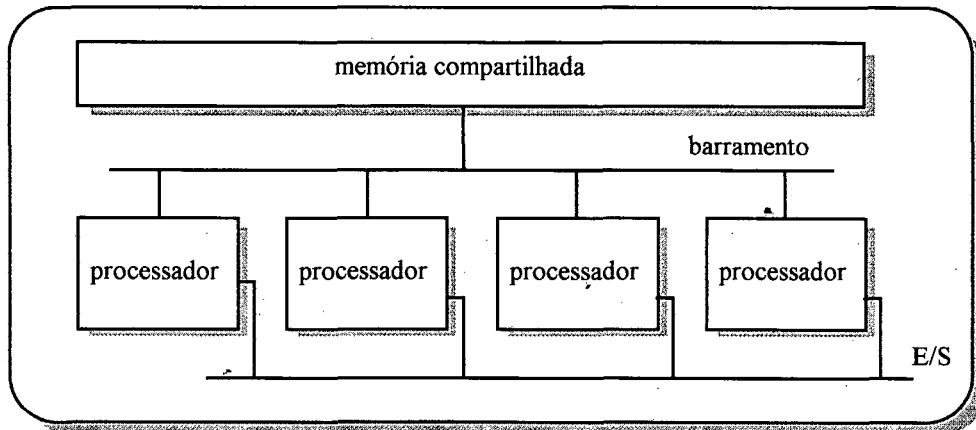


Figura 2-2. Multiprocessador: um sistema fortemente acoplado.

### 2.2.2. Redes de Computadores

As redes de computadores (figura 2-3) são a contrapartida dos multiprocessadores já que são consideradas sistemas muito fracamente acoplados. Nessa arquitetura, vários computadores são interligados através de redes de comunicação velozes.

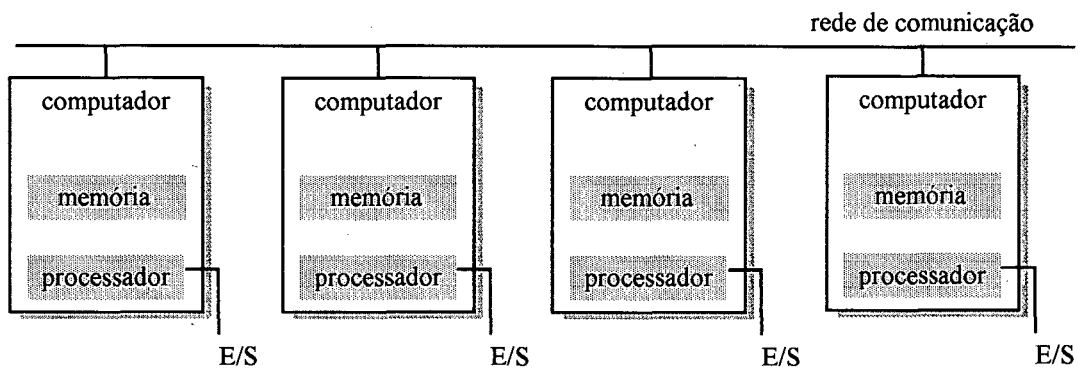


Figura 2-3. Rede de computadores: um sistema muito fracamente acoplado.

Existe atualmente muito investimento para se conseguir um aumento na velocidade das redes de comunicação. Consideradas historicamente como gargalos nas comunicações, já se consegue redes velocíssimas utilizando-se tecnologias recentes como a fibra ótica. Esse aumento de velocidade tem tornado esse tipo de arquitetura muito popular.

Entretanto, a maioria dessas redes de computadores são utilizadas com *sistemas operacionais de rede* [SIL94 - pág. 480], apenas para compartilhar periféricos dispendiosos como impressoras e discos magnéticos. Nessa filosofia não existe cooperação a nível de processamento. Por exemplo, ocorre freqüentemente o fato de um computador estar sobrecarregado com vários processamentos apesar de haver outros computadores ociosos.

### **2.2.3. Multicomputadores**

Os multicomputadores (figura 2-4) – objeto de estudo deste trabalho – são considerados sistemas fracamente acoplados pelo fato de não possuírem memória compartilhada. Devido a essa característica, alguns autores não fazem distinção entre multicomputadores e redes de computadores, considerando ambos como sistemas fracamente acoplados.

Apesar de conceitualmente parecidos, existe uma separação clara quando se considera a relevância dos canais de comunicação que interligam tais arquiteturas. As redes de computadores possuem canais de comunicação compartilhados pelos seus nós processadores, enquanto um multicomputador é constituído de canais de comunicação exclusivos para comunicações bipontuais.

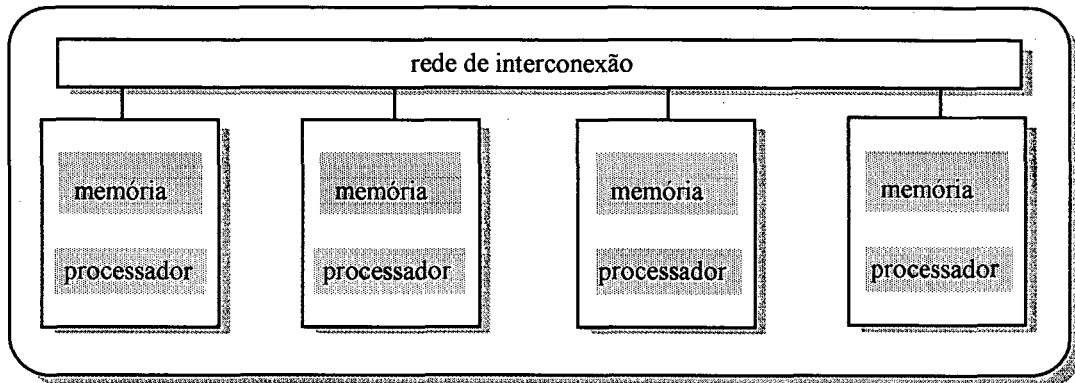


Figura 2-4. Multicomputador: um sistema fracamente acoplado.

Classificações do tipo sistemas fortemente, fracamente ou muito fracamente acoplados são relativas. Entretanto, elas nos ajudam a caracterizar melhor as máquinas paralelas conforme suas principais características, vantagens e restrições.

Dessa forma, parece claro que devido a limitações nas velocidades das comunicações, as redes de computadores não se adaptam perfeitamente a processos que se comunicam intensamente: O gargalo na velocidade de comunicação reduziria o ganho conseguido pelo paralelismo real.

A razão *tempo de processamento* dividido por *tempo de comunicação* é denominada *granularidade de processamento* [TAY89 - pág. 89]. As redes de computadores e os multicomputadores são sistemas voltados para aplicações com grande ou média granularidade de processamento, enquanto os multiprocessadores são ideais para aplicações com granularidade fina.

#### 2.2.4. Redes Estáticas e Redes Dinâmicas

Pode-se classificar os multicomputadores conforme a sua rede de interconexão levando-se em consideração se é possível ou não alterar sua topologia dinamicamente. Dessa forma, os *multicomputadores com rede de interconexão estática* diferem dos *multicomputadores com rede de interconexão dinâmica* pelo fato dos primeiros não poderem alterar sua topologia redirecionando suas conexões, enquanto que os últimos possuem flexibilidade para interligar seus nós dinamicamente ativando os elementos de chaveamento da rede [FEN81].

Quatro topologias clássicas de redes de interconexão estáticas são mostradas na figura 2-5 com os vértices representando os nós processadores e as arestas as interligações que os conectam.



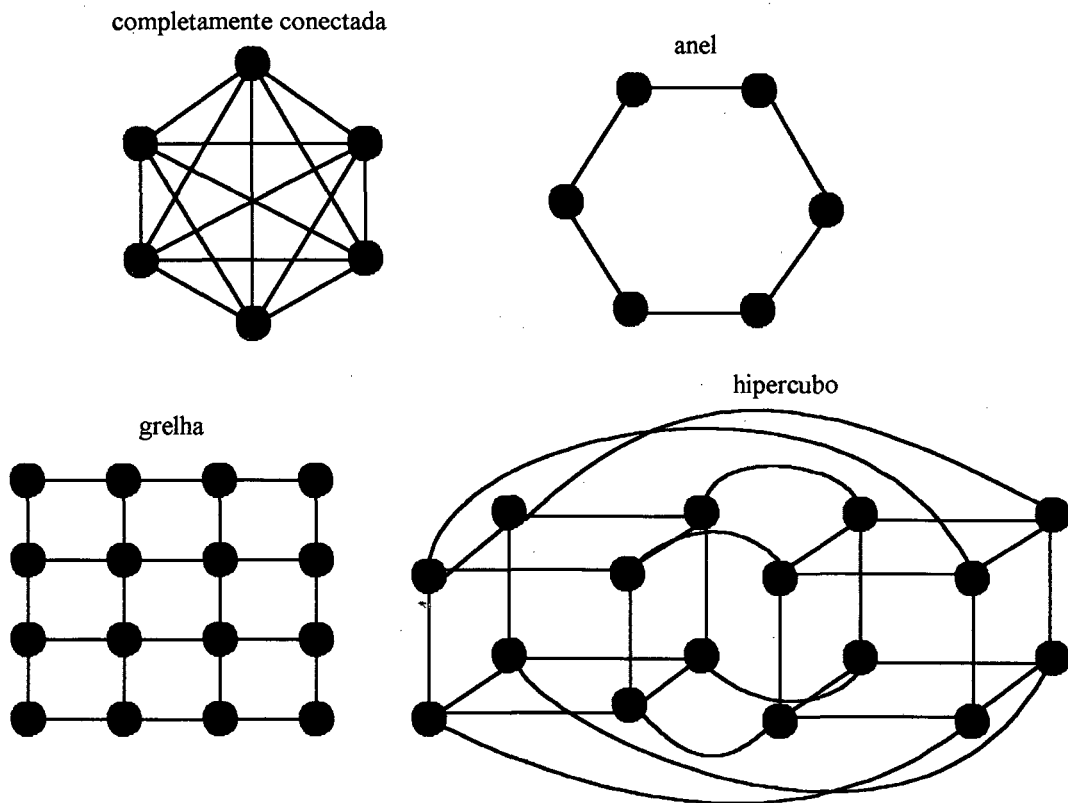


Figura 2-5. Rede completamente conectada, anel, grelha e hipercubo.

Do ponto de vista do desempenho, a primeira topologia, a *completamente conectada*, é a ideal, pois as mensagens enviadas entre nós são recebidas velozmente devido ao fato de cada nó possuir ligações com todos os outros. Entretanto, o custo dessa configuração é muito alto e, dependendo da quantidade de nós, até mesmo inviável, pois o número total de ligações é proporcional ao quadrado do número de nós [SIL94 - pág. 483, HWA93 - pág. 88].

Para solucionar o problema da limitação do número de ligações entre nós, implementam-se topologias que são parcialmente conectadas, isto é, nem todos os nós possuem ligações bipontuais entre si.

Em uma rede em anel, por exemplo, cada nó é fisicamente conectado apenas com seus dois vizinhos. O número de conexões cresce linearmente com o número de nós.

A *grelha* possui uma rede de interconexão plana, bidimensional, pois as conexões entre seus nós podem ser representadas num plano, sem cruzamento de suas linhas. Ela se adequa a problemas que por natureza são inerentemente bidimensionais, como, por exemplo, o reconhecimento de fotografias [TAN92 - pág. 372]. A *grelha* apresentada possui 16 nós, cada qual com no máximo 4 ligações com seus vizinhos.

O *hipercubo* possui uma representação mais complexa que não pode ser visualizada no campo bidimensional<sup>1</sup>. O exemplo apresentado é o de um 4-cubo – um hiper-cubo com 16 nós, cada qual com 4 ligações com outros nós.

Devido ao fato de não haver ligações bipontuais entre todos os nós, nas redes parcialmente conectadas, uma mensagem encaminhada de um nó qualquer para outro, necessita freqüentemente passar por nós intermediários antes de chegar ao seu destino. Esse fato, além de prejudicar a eficiência da comunicação, implica em outros problemas que necessitam ser solucionados:

- existe a necessidade de se escolher a melhor rota entre dois nós;
- um nó precisa reservar espaço para armazenar mensagens que não lhe são destinadas;
- as mensagens ficam limitadas ao tamanho máximo que um nó intermediário pode armazenar;

---

<sup>1</sup> Exceto em suas configurações mais básicas – 0-cubo, 1-cubo, 2-cubo e 3-cubo – que constituem na maioria das vezes objeto de estudo e curiosidade devido ao pouco número de nós.

- dois processos que se comunicam intensamente precisam ficar em nós próximos.

Uma outra forma de enfrentar o problema da limitação do número de conexões é estabelecendo uma forma dinâmica dos nós se interligarem. As redes dinâmicas são implementadas utilizando elementos de chaveamento e árbitros ao longo de seu caminho. Segundo Kai Hwang [HWA93 - pág. 80], enquanto as redes estáticas se ajustam melhor em computadores onde os padrões de comunicação são previsíveis, as redes dinâmicas servem para aplicações de propósito geral, pois podem implementar os padrões de comunicação baseados na demanda dos programas.

Uma rede de interconexão dinâmica que tem se tornado bastante popular é o *crossbar*. Ele pode ser identificado como um comutador de conexões, pois seu funcionamento é semelhante a um painel de controle telefônico o qual oferece conexões dinâmicas entre pares de telefones.

A figura 2-6 mostra um *crossbar* genérico  $n \times n$ , que interliga  $n$  processadores. Os processadores são representados pelos círculos negros e os pontos de conexão pelos retângulos, onde os hachurados representam os pontos conectados. Dessa forma, no *crossbar* apresentado, os processadores 2 e 3 possuem suas entradas e saídas conectadas de forma a poderem estabelecer uma comunicação bidirecional.

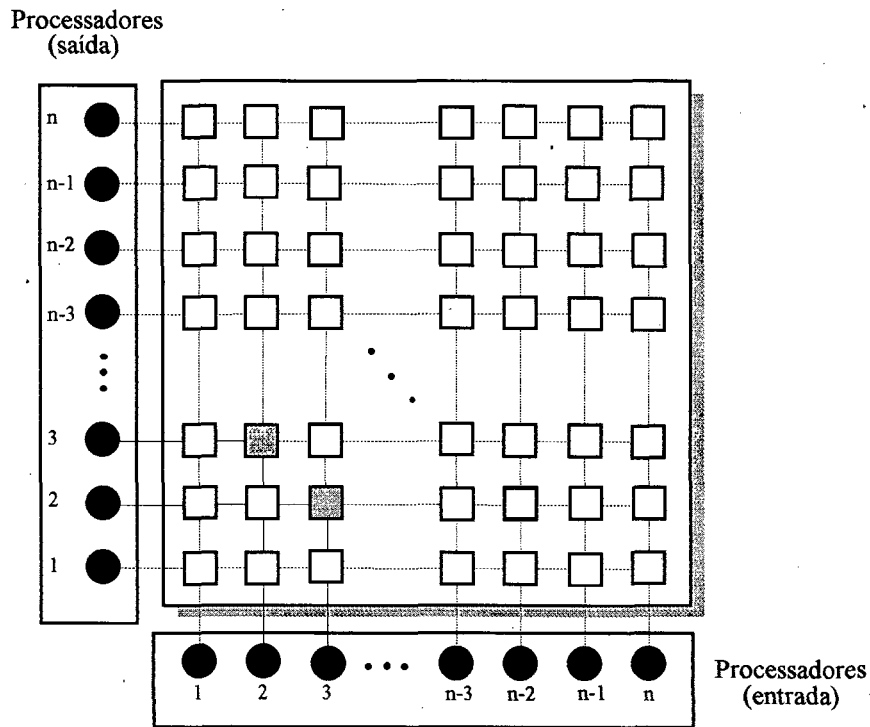


Figura 2-6. Comutador de conexões do tipo crossbar.

Redes dinâmicas *crossbar* ainda são dispendiosas devido ao fato de sua complexidade aumentar proporcionalmente ao quadrado do número de elementos a serem conectados, entretanto, com a tecnologia ótica e microeletrônica avançando, é previsto um aumento considerável em sua utilização.

### 2.3. Sistemas Operacionais Distribuídos

O compartilhamento de recursos computacionais é uma necessidade e cabe aos sistemas operacionais torná-lo transparente. Essa transparência é um dos objetivos mais cobiçados pelos usuários. Por exemplo, usuários desejam utilizar os recursos computacionais disponíveis, sem necessitar conhecer suas localizações. Além disso, se há vários usuários compartilhando um recurso, cada um deles deve possuir a sensação de ser o único a utilizá-lo. Mais ainda, se vários recursos do mesmo tipo estiverem disponíveis, um usuário deseja usá-los de forma a diminuir o seu tempo de processamento, sem precisar demandá-los explicitamente.

Sistemas operacionais que executam em coleções de processadores que não compartilham memória, como os multicomputadores e redes de computadores, e que realizam a tarefa de tornar transparente seu uso, são denominados *sistemas operacionais distribuídos* [MUL88 - pág. 53].

Existe atualmente uma grande demanda na pesquisa e desenvolvimento de sistemas operacionais distribuídos, pois, devido à complexidade maior das máquinas paralelas, muitas das soluções que eram adotadas nos sistemas tradicionais para alcançar tais transparências se tornaram inválidas.

O desafio para os próximos anos é produzir uma nova geração de sistemas operacionais que reúna uma coleção de dispositivos e processadores, e apresente aos usuários uma única máquina integrada, ao invés de um grupo de máquinas distintas que se comunicam através de algum protocolo de rede.

### 2.3.1. O Enfoque Micronúcleo

Outrora, sistemas operacionais eram confeccionados de forma totalmente desestruturada. Entretanto, com a crescente complexidade embutida nos sistemas operacionais passou-se a utilizar o desenvolvimento em camadas.

A figura 2-7 mostra a estrutura do primeiro sistema construído em camadas, o *THE* [TAN92 - pág. 21]. Desenvolvido em 1968 por *Dijkstra* e seus estudantes, o *THE* era um sistema operacional *batch*, composto por seis camadas.

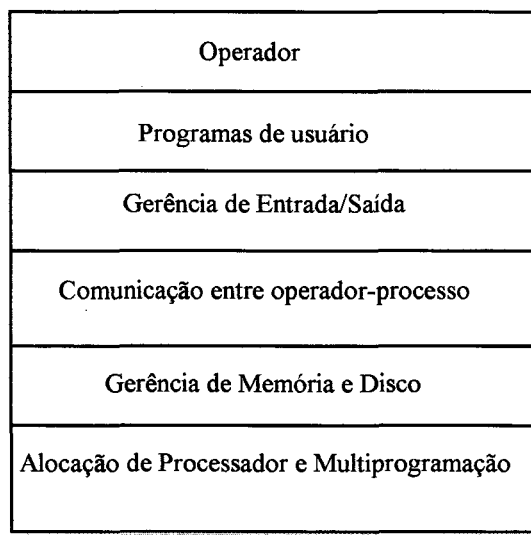


Figura 2-7. A estrutura em camadas do sistema operacional THE.

Na técnica de hierarquia de camadas, grupos de funções de sistemas operacionais são divididos em camadas que se comunicam apenas com as adjacentes superiores e inferiores. A modularidade embutida nesse tipo de sistema, garante que a camada inferior oferece serviços necessários à superior. Por exemplo, no *THE*, a camada de gerência de memória e disco, alocava espaços em memória e em páginas no disco. Isso possibilitava às camadas superiores não precisar se preocupar com essas tarefas.

Atualmente, para conseguir maior flexibilidade, existe uma crescente tendência de se construir sistemas operacionais com um núcleo mínimo, um *micronúcleo*, apenas com as funções que necessitam realmente executar em modo supervisor e em espaço privilegiado [VAR94 - pág. 122]. Essas funções são implementadas de forma a fornecer mecanismos necessários e suficientes para as peças externas ao núcleo implementarem suas políticas.

Essa filosofia apresenta duas grandes vantagens. A primeira é que, ao deixar a implementação das políticas fora do núcleo, é possível a um sistema operacional emular diversos outros. Essa característica é importante nos dias atuais, pois sistemas operacionais novos, ao serem lançados, necessitam emular outros já estabelecidos para serem viáveis comercialmente até o surgimento de aplicações específicas para ele.

Um exemplo é o sistema operacional *Mach*. Seu micronúcleo, seus sub-sistemas e os processos de usuários compõem um sistema de três camadas que podem ser vistos esquematicamente na figura 2-8 [TAN92 - pág. 639, SIL94 - pág. 660].

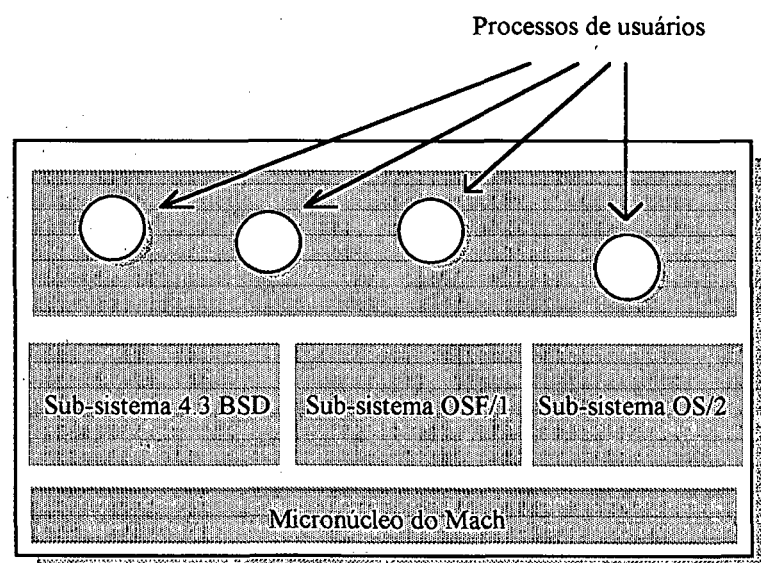


Figura 2-8: O micronúcleo do Mach e seus sub-sistemas.

A segunda vantagem é que a filosofia da utilização de um micronúcleo se adapta perfeitamente ao modelo cliente-servidor, o qual, por sua vez, se adequa aos sistemas operacionais distribuídos. No modelo cliente-servidor, o sistema operacional é dividido em diversas partes cada uma das quais se encarrega de um determinado serviço do sistema, podendo essas partes serem distribuídas nos diversos processadores da máquina.

Conforme pode ser observado na figura 2-9, a técnica da utilização de micronúcleos em sistemas distribuídos substitui a estratificação vertical, inerente da técnica da confecção de sistemas em camadas, pela horizontal. Essa é uma característica importante, pois embute grande flexibilidade na confecção e na configuração desses sistemas de forma a conseguir se escapar da rigidez existente na hierarquia em camadas.

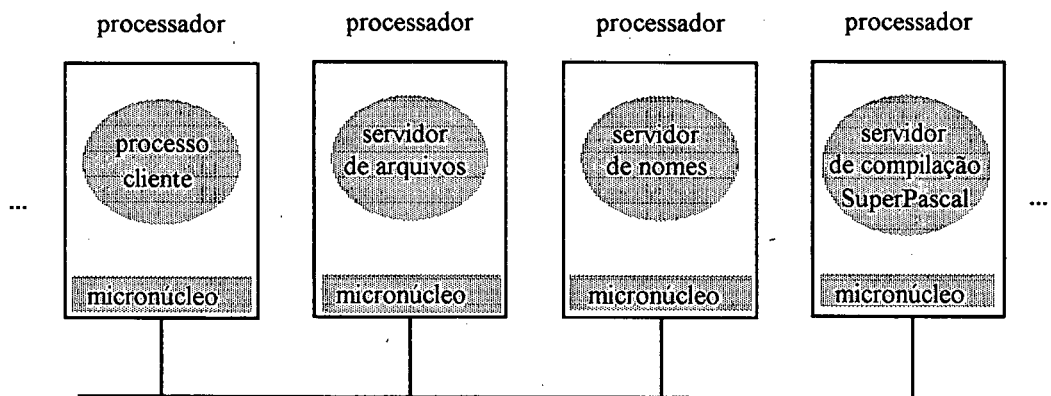


Figura 2-9. O modelo cliente-servidor em um sistema distribuído.

O interessante nesse esquema é que um determinado servidor pode ser replicado em diversos processadores possibilitando uma maior eficiência e tolerância a falhas. Uma pane em um desses servidores replicados significa perda de desempenho, uma degradação parcial, ao invés da interrupção do serviço [MUL88 - pág. 55].



Uma outra característica é o fato da distribuição de serviços permitir uma extensibilidade maior do sistema. Serviços podem ser acrescentados e retirados gradativamente tornando o sistema maleável e acessível a uma gama maior de usuários.

As funções básicas de um micronúcleo de sistemas operacionais distribuídos são apresentadas a seguir [VAR94 - pág. 122]:

- *Gerência de Interrupções e Código Dependente do Processador*

Empacotar todo o código dependente do *hardware* no núcleo, torna o sistema protegido dos processos não privilegiados – que executam em modo usuário –, além de prover uma enorme portabilidade do sistema.

A identificação e o tratamento das interrupções ocorridas no sistema são tarefas executadas pelos micronúcleos. Entretanto, alguns sistemas, como o CHORUS [POU94], permitem que processos externos ao núcleo executem rotinas manipuladoras de interrupção. Essa característica se afasta da definição de micronúcleos, pois permite a códigos externos a eles executarem em modo supervisor. A argumentação utilizada, é que dessa forma é oferecida grande flexibilidade ao sistema, pois, os controladores de dispositivos podem se situar externamente aos núcleos.

- *Gerência de Processos*

Na filosofia de implementação de micronúcleos, pretende-se colocar em seu interior apenas os mecanismos essenciais para gerenciamento de processos, e um escalonador de processos é parte importante de um gerente de processos.

Um conceito moderno e que tem tomado corpo recentemente, é a utilização de diversos fluxos de execução independentes dentro de um processo. Esses fluxos, denominados *threads* [TAN92 - pág. 507], compartilham informações existentes no processo como a memória e os arquivos abertos. Dessa forma, os micronúcleos modernos, costumam implementar também escalonadores de *threads*.

Alguns micronúcleos vão mais além, mantendo em seu interior apenas o *despachante* – responsável pela troca de contexto durante o escalonamento – deixando o escalonador de fora.

#### • *Comunicação entre Processos*

Sistemas operacionais distribuídos são implementados em grupos de processadores que não possuem memória compartilhada e, por isso, o paradigma básico de comunicação entre processos é o da troca de mensagens. O micronúcleo desses sistemas deve fornecer funções para comunicação entre processos que sejam suficientemente flexíveis para os sistemas operacionais implementarem seus serviços de comunicação.

Duas funções básicas de comunicação entre processos são o envio e a recepção de mensagens. Entretanto, diversas alternativas podem ser adotadas para essas funções como, por exemplo, se elas são bloqueantes ou não, se o endereçamento é direto ou através de caixas postais, se é possível enviar mensagens para uns (*multicasting*) ou para todos (*broadcasting*).

### • *Gerência de Memória*

Historicamente, memória é um recurso escasso aos processos. Técnicas complexas de gerenciamento de memória utilizando memória secundária, como a paginação, foram temas de vários estudos e estão implementadas em diversos sistemas operacionais. Na filosofia da implantação de micronúcleo, costuma ser deixado em seu interior apenas o código que controla o *hardware* de paginação, ficando o paginador, que implementa as estratégias de troca de páginas, fora do núcleo.

### **2.3.2. Comunicação entre Processos**

Como foi discutido anteriormente, os micronúcleos dos sistemas operacionais distribuídos fornecem os mecanismos básicos para comunicações entre processos. Utilizando esses mecanismos, os sistemas operacionais implementam as suas chamadas de sistema referentes às comunicações entre processos. As linguagens de programação, por sua vez, utilizam essas chamadas de sistema para fornecer aos usuários formas de comunicação entre processos.

Parece claro que os mecanismos fornecidos pelos micronúcleos devem ser flexíveis e eficientes o suficiente para os sistemas operacionais implementarem suas chamadas de sistema, que também devem ser flexíveis e eficientes o suficiente para o uso das linguagens de programação.

As linguagens de programação podem possuir funções que espelham diretamente as chamadas que o sistema operacional fornece. Por exemplo, se existe no sistema as chamadas *send* e *receive* para, respectivamente, enviar e receber mensagens entre processos, uma linguagem implementada nesse sistema pode possuir embutidas duas funções correspondentes a elas. Porém, essa filosofia amarra a linguagem a um determinado sistema, além de, normalmente, dar pouca maleabilidade aos programadores.

A tendência moderna é que as linguagens forneçam abstrações de mais alto-nível aos programadores. Apesar dessa filosofia muitas vezes prejudicar o desempenho devido a chamadas de sistema inadequadas de alguns sistemas operacionais, ela possui vantagens maiores com relação à produtividade do programador. Uma analogia que pode ser feita é a da linguagem *assembly* com relação a uma linguagem de alto-nível. Apesar dos *overheads* destas últimas, as vantagens oferecidas e enumeradas pela engenharia de *software* autorizam a utilização de *assembly* apenas para aplicações ou rotinas específicas onde a velocidade e espaço em memória são essenciais.

Um exemplo interessante de uma abstração de alto-nível oferecida por uma linguagem é proposta em *Orca*. Essa linguagem propõe que a comunicação entre processos se faça através de objetos compartilhados. Em [BAL88] são discutidas formas eficientes de se implementar semelhante abstração em máquinas que não possuem memória compartilhada. No caso dessa linguagem, consegue-se uma melhor implementação se o sistema operacional fornecer uma forma de comunicação confiável com *broadcasting*. Embora no caso de *Orca* o *broadcasting* não seja essencial, torna sua implementação mais eficiente. Entretanto, a falta de chamadas específicas pode, em alguns casos, inviabilizar a implementação de uma determinada linguagem.

### 2.3.3. Chamada de Procedimento Remoto

No modelo cliente-servidor, processos usuários requisitam serviços de processos servidores do sistema. Muitos sistemas operacionais oferecem uma abstração que facilita essa forma de comunicação com o objetivo de apresentar a comunicação entre esses processos como se fossem chamadas de procedimento. Naturalmente, se os processos estão colocados em processadores com memórias privadas distintas, a simulação da chamada é implementada através da comunicação efetiva entre os processos. Essas comunicações são ocultadas pelos sistemas operacionais e o mecanismo recebe a denominação de *chamada de procedimento remoto (remote procedure call – RPC)*.

A maneira mais difundida de ocultar o mecanismo que ocorre durante uma comunicação cliente-servidor é fornecendo uma biblioteca de procedimentos que podem ser ligados junto com os programas usuários. Esses procedimentos, denominados *stubs*, criam uma mensagem que contém informações como o tipo de serviço requisitado e os parâmetros fornecidos, e chamam o núcleo para enviá-la ao servidor.

A figura 2-10 esquematiza uma chamada de procedimento remoto [COU88 - pág. 99, TAN92 - pág. 421].

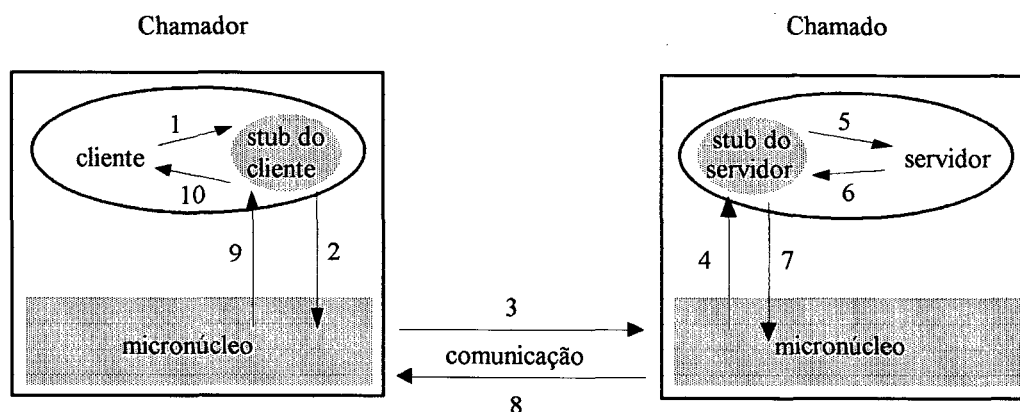


Figura 2-10. Uma chamada de procedimento remoto.

1. O processo cliente requisita serviço ao processo servidor fazendo uma chamada de procedimento.
2. O *stub do cliente* empacota a requisição e seus parâmetros em uma mensagem e pede para o micronúcleo enviá-la.
3. O micronúcleo envia a mensagem.
4. Do outro lado da linha, o micronúcleo do nó onde se encontra o processo servidor recebe a mensagem e a envia para o *stub do servidor*.
5. O *stub do servidor* desempacota a mensagem, formando uma chamada de procedimento ao servidor.
6. O servidor executa o serviço e retorna o resultado para o *stub do servidor*.
7. O *stub do servidor* empacota a resposta em uma mensagem e pede para o micronúcleo enviá-la.
8. O micronúcleo envia a mensagem.

9. O micronúcleo do nó onde se encontra o processo cliente recebe a mensagem e a envia para o *stub do cliente*.
10. O *stub do cliente* desempacota a resposta e a retorna para o processo cliente.

Para o processo cliente, todas essas etapas ocorreram de forma transparente. Ele simplesmente faz uma chamada de procedimento, aguarda ela ser executada e recebe a resposta. Essa abstração garante uma produtividade muito maior do que se todas essas etapas tivessem que ser explicitamente programadas em cada processo que executasse nesse sistema.

No exemplo anterior, o processo cliente, ao requisitar um serviço, fica bloqueado até o serviço ser completado. Por outro lado, o processo servidor também aguarda bloqueado a chegada de uma requisição de serviços.

O paradigma de comunicação utilizado em RPC é síncrono. Nesse conceito, o processo que deseja se comunicar executa uma função de transmissão de mensagens que bloqueia o processo até que o receptor esteja apto a recebê-la. Por outro lado, o processo que deseja receber uma mensagem antes que esta seja enviada, fica suspenso até que isso aconteça. Essa característica força um *rendez-vous* (encontro em francês) entre o emissor e o receptor.

O RPC se torna cada vez mais importante na medida em que o modelo cliente-servidor se fortalece nos sistemas operacionais distribuídos. A importância desse modelo tem vindo sistematicamente à tona a ponto de autores preverem que os futuros sistemas operacionais distribuídos serão orientados a objetos e baseados no modelo cliente-servidor, isto é, os objetos serão gerenciados pelos serviços, e os clientes farão operações nesses objetos através da mediação de processos servidores [MUL88 - pág. 59].

### 3. Arquitetura do Multicomputador Nó //

Neste capítulo é apresentada a arquitetura do multicomputador Nó // proposta em [COR93], sendo feita uma breve comparação com outros multicomputadores.

A especificação do multicomputador Nó // é devida a Hermann Lucke do grupo de Arquiteturas de Computadores do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal de Santa Catarina.

Conforme pode ser observado na figura 3-1, a arquitetura do Nó // é constituída de diversos nós processadores homogêneos, conectados entre si por intermédio de dois dispositivos distintos: um comutador de conexões do tipo *crossbar* e um barramento de serviços.

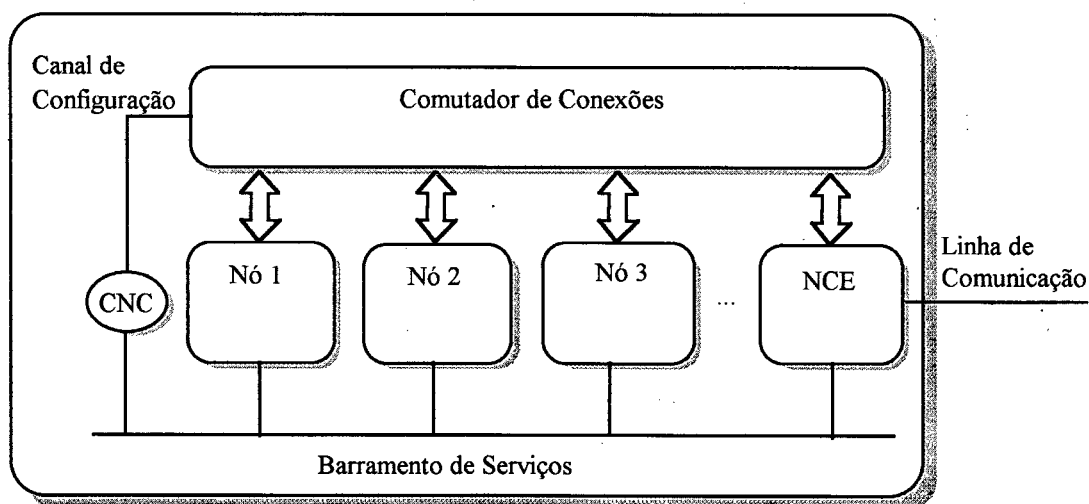


Figura 3-1. A arquitetura do Nó //.



### 3.1. Os Nós

Cada nó dessa arquitetura possui um processador, memória RAM privativa e uma pequena memória ROM, com seu respectivo microcódigo. Na figura 3-2, que apresenta detalhadamente um nó, pode ser observado que todos eles possuem também um ou vários canais de comunicação bidirecionais conectados ao comutador de conexões, para troca de mensagens. Além disso, cada nó possui uma interligação com o barramento de serviços, cuja função será descrita em detalhes mais adiante.

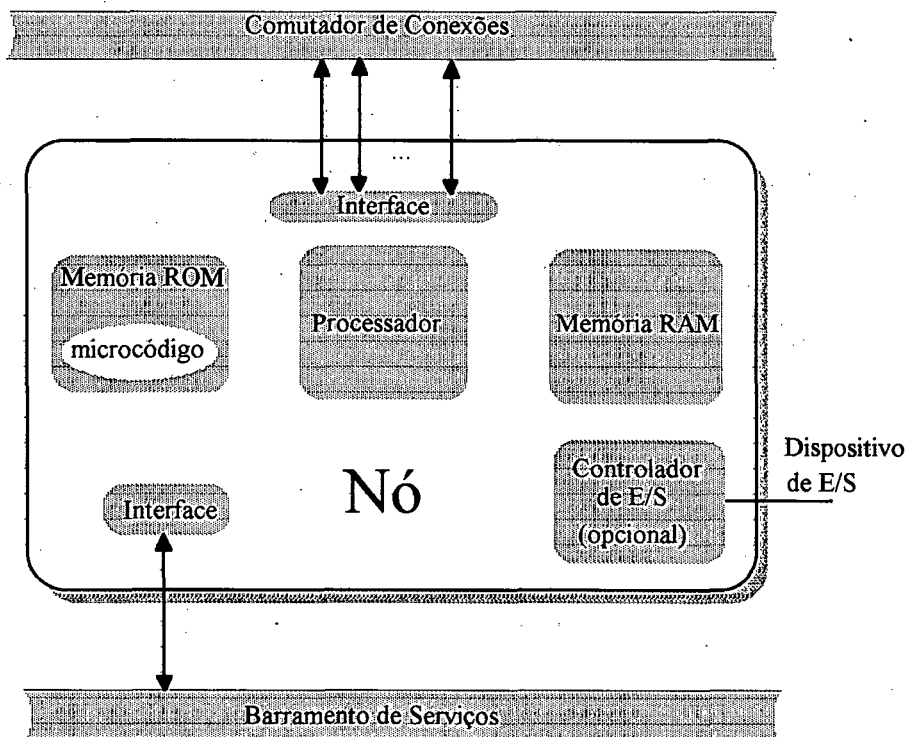


Figura 3-2. A estrutura interna de um nó.

### 3.2. O Comutador de Conexões

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

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

### 3.3. O Controlador de Nós e Conexões (CNC)

Um dispositivo inteligente, denominado *Controlador de Nós e Conexões* (CNC), é responsável no Nó // por comandar o comutador de conexões através de um *canal de configuração* (figura 3-1). Todos os demais nós podem requisitar conexões bipontuais entre si se comunicando com o CNC através do barramento de serviços, que funciona como via de comunicação entre um nó qualquer e esse dispositivo.

De forma semelhante a um nó qualquer da arquitetura, o CNC é constituído de processador, memória RAM e memória ROM com o seu microcódigo.

Entretanto, a capacidade de comandar o comutador de conexões o distingue dos demais nós. Devido a essa importante atribuição, ele é dedicado exclusivamente às tarefas relacionadas com essas funções, não possuindo outras conexões com o comutador de conexões e, conseqüentemente, não se comunicando com outros nós através desse dispositivo.

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

### 3.4. O Barramento de Serviços

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

As mensagens que trafegam por esse barramento e que são recebidas e tratadas pelo CNC são as seguintes:

<b>Mensagem</b>	<b>Parâmetro</b>	<b>Retorno</b>
CONNECT	nid	nada

Essa mensagem é uma requisição ao CNC para que ele estabeleça uma conexão bipontual entre o nó do processo que pediu a conexão e o nó *nid*. A conexão só é realmente estabelecida quando o processo no nó *nid* também pede uma conexão e, enquanto isso não ocorre, a requisição permanece numa fila de requisições de conexões geridas pelo CNC. Somente quando a conexão é realmente efetivada é retornado uma mensagem de confirmação para ao processo que requisitou a conexão.

<b>Mensagem</b>	<b>Parâmetro</b>	<b>Retorno</b>
-----------------	------------------	----------------

CONNECT_ANY	nenhum	nid
-------------	--------	-----

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

<b>Mensagem</b>	<b>Parâmetro</b>	<b>Retorno</b>
-----------------	------------------	----------------

DISCONNECT	nid	nada
------------	-----	------

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

<b>Mensagem</b>	<b>Parâmetro</b>	<b>Retorno</b>
-----------------	------------------	----------------

ALLOCATE	nid	nada
----------	-----	------

Essa mensagem é um pedido para o CNC reservar o nó *nid* para o processo que enviou a mensagem. Após reservar o nó, o CNC estabelece, uma conexão entre o nó reservado e o nó onde se encontra o processo que pediu a alocação.

Se o nó requisitado já está alocado, essa chamada retorna uma mensagem de erro.

<b>Mensagem</b>	<b>Parâmetro</b>	<b>Retorno</b>
ALLOCATE ANY	nenhum	nid

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

<b>Mensagem</b>	<b>Parâmetro</b>	<b>Retorno</b>
DEALLOCATE	nid	nada

Essa mensagem é um pedido ao CNC para que ele libere o nó onde executa o processo que enviou a mensagem.

### **3.5. O Nó de Comunicação Externa (NCE)**

Um nó especial dessa arquitetura possui a tarefa de fazer a comunicação com o "mundo externo" através de uma linha de comunicação. Esse nó é denominado Nó de Comunicação Externa (NCE)

Máquinas com esta arquitetura podem ser usadas como nós independentes em redes locais (e é esta a vocação das mesmas), conforme o modelo de *pool de processadores* [TAN92 - pág. 531]. Os processadores podem ser alocados por demanda, segundo a necessidade dinâmica dos processos dos programas paralelos.

### 3.6. O Nó // e outros Multicomputadores

Conforme foi apresentado em 2.2.4, os multicomputadores podem ser subdivididos em multicomputadores com rede de interconexão dinâmica e com rede de interconexão estática. Foram levantados alguns problemas existentes nas redes estáticas com relação à necessidade de armazenamento temporário e roteamento de mensagens.

No Nó //, em um determinado instante, qualquer nó pode ser conectado diretamente a outro através do *crossbar*, eliminando *overheads* com roteamento e administração de espaço para armazenamento de mensagens para nós intermediários.

Não há armazenamento temporário de mensagens em nós durante o roteamento, portanto, não há necessidade de limitar o tamanho delas.

Mais ainda, a distância entre dois nós nesse tipo de arquitetura é sempre a mesma, o que facilita a colocação de processos em processadores, pois não existe o conceito de "nó mais próximo". O controle e a escolha de nós livres é efetuada pelo CNC, segundo um algoritmo simples.

Entretanto, se mesmo com todas as vantagens descritas, a opção for por alguma arquitetura com rede estática, ainda assim é possível utilizar-se o Nó //, configurando suas conexões conforme uma topologia estática<sup>2</sup>.

---

<sup>2</sup> Se o número de canais conectados a cada nó for insuficiente para simular a arquitetura estática, é necessário utilizar-se de software de forma a implementar um número de canais lógicos superior ao de canais físicos.

## 4. Simulador do Multicomputador Nó //

Neste capítulo justifica-se a necessidade da existência de um simulador do multicomputador Nó //, apresenta-se a configuração mínima da máquina simulada e, descreve-se o simulador implementado.

### 4.1. Necessidade de um Simulador

Apesar da arquitetura do Nó // ser de concepção simples se comparada com outras existentes ou propostas, ela é uma solução inovadora. Esse fato torna necessário o desenvolvimento de um sistema operacional e de aplicações para o Nó // desde o princípio, utilizando os vários nós processadores existentes na arquitetura no sentido de dividir as tarefas em processos que possam executar paralelamente. Eles necessitam de testes, ajustes, avaliações e refinamentos antes de serem considerados prontos para executar na máquina.

É importante que o desenvolvimento destes programas se dê em paralelo com a construção do *hardware* para que, num prazo pequeno, tenha-se um Nó // (*hardware*-sistema operacional-aplicações) em funcionamento.

Devido à necessidade constante de avaliações e testes nos programas, esse desenvolvimento simultâneo de *hardware/software* se inviabilizaria se não houvesse um programa simulador do Nó //. A tarefa desse programa é simular as funções básicas do *hardware* e microcódigo em ROM, de forma que o código desenvolvido sobre o simulador possa ser transportado posteriormente para o hardware, quando este estiver pronto, sem nenhuma (ou quase nenhuma) alteração.

Além disso, o simulador permitirá fazer uma coleta de dados sobre cada elemento da arquitetura do Nó //. Esses dados serão utilizados para alimentar outro simulador – para análise do desempenho – que está sendo desenvolvido utilizando ferramentas adequadas para esse tipo de simulação [MER95].

A utilização dos dois simuladores possibilitará mensurar velocidades e constatar eventuais gargalos na arquitetura proposta.

## 4.2. Sistema de Desenvolvimento

Ao se implementar o simulador, optou-se por fazê-lo simulando uma configuração mínima do Nó //. Essa configuração que mantém as propriedades essenciais do modelo proposto, conterà 32 nós, cada qual com 64 *Kbytes* de memória, e apenas um canal de comunicação ligando-os ao comutador de conexões. Além disso, considera-se o Nó // fisicamente conectado a uma estação de trabalho autônoma, conforme mostra a figura 4-1.

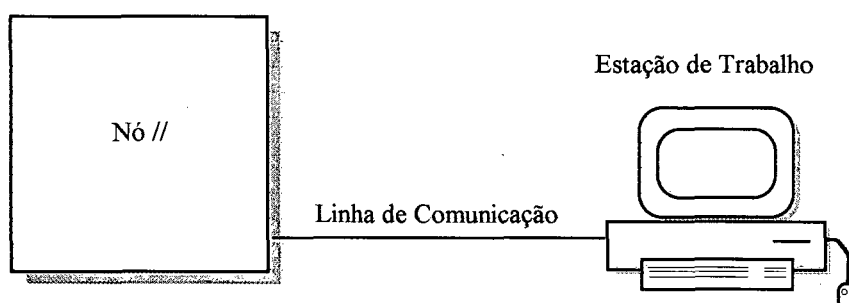


Figura 4-1. Nó // conectado a uma estação de trabalho.

Essa simplificação mantém a propriedade principal do modelo: uma tarefa disparada na estação de trabalho pode ser subdividida em processos que podem ser alocados em nós do Nó // e executados paralelamente entre si.



### 4.3. Implementação

Para implementar o simulador foram utilizados dois microcomputadores IBM-PC i486, um para executar o simulador sobre o sistema MS-DOS [DUN86] e o outro para servir de estação de trabalho utilizando o sistema operacional Linux [KIR94].

Duas linhas distintas poderiam ser seguidas na construção do simulador. A primeira seria desenvolvê-lo integralmente desde o princípio. A segunda linha seria alterar algum produto já existente, desde que ele oferecesse flexibilidade para que, sobre as suas bases pudesse se construir o simulador.

O segundo enfoque foi adotado ao constatar-se a adequação do sistema operacional XINU [COM84] como base para o simulador. (O Apêndice A resume as principais características do XINU.) As razões principais de sua escolha foram a simplicidade, a disponibilidade e a facilidade de alteração de seu código-fonte para atender as necessidades do simulador, que se resumem basicamente em duas:

- *Simulação dos Nós Processadores*

Existe a necessidade de simular cada nó processador do Nó //, incluindo o NCE e o CNC – que podem ser abstraídos como nós especiais. Além disso, é necessário a simulação do paralelismo real de processamento existente entre os nós processadores.

- *Simulação da Comunicação*

A comunicação entre os elementos processadores do Nó //, tanto pelos canais ligados ao comutador de conexões quanto pelo barramento de serviços necessitam ser simulados.

O código-fonte do XINU é escrito em linguagem C, com pequenos trechos em linguagem *assembly*, e encontra-se disponível para várias plataformas tanto à nível de máquinas (IBM-PC, Macintosh) quanto de compiladores (Microsoft C, Borland C).

Diversas camadas de serviços oferecidas pelo XINU são desnecessárias ao simulador e foram retiradas do código, gerando um código final mais enxuto.

As principais partes do XINU utilizadas pelo simulador são brevemente descritas a seguir:

- *Gerente de Processos*

Processos XINU são semelhantes às funções das linguagens de programação convencionais com exceção de possuírem fluxos de execução que executam de forma concorrente entre si.

O gerente de processos do XINU oferece serviços para criar (*create*), remover (*kill*), suspender (*suspend*) e reativar (*resume*) processos. Ele foi alterado para gerar os processos que vão simular todos os nós do Nó //. Dessa forma, a simulação de cada nó é feita através da criação de um processo (*create*) no XINU, conforme pode ser observado na figura 4-2.

```
// cria os nós do Nó //  
for( no = 0 ; no < TOTAL_NOS ; no++ )  
    create(CodigoNo, TAM_PILHA, INITPRIO, "NO", 0);
```

Figura 4-2. Trecho de código que cria processos XINU.

Os parâmetros passados a função *create* são descritos no Apêndice A.

O escalonamento preemptivo de processos oferecido pelo XINU é utilizado para simular o paralelismo real de processamentos existentes entre nós do Nó //.

- *Gerente de Memória*

Cada nó da máquina possui sua memória privativa. Para simular a memória de cada nó foram alterados os serviços *getmem* e *freemem* que o XINU oferece. Os processos XINU requisitam memória através da chamada *getmem* para simular as memórias privativas dos nós.

A figura 4-3 mostra um trecho do código que simula um nó, e nele pode ser visto a alocação de memória para simular a memória do nó.

```
// Código do nó
int CodigoNo(void)
{
    ...
    // aloca a memória para simular a memória do nó
    mem = getmem(TAM_NO);
}
```

Figura 4-3. Trecho de código que simula um nó.

- *Comunicação entre Processos*

O XINU possui duas funções *send* e *receive* que servem para comunicação entre seus processos. A alteração dessas funções permitiu simular o envio e o recebimento de *bytes* pelos nós através do barramento de serviços e do comutador de conexões.

A figura 4-4 esquematiza a utilização do XINU como base para a confecção do simulador e mostra o relacionamento entre os diferentes componentes do projeto Nó //.

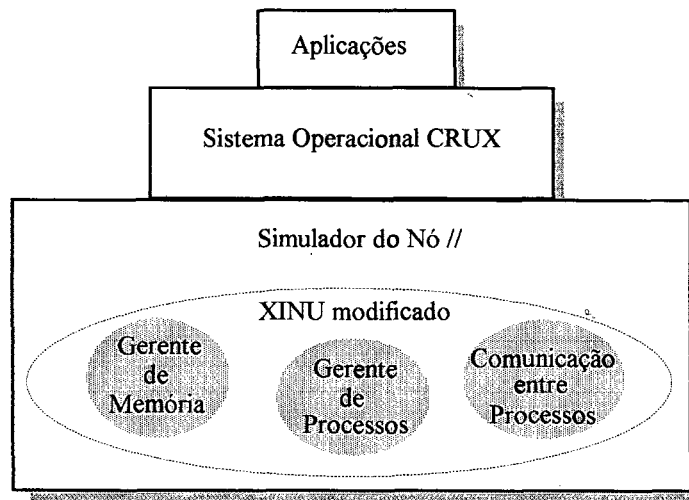


Figura 4-4. O XINU utilizado como base para o simulador do Nó //.

É importante ressaltar que embora a confecção do simulador se constituísse num trabalho relevante, durante o seu desenvolvimento sempre se manteve em consideração o seu objetivo principal que é o de oferecer o mais rapidamente possível um ambiente completo de programação equivalente ao do Nó //.

### 4.3.1. Alterações no XINU

Embora se mostrasse adequado para servir de base para o simulador, o XINU necessitou de diversas alterações, principalmente na forma de execução das chamadas de sistema e na gerência de memória. Elas são descritas a seguir.

- *Chamadas de sistema Implementadas através de Traps*

Na implementação original do XINU, os programas de usuário interagem diretamente com as camadas mais internas do sistema operacional. Os processos XINU após compilados são ligados ao próprio código do sistema XINU. Por isso, os processos conhecem o endereço das rotinas que implementam as chamadas de sistema. Toda chamada de sistema é efetuada como qualquer rotina do XINU, através de um *call*.

Essas características inviabilizariam a carga dinâmica de processos, necessária ao simulador. Para servir de base para o simulador foi necessário então implementar um outro esquema de chamadas de sistema, através de interrupções de *software*.

Nesse esquema, é gerada uma biblioteca de chamadas de sistema. Essa biblioteca é ligada junto com as aplicações que desejam fazer as chamadas. Ao ser feita a chamada, a rotina respectiva da biblioteca armazena os valores necessários em determinados registradores da máquina e gera uma interrupção pré-estabelecida.

O XINU captura essa interrupção, sua rotina de tratamento recebe o controle, verifica através dos valores dos registradores qual o serviço requisitado e quais os parâmetros necessários, executa a função e retorna valores nos registradores.

A rotina da biblioteca, por sua vez, interpreta os valores retornados pelos registradores, e retorna os valores da forma padrão da chamada de sistema.

- *Alteração no Gerente de Memória*

O gerente de memória original do XINU libera apenas 64 *Kbytes* para serem alocados entre todos os processos, o que é insuficiente para o simulador trabalhar com o número de nós que se pretende simular. Essa limitação sem razão aparente se deve, provavelmente, a motivos históricos. Portanto, um esquema alternativo necessita ser implementado para contornar esse problema. Duas alterações são propostas a seguir:

A primeira é alterar o gerente de memória para ele utilizar toda a memória abaixo dos 640 *Kbytes* disponíveis existente. Essa memória é denominada memória real, em contraste a memória estendida que reside acima do primeiro Megabyte. Essa alteração iria liberar uma quantidade de memória ainda insuficiente para simular todos os nós pretendidos.

A segunda alteração, que pode ser concebida como uma segunda etapa da primeira, consiste em fazer permutas (*swapping*) de processos que não estão em execução. A maneira mais simples seria fazer permutas em disco, porém a velocidade da simulação seria muito prejudicada. Pretende-se, por isso, utilizar a memória acima do primeiro *megabyte*, através de um gerenciador de memória expandida EMM386.SYS, existente no MS-DOS [DUN86 - pág. 185]. Esse gerenciador irá criar bancos de 64 *Kbytes* que serão utilizados para se efetuar permutas em memória. Esse esquema é extremamente veloz, principalmente pelo fato do gerenciador não efetuar efetivamente cópias de memória, e sim, utilizar de forma totalmente transparente, a memória virtual e mapeamentos de memória.

## 5. Sistema Operacional do Multicomputador Nó //

Neste capítulo propõe-se um sistema operacional para o multicomputador Nó //. Esse sistema, denominado CRUX, é constituído de micronúcleo distribuído, de uma biblioteca de funções de acesso às chamadas de sistemas e de um servidor.

### 5.1. Descrição

O sistema operacional para o Nó // está sendo desenvolvido a partir de uma proposta inicial descrita em [COR93]. Sua interface de programação é compatível com a do UNIX, o que permitirá importar mais facilmente aplicações de outros sistemas UNIX existentes. Pretende-se, seguindo a tendência mundial no sentido da padronização, implementar o padrão POSIX.1 (IEEE Portable Operating System Interface for Computing Environments) [LEW91].

A simplicidade de conceitos utilizados na arquitetura do Nó // levou a um esquema de implementação de um sistema operacional despojado. No sistema proposto, não há técnicas sofisticadas de gerenciamento de memória, tais como paginação ou segmentação. Não há, também, escalonamento de processos. Assume-se que cada nó possua apenas um processo e que este caiba integralmente em sua memória privativa.

Essas pré-concepções são perfeitamente cabíveis no caso de muitas arquiteturas paralelas já que o preço dos processadores e memórias vêm caindo vertiginosamente nos últimos anos, tornando ultrapassadas as técnicas descritas acima que foram criadas para gerenciar recursos escassos na época.

Existem antecedentes com relação à gerência da memória. O sistema operacional distribuído *Amoeba* [TAN92 - pág. 602], por exemplo, também considera a memória um recurso abundante. Para ser executado nesse sistema, um processo necessita caber integralmente na memória.

Com relação à gerência de processos, a ausência de escalonamento implica em grande simplicidade de código. Não havendo mais que um processo por nó, não existe a necessidade de se executar balanceamento de carga entre processadores.

Outra argumentação válida para essas simplificações é que o conceito de *thread* poderá ser futuramente implementado no sistema operacional proposto. *Threads* são adequados para implementação de servidores, por exemplo, onde o serviço oferecido pode ser subdividido em diversos fluxos seqüenciais.

*Threads* também são excelentes para serem utilizados na manipulação de eventos assíncronos, como os sinais UNIX [BRI91]. Neste caso, *threads* são criados com a incubência única de recebê-los e tratá-los.

## 5.2. Processos

Processos CRUX são programas em execução nos nós do Nó //. Um processo é constituído pelo seu espaço de endereçamento, dados, pilha, valores dos registradores e outras informações necessárias para sua execução.

A figura 5-1 apresenta um processo colocado em um nó qualquer do Nó //.



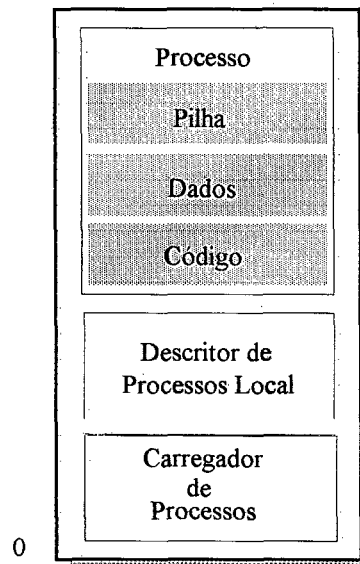


Figura 5-1. Um processo colocado na memória de um nó.

Conforme pode ser observado, existem três componentes principais descritos a seguir:

- *Carregador de Processos*

Processos CRUX são criados através da chamada *fork*, que é uma chamada de sistema compatível com UNIX. Neste caso, o processo que é criado é enviado para um nó disponível que o recebe e o "instala". O código responsável por receber e instalar o processo e seu descritor na memória é o carregador de processos.

- *Descritores de Processos Local*

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

- *Processo*

Um processo colocado na memória é composto de seu código, sua área de dados e pilha. A área de código consiste em uma seqüência de padrões de *bytes* que o processador interpreta como instruções de máquina. A área de dados corresponde à seção de dados inicializados e não inicializados existentes no arquivo executável. A área de pilha é criada e pode ser alterada dinamicamente durante a execução do programa [BAC86 - pág. 25].

Na implementação inicial do CRUX utilizou-se um formato de programa que não necessita de relocação de seus endereços durante sua carga. Seu formato é semelhante aos dos programas .COM existentes nos sistemas MS-DOS [DUN86 - pág. 23]. Essa característica permitiu uma grande simplificação nas tarefas dos carregadores de processos.

### 5.3. Micronúcleo CRUX

O sistema operacional do Nó // é fundamentado sobre um micronúcleo genérico que executa em cada nó processador da máquina. Esse micronúcleo oferece um reduzido número de serviços básicos – acessíveis através de uma biblioteca de funções compatíveis com o UNIX – os quais são utilizados pelos processos.

Na implementação sobre o simulador, o micronúcleo não é implementado como entidade separada dos processos (esta separação será realizada posteriormente). Rotinas de biblioteca do micronúcleo são ligadas aos códigos dos processos. A figura 5-2 representa os processos colocados nos nós com seus respectivos micronúcleos já anexados em seu código.

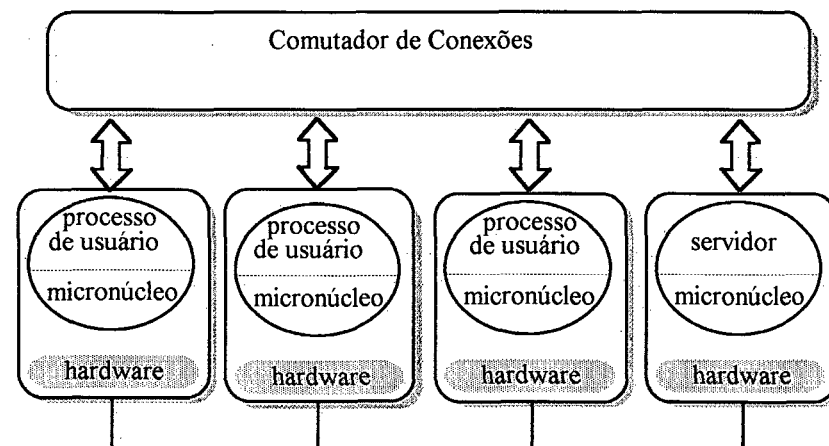


Figura 5-2. O micronúcleo embutido no código do processo.

Apesar de todo o trabalho aqui apresentado ser direcionado para esse sistema operacional compatível com o UNIX, é importante observar que os serviços oferecidos pelo micronúcleo proposto são genéricos o suficiente para se implementar outros sistemas compostos de outros processos servidores.

Na figura 5-3. são apresentadas as camadas que compõe o micronúcleo do CRUX.

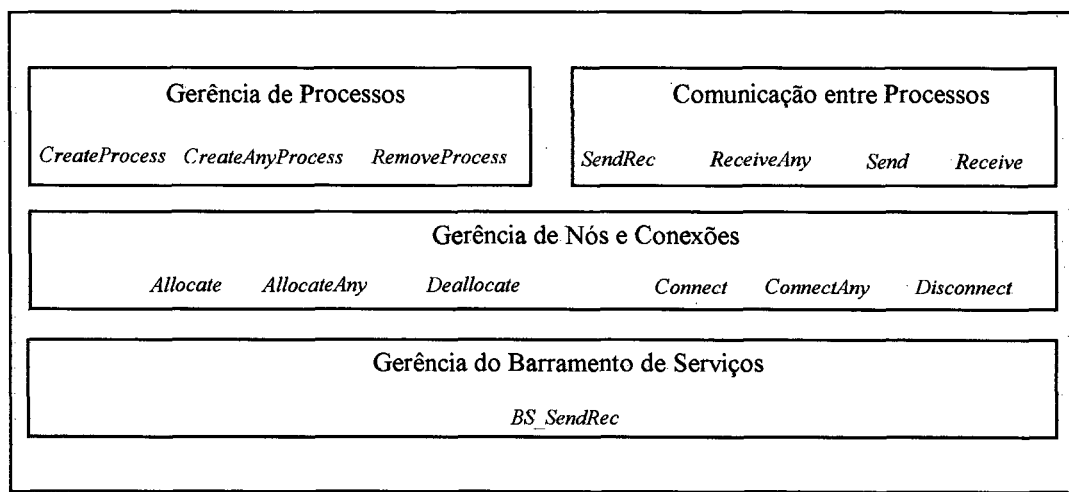


Figura 5-3. As camadas do micronúcleo.

### 5.3.1. Gerência do Barramento de Serviços

A camada de gerência do barramento de serviços é a responsável pelo controle do envio e recebimento de mensagens entre um nó qualquer e o CNC, pelo barramento de serviços. Sua interface de programação consiste de uma única função, descrita a seguir, que é responsável por enviar uma mensagem e aguardar sua resposta.

Tabela 5-1. Função da camada gerência do barramento de serviços

Função	Descrição
<i>BS_SendRec(msg)</i>	Envia uma mensagem para o CNC e aguarda a resposta.

A implementação dessa função consiste na interação direta com o *hardware* do barramento de serviços para o envio e recebimento de mensagens através deste dispositivo.

O CNC é concebido como um servidor que controla a conexão e a alocação de nós. Assim, ele implementa internamente duas funções *BS\_ReceiveAny* e *BS\_Send* que servem, respectivamente, para receber requisições e enviar as respostas. Dentro deste contexto, a função *BS\_SendRec* é utilizada por processos clientes que requisitam nós e conexões.

### 5.3.2. Gerência de Nós e Conexões

A camada de gerência de nós e conexões é responsável por oferecer serviços de conexão e alocação de nós. Seus serviços são implementados utilizando a função *BS\_SendRec* da camada inferior e são mostradas a seguir.

Tabela 5-2. Funções da camada gerência de nós e conexões

<b>Função</b>	<b>Descrição</b>
<i>Allocate(nid)</i>	Aloca o nó <i>nid</i> .
<i>AllocateAny()</i>	Aloca um nó livre qualquer.
<i>Deallocate()</i>	Desaloca o nó corrente.
<i>Connect(nid)</i>	Pede conexão com o nó <i>nid</i> .
<i>ConnectAny()</i>	Pede conexão com um nó qualquer.
<i>Disconnect()</i>	Pede desconexão com o nó conectado.

A implementação destas funções consiste num mapeamento direto da função e seus parâmetros em uma mensagem a ser enviada pelo barramento de serviços ao CNC. As mensagens correspondentes foram apresentadas anteriormente em 3.4.

A sintaxe das funções apresentadas considera a existência de apenas um canal de comunicação em cada nó. Por esse motivo, torna-se desnecessária a passagem da identificação do nó a ser desconectado como parâmetro da função *Disconnect*, pois o micronúcleo conhecendo o nó ao qual o seu está conectado monta a mensagem ao CNC enviando junto o identificador do nó.

A função *Deallocate* é a última função chamada pelo processo ao terminar. Depois de sua execução, o nó onde está colocado o processo que a chamou é liberado para ser alocado para outros processos.

### 5.3.3. Gerência de Processos

A camada de gerência de processos e a de comunicação entre processos, oferecem serviços aos processos externos ao micronúcleo. Por esse motivo, o conjunto dos serviços oferecidos constitui a interface de programação do micronúcleo.

Na camada de gerência de processos, existem três funções responsáveis por criar e remover processos.

Tabela 5-3. Funções da camada de gerência de processos.

Função	Descrição
<i>CreateProcess(nid)</i>	Cria um processo no nó <i>nid</i> .
<i>CreateAnyProcess()</i>	Cria um processo num nó qualquer.
<i>RemoveProcess()</i>	Remove o processo do nó corrente.

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

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

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

#### **5.3.4. Comunicação entre Processos**

Das quatro funções existentes na camada de comunicação entre processos, mostradas na tabela 5-4, três são orientadas à disciplina de comunicação cliente-servidor (*SendRec*, *ReceiveAny* e *Send*).

A utilização da função *Send* em conjunção à *Receive* permite a utilização da disciplina produtor-consumidor na qual é necessário apenas um sentido de comunicação onde um processo se responsabiliza por enviar mensagens e outro por recebê-las.

As interações entre os processos existentes no sistema seguirão na grande maioria das vezes a disciplina de comunicação cliente-servidor. Entretanto, ocorrerão desvios desta disciplina nas chamadas de sistema referentes à criação dinâmica de processos, conforme será visto na seção 5.4.3.

Tabela 5-4. Funções da camada de comunicação entre processos.

Função	Descrição
<i>SendRec(nid)</i>	Envia uma mensagem para o nó <i>nid</i> e aguarda uma resposta.
<i>ReceiveAny()</i>	Aguarda uma mensagem de um nó qualquer.
<i>Send(nid)</i>	Envia uma mensagem para o nó <i>nid</i> .
<i>Receive(nid)</i>	Aguarda uma mensagem do nó <i>nid</i> .

Na disciplina cliente-servidor, um processo servidor executa *ReceiveAny* para receber uma requisição de algum processo cliente. Este último, por sua vez, executa *SendRec*, enviando uma requisição de serviço ao servidor e aguardando a resposta. O servidor, após receber e atender a requisição de serviço, executa *Send* enviando a resposta (figura 5-4).

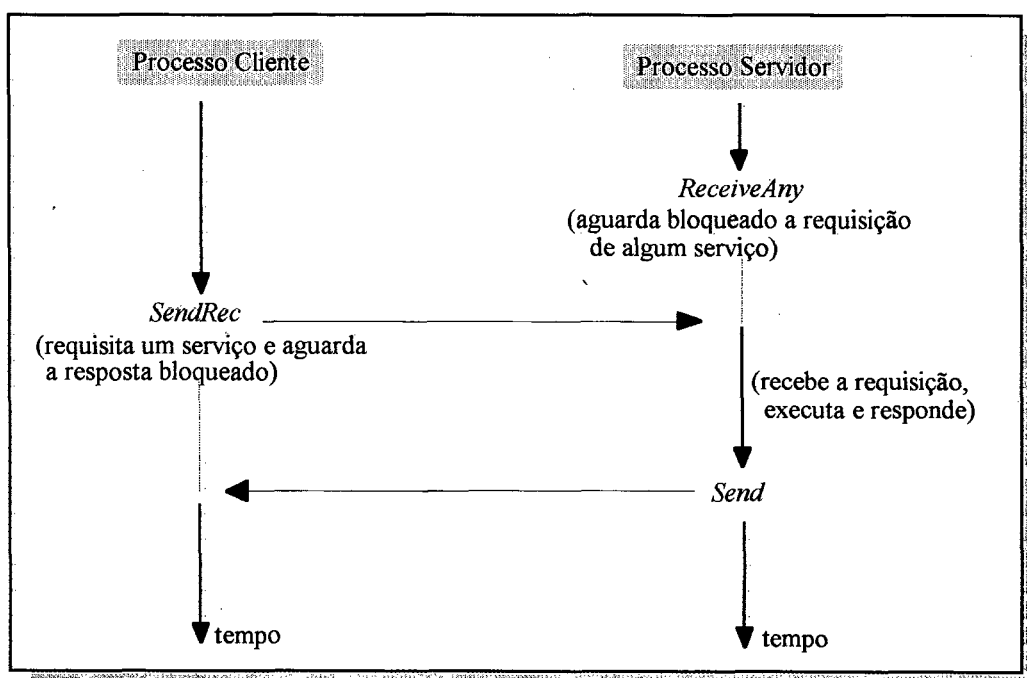


Figura 5-4. Comunicação cliente-servidor utilizando serviços do micronúcleo.



A implementação das funções das camadas de comunicação entre processos se utiliza dos serviços oferecidos pela camada de gerência de nós e conexões. Na figura 5-5 é representada a comunicação entre um processo cliente e um processo servidor, colocados em nós distintos.

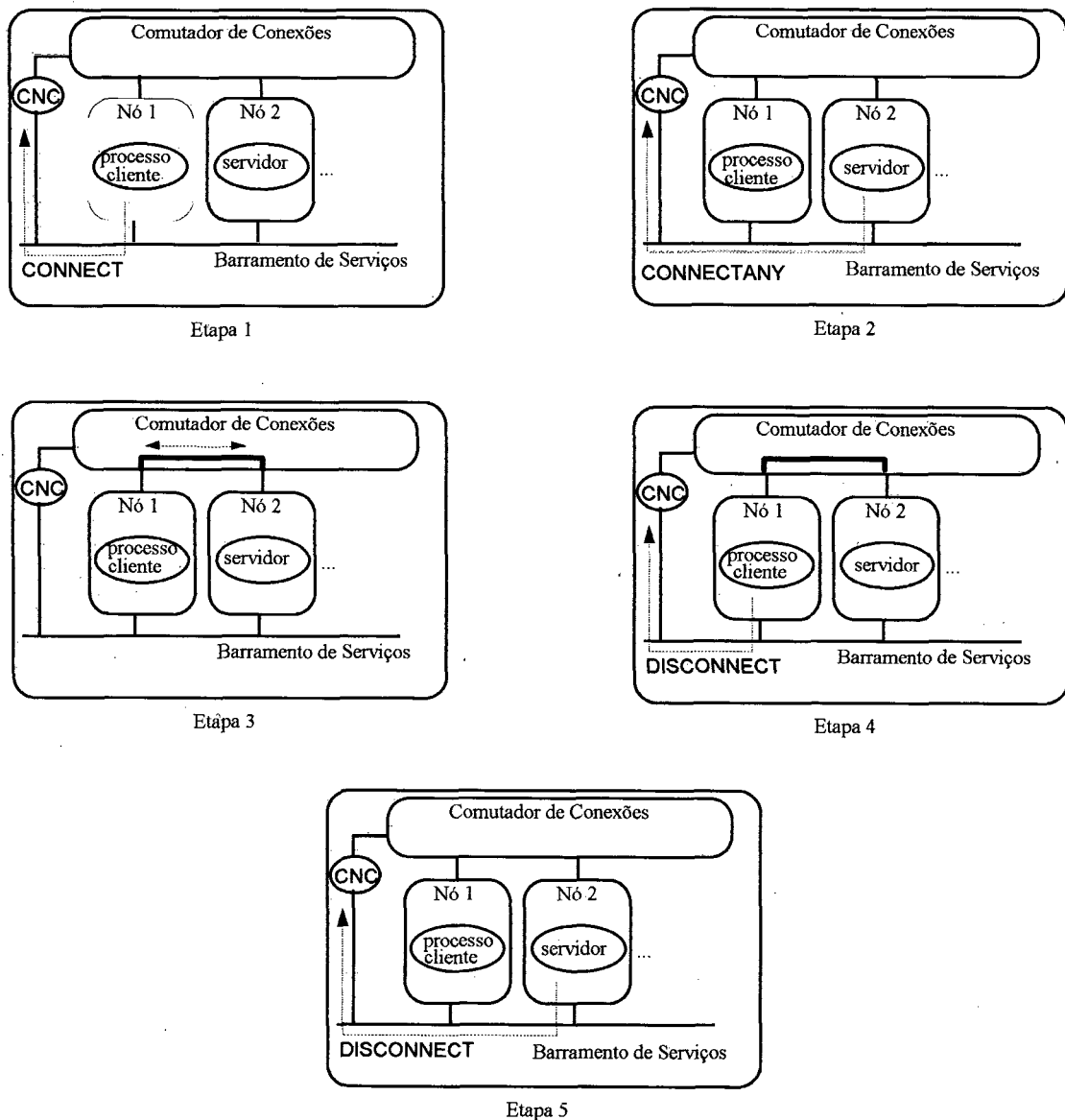


Figura 5-5. Comunicação entre um processo cliente e um servidor.

O processo cliente deseja enviar uma requisição para o processo servidor e, com esse objetivo, executa *SendRec*. Esta função chama *Connect* pedindo uma conexão com o nó onde está colocado o processo servidor. Essa última função gera uma mensagem CONNECT que é enviada para o CNC.

O servidor, deseja receber requisições e executa a função *ReceiveAny* que chama *ConnectAny* que, por sua vez, gera a mensagem CONNECT\_ANY ao CNC. Devido ao fato de ambas as partes concordarem em se conectar, a conexão é efetuada.

Na etapa 3 é representada a comunicação fluindo entre os processos cliente e servidor. A transferência de informações de um nó para outro é realizada de forma síncrona. Desta forma, o processo cliente permanece bloqueado até o servidor responder sua requisição através de *Send*.

Nas etapas 4 e 5, após a comunicação ser efetuada, *SendRec* e *Send* chamam *Disconnect* que enviam mensagens DISCONNECT para o CNC, e este desfaz a conexão.

## 5.4. Biblioteca CRUX

A fim de oferecer uma interface de programação compatível com o UNIX, existe uma biblioteca de funções que podem ser ligadas com os programas de usuário. Através dessa biblioteca, os processos acessam o conjunto de chamadas do sistema CRUX (figura 5-6).

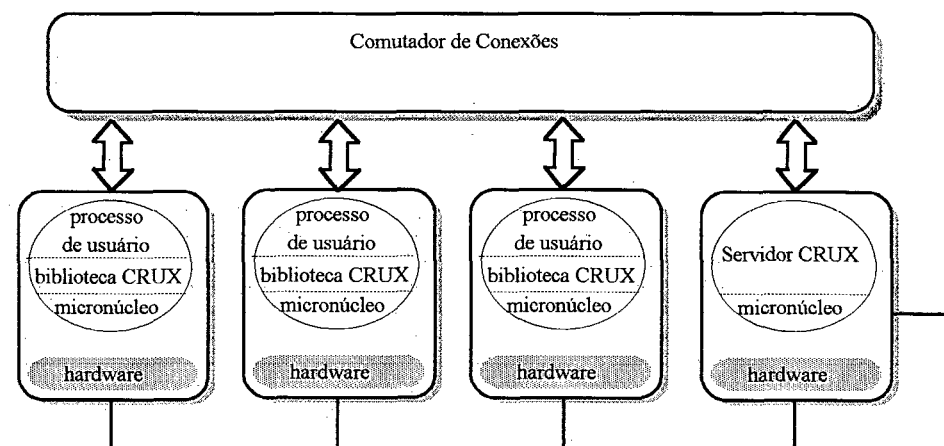


Figura 5-6. A biblioteca CRUX.

Do ponto de vista da implementação, essas chamadas podem ser subdivididas em três tipos:

- chamadas resolvidas localmente;
- chamadas resolvidas pelo servidor CRUX;
- chamadas especiais.

#### 5.4.1. Chamadas Resolvidas Localmente

Existem várias chamadas de sistema que não exigem interação externa, isto é, são resolvidas localmente no próprio nó, como, por exemplo, *getpid*.

Cada processo em execução no CRUX possui um valor numérico único, denominado *pid*, que serve para identificá-lo. Este identificador é formado na criação do processo, pela combinação do número do nó onde o processo está colocado, *nid*, que é retornado pelo CNC, e o valor *id* retornado pelo servidor CRUX.

A maioria das chamadas de sistema são implementadas interagindo com o servidor CRUX e, desta forma, precisam recuperar o *pid* para passá-lo ao servidor. Portanto, para melhorar o desempenho, o descritor de processos local, presente junto com o processo na memória de cada nó, possui uma entrada para armazenar o *pid* do processo. Graças a essa característica, a implementação de *getpid* é trivial, e consiste em um acesso ao descritor de processos local.

A chamada UNIX *brk* [TAN92 - pág. 294]<sup>3</sup> também é resolvida localmente. Ela é implementada no código contido na própria biblioteca CRUX e não se utiliza de serviços do micronúcleo. Seu objetivo é devolver ao processo a localização do topo do segmento de dados.

#### 5.4.2. Chamadas Resolvidas pelo Servidor CRUX

A grande maioria das chamadas de sistema são resolvidas inteiramente no servidor CRUX. A implementação dessas chamadas consiste numa comunicação cliente-servidor, utilizando os serviços oferecidos pela camada de comunicação entre processos existentes no micronúcleo.

Com o objetivo de demonstrar a implementação dessas chamadas, na figura 5-7 é descrito o algoritmo da chamada *close*. Ela é uma função simples que fecha o arquivo cujo o descritor é passado como parâmetro. O protótipo desta chamada é apresentada a seguir [LEW91 - pág. 242] segundo o padrão POSIX.1:

```
int close ( int desarg );
```

---

<sup>3</sup>Apesar de *brk* ser uma chamada já estabelecida em sistemas UNIX, ela não está especificada em POSIX.1, pois este tópico é considerado muito dependente de máquina para conseguir-se uma padronização.

onde o argumento *desarq* consiste no descritor do arquivo a ser fechado.

Recupera o identificador de processo <i>pid</i> existente no descritor de processos local
Monta uma mensagem composta pelo tipo da chamada ( <i>close</i> ), o <i>pid</i> e <i>desarq</i>
Envia a mensagem utilizando <i>SendRec</i> ao servidor CRUX e aguarda a resposta
Retorna da função com o valor da resposta do servidor CRUX

Figura 5-7. Algoritmo da chamada de sistema *close*.

### 5.4.3. Chamadas Especiais

Algumas chamadas de sistema possuem comportamentos que as diferenciam das outras. Essas chamadas são relacionadas com a gerência de processos, e suas implementações são vistas a seguir:

#### 5.4.3.1. *Fork*

Processos são criados através da chamada *fork*. O novo processo criado é denominado processo filho e é uma cópia do seu processo criador, o processo pai. Entretanto, o processo filho possui um identificador de processo *pid* distinto.

Para acessar essa chamada de sistema, POSIX.1 define uma função cujo protótipo é [LEW91 - pág. 285]:

```
pid_t fork();
```

A chamada *fork* retorna 0 para o filho e, o *pid* do processo filho para o pai. Um programa que executa *fork* tipicamente testa o valor de retorno e executa tarefas diferentes para o processo filho e processo pai, conforme mostra a figura 5-8.

```
if ( fork() == 0 ) {  
    // processo filho executa a partir deste ponto  
}  
else {  
    // processo pai executa a partir deste ponto  
}
```

Figura 5-8. Um programa típico utilizando *fork*.

A implementação da chamada *fork* no CRUX é uma tarefa composta de várias etapas, e envolve interação com o servidor CRUX e com o CNC. A seguir, são esquematizadas as etapas que ocorrem durante a execução do *fork*, através da figura 5-9 e de uma breve descrição de cada etapa.

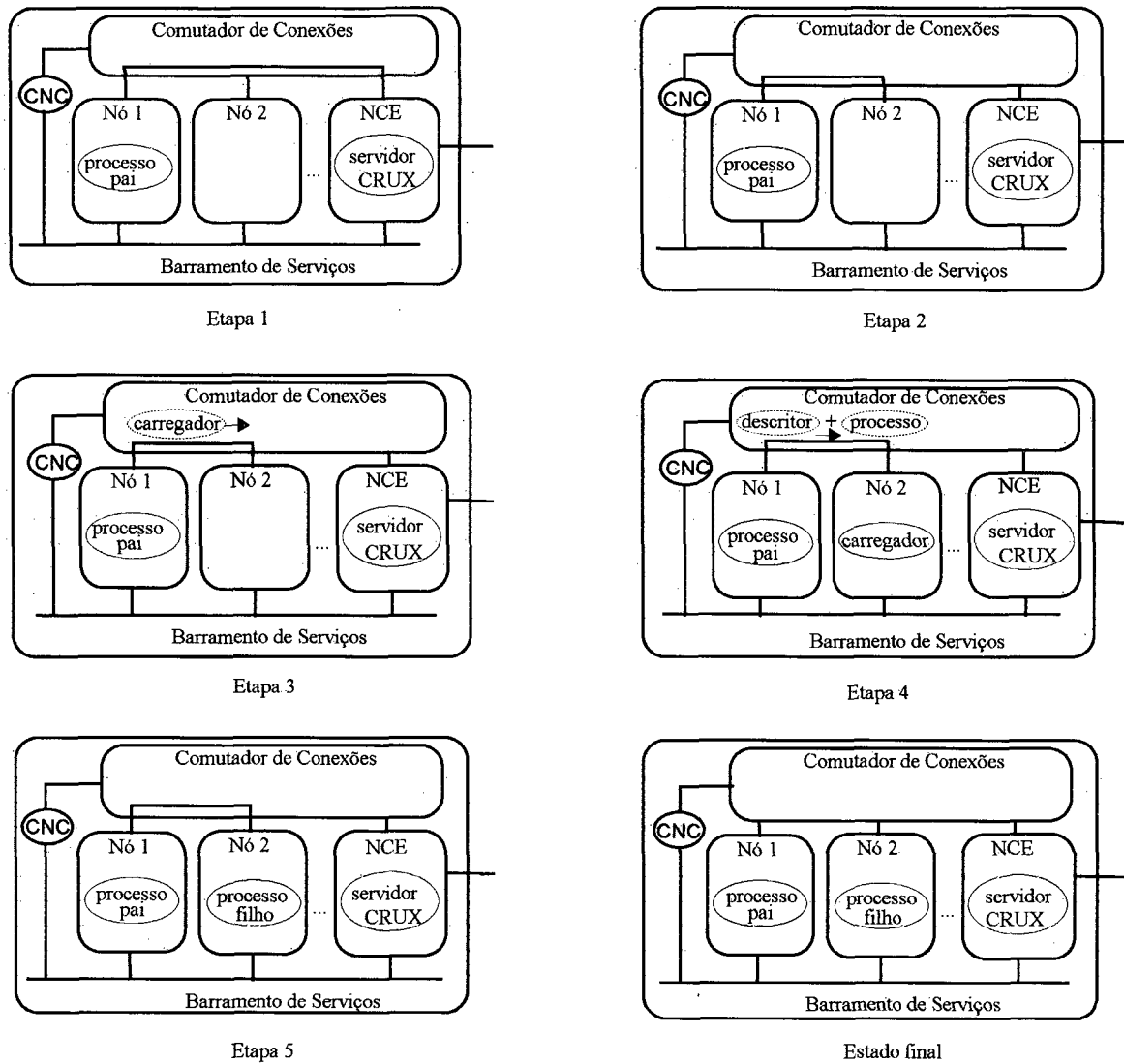


Figura 5-9. Etapas da chamada de sistema *fork*.

Na etapa 1 *fork* executa uma chamada *SendRec* ao servidor CRUX. Esta chamada, que é um pedido para ser criado um processo novo, provoca uma conexão entre o nó do processo pai e o nó do servidor CRUX. Este usa uma entrada livre para o novo processo em sua tabela de processos e retorna através de *Send* um valor *id* que é único para cada processo.

Na etapa 2 a função *fork* executa uma chamada *CreateProcess* ao micronúcleo. Esta, por sua vez, chama *Allocate* para enviar uma mensagem pelo barramento de serviços, através de *BS\_SendRec*, ao CNC que aloca um nó disponível, estabelece uma conexão entre os dois nós, e retorna o identificador de nós *nid*.

Na etapa 3 a função *CreateProcess*, utilizando *Send*, envia um carregador para o nó alocado que é recebido e instalado pelo código gravado em ROM. O código do carregador é compacto e é ligado junto com o código de *fork* durante o processo de compilação e geração da biblioteca de chamadas de sistema.

Na etapa 4, o carregador já em execução recebe o descritor e o código do processo e os instala no nó alocado.

Na etapa 5, ainda na função *CreateProcess*, os dois processos, pai e filho, pedem desconexão, através de uma chamada *Disconnect*. *CreateProcess* termina, retornando *nid*, o identificador do nó onde foi instalado o processo novo.

A função *fork* utiliza o identificador *id*, retornado pelo servidor CRUX, e o identificador de nós *nid*, retornado por *CreateProcess*, e os mescla formando o identificador *pid*. Este identificador do novo processo criado é retornado por *fork* ao processo pai e pode ser recuperado pelo processo filho através da chamada de sistema *getpid*.

É interessante observar que o micronúcleo desconhece *pid*. Suas funções trabalham exclusivamente sobre identificador de nós *nid*. Essa característica se deve ao fato do identificador de processos *pid*, se referir exclusivamente a processos UNIX, e o micronúcleo ser genérico, conforme já foi ressaltado anteriormente.



### 5.4.3.2. Exec

A chamada de sistema *exec* carrega o conteúdo de um arquivo executável (isto é, um outro programa) sobre a memória do processo que a chama.

Para acessar a chamada de sistema *exec*, o POSIX.1 define um conjunto de funções denominado *família exec*. A função *execve* é a mais genérica e seu protótipo é apresentado a seguir [LEW91 - pág. 262]:

```
int execve(char *caminho, char **argv, char **envp);
```

O parâmetro *caminho* contém o nome completo do arquivo a ser carregado. O parâmetro *argv* é um ponteiro para um vetor de argumentos passados para o novo programa. O parâmetro *envp* é um ponteiro para um vetor de *strings* de variáveis de ambiente.

As etapas necessárias para a execução de *exec* são apresentadas na figura 5-10.

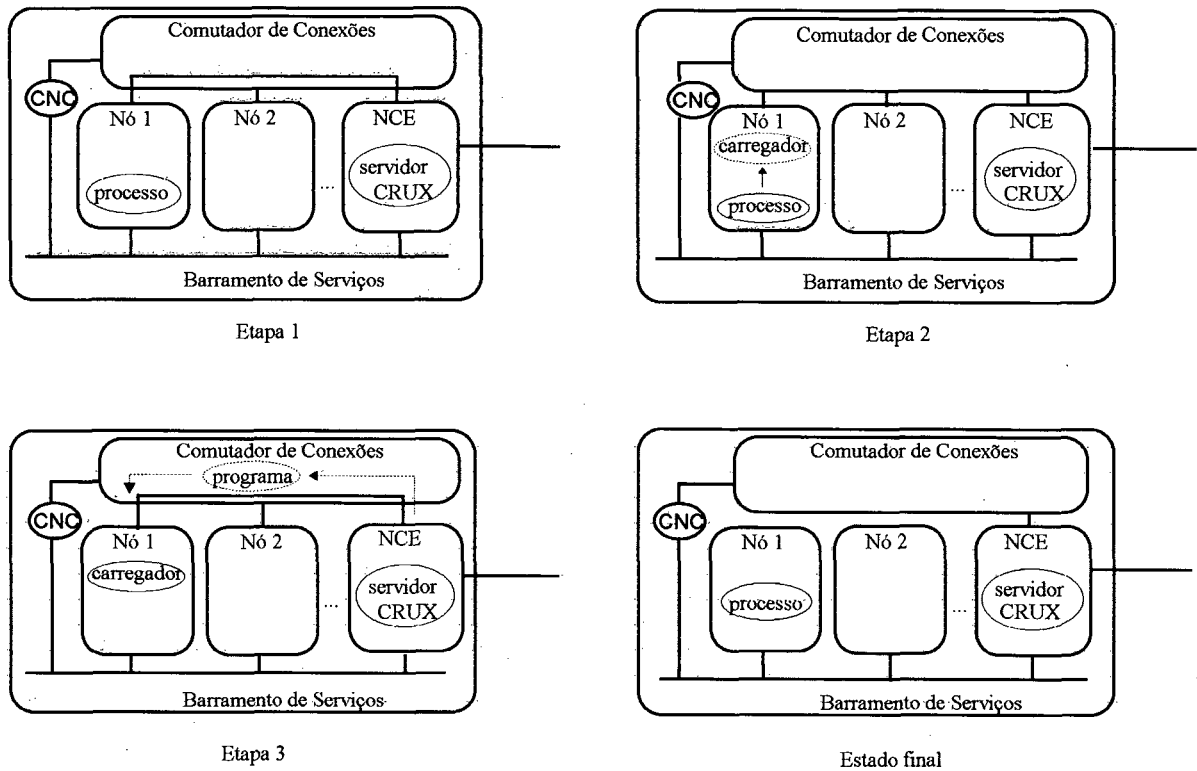


Figura 5-10. Etapas da chamada de sistema *exec*.

Na etapa 1, *exec* faz uma chamada *SendRec* enviando ao servidor CRUX o nome do arquivo. O servidor CRUX devolve, através de *Send*, o tamanho do arquivo e se prepara para enviá-lo.

Na etapa 2, *exec* copia seu carregador para o início da memória, e, em seguida, passa a executá-lo.

De forma semelhante à função *fork*, *exec* possui um código de carregador ligado junto ao seu código durante o processo de compilação.

É valioso observar que a área de dados do processo será encoberta pelo programa que será carregado. Alguns dados necessitam ser passados ao carregador, como, por exemplo, o tamanho do programa a ser recebido. Outros necessitam ser passados ao novo programa quando este começar a executar, como, por exemplo, as variáveis de ambiente.

O descritor de processos local, que permanece intacto durante todas as etapas de *exec*, é usado para armazenar os dados que necessitam ser passados para o carregador e para o novo programa.

Na etapa 3, o carregador recebe o programa, e o instala em seu nó, passando, em seguida, a executar o seu código.

#### **5.4.3.3. Exit**

A chamada *exit* causa o término normal de um processo. Seu protótipo é apresentado a seguir [LEW91 - pág. 266]:

```
void exit(int status);
```

onde *status* é o valor de término do programa passado para o processo pai.

Ela se constitui numa chamada especial por envolver, além da interação do processo cliente com o servidor CRUX, a interação com processo pai. No UNIX, um processo ao terminar sua execução, passa um código de retorno ao processo pai. Essa passagem se dá através da combinação da chamada *exit* executada pelo processo filho com a chamada *wait*, executada pelo processo pai.

Durante sua execução, diversas situações devem ser consideradas: o processo pai executou *wait* antes do filho executar *exit*, ou se o processo filho executou *exit* antes do pai executar *wait*, ou se o processo pai terminou sem executar *wait*.

A figura 5-11. mostra o algoritmo da implementação de *exit*.

<p>Recupera o identificador de processos <i>pid</i> existente no descritor de processos local</p> <p>Monta uma mensagem composta pelo tipo da chamada (<i>exit</i>) e o <i>pid</i></p> <p>Envia a mensagem utilizando <i>SendRec</i> para o servidor e aguarda a resposta</p> <p>Se resposta é: pai ainda não executou <i>wait</i></p> <p style="padding-left: 40px;">Aguarda, bloqueado, utilizando <i>Receive</i>, mensagem do pai avisando que ele executou <i>wait</i></p> <p style="padding-left: 40px;">Envia, utilizando <i>Send</i>, mensagem para o pai com o código de término</p> <p>Se resposta é: pai já executou <i>wait</i></p> <p style="padding-left: 40px;">Desbloqueia o pai enviando-lhe mensagem, através de <i>Send</i>, com o seu código de término</p> <p>Se resposta é: pai terminou sem esperar</p> <p style="padding-left: 40px;">Nada faz</p> <p>Chama a função do micronúcleo <i>RemoveProcess</i> para remover seu contexto.</p>
--

Figura 5-11. Algoritmo de *exit*.

#### 5.4.3.4. *Wait*

A função *wait* permite que um processo aguarde, bloqueado, o término de um de seus filhos. De forma semelhante a *exit*, essa chamada envolve interação com outros processos além do servidor CRUX.

Seu protótipo segundo o POSIX.1 é [LEW91 - pág. 507]:

```
pid_t wait (int * statloc);
```

onde *statloc* é um ponteiro para um inteiro onde o código de término do processo filho será armazenado. Essa função retorna o *pid* do processo filho que terminou.

A figura 5-12 apresenta o algoritmo de *wait*.

<p>Recupera o identificador de processo <i>pid</i> existente no descritor de processos local</p> <p>Monta uma mensagem composta pelo tipo da chamada (<i>wait</i>) e o <i>pid</i></p> <p>Envia a mensagem utilizando <i>SendRec</i> para o servidor CRUX e aguarda a resposta</p> <p>Se resposta é: nenhum filho terminou</p> <p style="padding-left: 40px;">Aguarda mensagem de algum filho (<i>ReceiveAny</i>) com seu código de término</p> <p>Se resposta é: filho já tinha terminado</p> <p style="padding-left: 40px;">Desbloqueia o filho enviando-lhe mensagem (<i>Send</i>)</p> <p style="padding-left: 40px;">Recebe (<i>Receive</i>) o código de término do filho</p> <p>Se resposta é: não possui filhos</p> <p style="padding-left: 40px;">Nada faz</p>
--

Figura 5-12. Algoritmo de *wait*.

## 5.5. Servidor CRUX

O servidor CRUX é um servidor único no sistema, responsável por gerenciar duas funções importantes:

- *Funções de processos UNIX*

Como foi visto anteriormente, as funções *fork*, *exit*, *wait* e *exec*, todas relacionadas a processos, são implementadas utilizando chamadas de procedimento remoto. Com o objetivo de atender essas requisições, o servidor CRUX mantém uma *tabela de processos*, na qual são armazenadas as informações dos processos existentes no sistema.

- *Funções de sistema de arquivos UNIX*

As chamadas padrão de sistema de arquivos como, por exemplo, *open*, *close*, *lseek*, *read* e *write*, são tratadas pelo servidor CRUX, que, para tal, se comporta como um servidor de arquivos. Cada processo pode manter diversos arquivos abertos, cujos descritores de arquivos [BAC86 - pág. 93] são armazenados na tabela de processos mantida pelo servidor CRUX.

### **5.5.1 Implementação do Servidor CRUX**

A implementação do servidor utilizou um expediente que simplificou muito os algoritmos utilizados. Considerando-se o enfoque utilizado de reservatório de processadores, toda a parte relacionada aos serviços de arquivos foi mantida fora do Nó //. Utilizou-se a interface de programação do UNIX (Linux) da estação de trabalho, para implementar-se as funções de sistemas de arquivos.

Um programa, denominado servidor de arquivos, foi desenvolvido para executar na estação de trabalho, sobre o sistema Linux, com a responsabilidade de receber requisições relacionadas a arquivos, executá-las e respondê-las.

Dessa forma, o servidor CRUX não implementa o servidor de arquivos e o controlador de disco (*device driver*) no Nó //. Toda chamada de procedimento remoto relacionada a arquivos, é passada para o servidor de arquivos na estação de trabalho (figura 5-13)

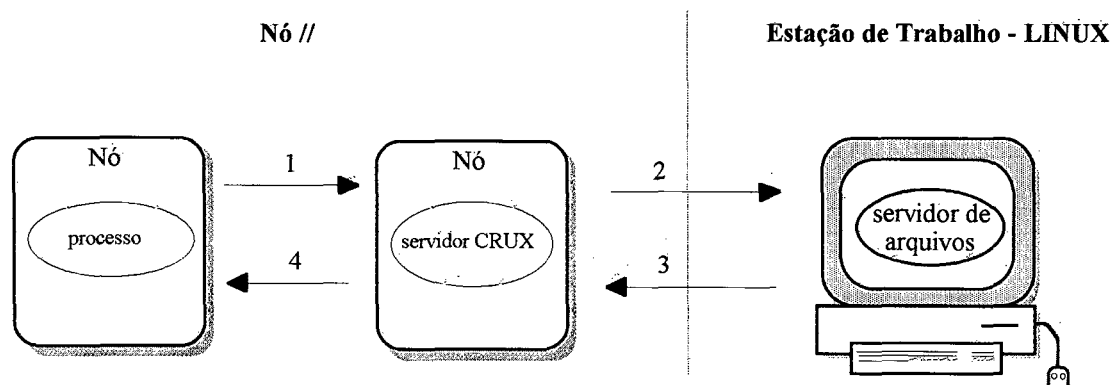


Figura 5-13. Utilização de um servidor de arquivos externo ao Nó //.

O processo requisita um serviço ao servidor CRUX fazendo uma chamada de procedimento remoto (1). O servidor constata que a chamada se refere a serviços de arquivo e faz uma chamada de procedimento remoto, passando os mesmos parâmetros que recebeu ao servidor de arquivos (2). O servidor de arquivos recebe a mensagem, executa a tarefa e responde ao servidor CRUX (3). Este último recebe a resposta e envia ao processo que requisitou o serviço (4).

As requisições ao servidor CRUX relacionadas com processos são tratadas no próprio servidor. Com esse objetivo ele gerencia uma tabela de processos, na qual cada processo possui uma entrada que armazena informações pertinentes.

Por exemplo, cada processo possui um processo pai e um estado associado, informações que são armazenadas na tabela de processos. Quando um processo executa *exit*, uma chamada de procedimento remota é feita ao servidor, este verifica se o pai do processo havia executado um *wait*. Caso afirmativo, o processo filho, que fez a chamada pode terminar, sua entrada na tabela de processos é liberada, e a informação que o processo pai, já executou *wait* é retornada ao filho.

Se o pai ainda não executou essa chamada, o estado do filho, armazenado na tabela, passa a ser, EXITING, que é o estado que informa que o processo está bloqueado esperando o pai executar *wait*. Em seguida a informação que o pai ainda não executou *wait* é retornada ao processo filho.

A tabela de processos, centralizada no servidor CRUX, e os descritores de processos locais, distribuídos em cada nó, são responsáveis por fornecer ao sistema CRUX todas as informações necessárias para ele gerenciar os processos e atender suas requisições de serviços.

## 5.6. Interpretador de Comandos

O interpretador de comandos do UNIX – o *shell* – constitui um processo comum de usuário. Por essa característica, ele pode ser substituído pelos usuários por outros interpretadores alternativos. Apesar dessa flexibilidade, existe uma tentativa para sua padronização (bem como dos comandos UNIX), através do documento POSIX.2 [LEW91].

As funções do *shell* podem ser resumidas, basicamente, em receber um comando e criar um processo filho para executá-lo. Sua implementação constitui num tópico interessante pelo fato de envolver diversas chamadas de sistema.

Na figura 5-14 pode ser vista a implementação bastante simplificada de um trecho do código do *shell*.



```
for(;;) {                                // ciclo "eterno"
    write(STDOUT, "\n$", 3);              // apresenta o prompt
    read(STDIN, comando, sizeof(comando)-1); // lê o comando
    if(fork() == 0)                       // cria um processo filho
        execve(comando, argv, envp);     // filho: dispara o comando
    else
        wait(&status);                   // pai: aguarda término do filho
```

Figura 5-14. Um trecho do programa *shell*.

Diversas simplificações são colocadas neste código. Por exemplo, não é testado se o comando lido é válido antes de executar o *fork*, nem é testado o retorno de erro das funções chamadas.

A figura 5-15 apresenta graficamente a seqüência de funcionamento da execução de *shell*.

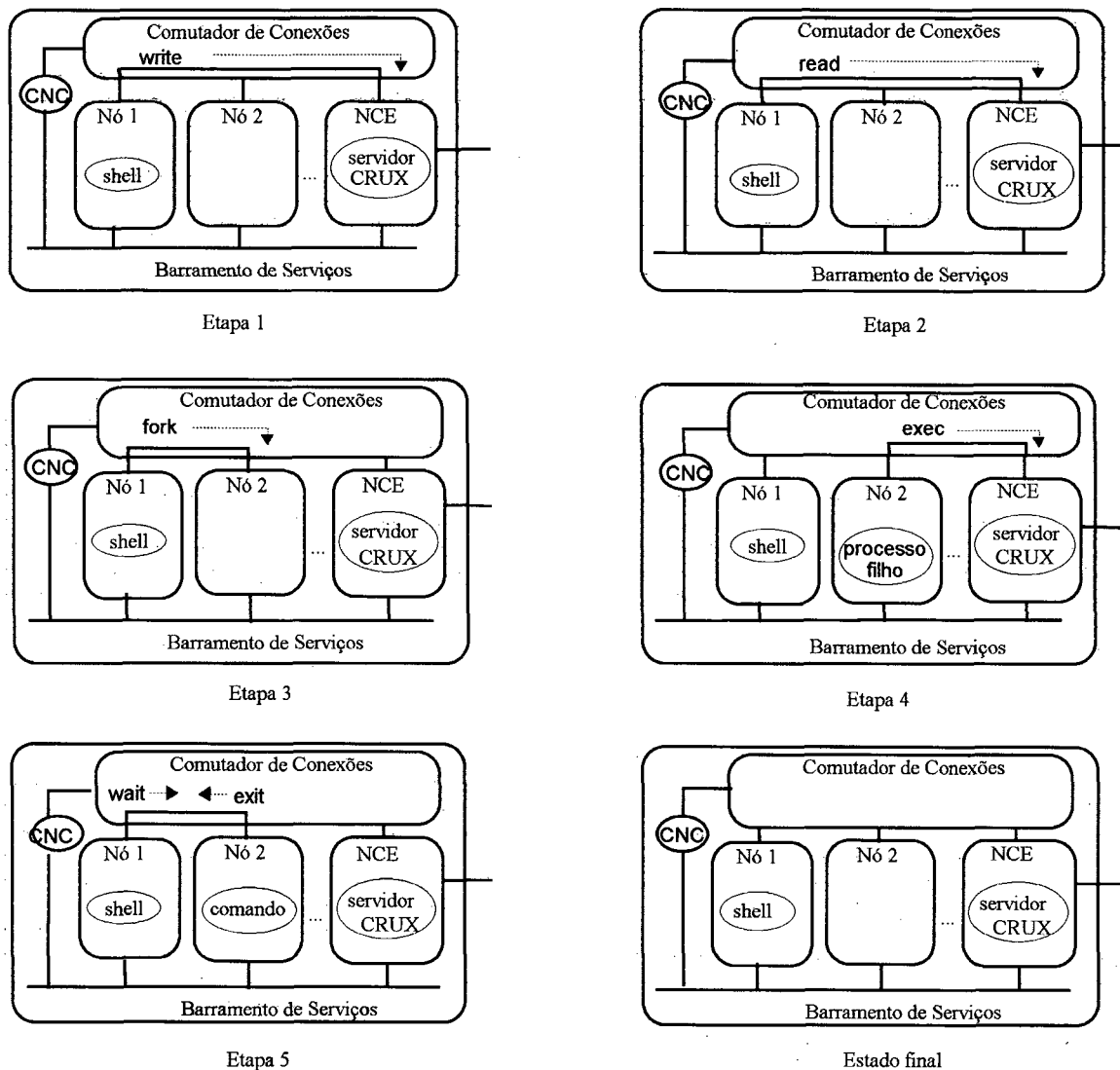


Figura 5-15. Execução do *shell*.

Nas etapas 1 e 2, o *shell* interage com o servidor CRUX escrevendo uma mensagem de *prompt* e lendo uma seqüência de caracteres contendo o comando.

Na etapa 3, o processo *shell* cria, através de *fork*, um processo filho para executar o comando. A seqüência de interações com o servidor CRUX e com o CNC oriundas da função *fork*, já foram apresentadas anteriormente em 5.4.3.1 e, por isso, não são detalhadas aqui.

Na etapa 4, o processo filho executa o comando carregando um programa para executá-lo através de *exec*.

Na etapa 5, após o comando ser executado, o processo filho termina através de *exit*. O shell que aguardava seu término através de *wait* recebe o valor de retorno.

A conexão entre os dois nós é desfeita, e o nó onde estava colocado o processo filho é liberado, conforme mostrado no estado final.

## 5.7. Carga Inicial do Sistema Operacional

A seguir são apresentadas as etapas necessárias para a carga inicial do sistema operacional proposto para o multicomputador Nó //. A implementação dessas etapas, como um todo, se constituiu em um trabalho significativo dentro do desenvolvimento do sistema.

A carga inicial do sistema, comumente denominada de *bootstrap*, é uma tarefa composta de várias etapas. Optou-se no Nó // por manter o microcódigo em ROM o mais despojado possível e, dessa forma, os programas e tabelas necessários para o funcionamento do sistema operacional da máquina são carregados a partir de meios magnéticos.

Alguns processos criados nas etapas de *bootstrap* correspondem a programas carregadores que possuem tempo de vida limitado, existindo apenas para carregar e conduzir outros processos para seus nós de destino. Esses processos, pelo fato de serem temporários, não necessitam de descritores de processo.

A figura 5-16 esquematiza as principais etapas da carga inicial do sistema.

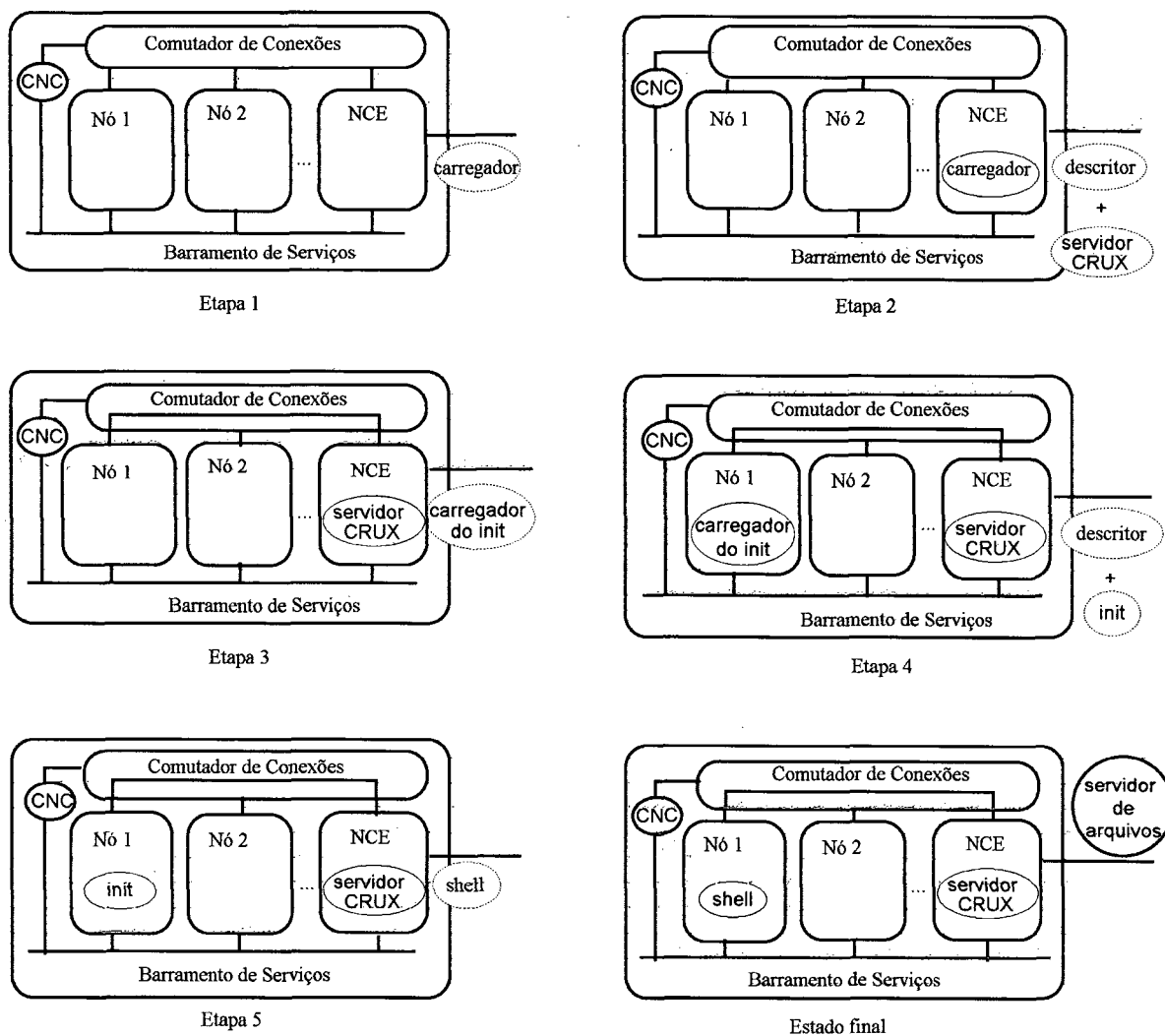


Figura 5-16. Etapas da carga inicial do sistema operacional CRUX.

Na etapa 1, o microcódigo do NCE, que possui contato com o mundo exterior, fica aguardando na linha de comunicação um programa *carregador* da estação de trabalho.

Na etapa 2, o carregador, já em execução no NCE, recebe o servidor CRUX e seu descritor, e os carrega no mesmo nó em que se encontra. Apesar de ainda ocupar espaço na memória, o carregador não mais executa e, para maior legibilidade, não é representado na figura.

Na etapa 3, o servidor CRUX requisita um nó ao CNC. Este seleciona um nó e estabelece uma conexão entre o nó alocado e o NCE. O servidor CRUX aguarda o programa *carregador do init*, o recebe e o reenvia para o nó alocado.

Em seguida, na etapa 4, o servidor CRUX recebe o processo *init* e seu descritor e os envia para o *carregador do init*, que os carrega no mesmo nó em que se encontra.

O *init* é um processo tradicional de sistemas operacionais compatíveis com UNIX, cuja principal tarefa é disparar um processo *login* em cada terminal existente. O processo *login* é responsável por controlar o acesso de usuários através de contas e senhas. Após o usuário ser autorizado a utilizar a máquina, o *init* é responsável por carregar o interpretador de comandos.

Finalmente, na etapa 5, o processo *init* carrega o processo interpretador de comandos (*shell*). Essa carga se dá através de uma chamada de sistema *exec*. Dessa forma, o *shell* é carregado sobre a imagem de memória do próprio *init*.

Ainda na figura 5-16, pode ser vista a configuração final do sistema após a sua carga.

No UNIX tradicional, todos os processos de usuários disparados são filhos ou descendentes diretos do *init* [BAC86]. Nessa filosofia, se um processo termina antes de um de seus filhos, o sistema torna tais "órfãos" como filhos do *init*. Uma característica exclusiva do sistema em execução no Nó //, é o fato do processo *init* não existir durante todo o tempo de vida do sistema. Isto deve-se ao fato de ser inconveniente manter um processo quase sem função ocupando um nó, como seria o caso do processo *init*. No Nó // já existe o servidor CRUX que é um processo permanente no sistema. Dessa forma, o servidor CRUX é escolhido como o processo pai "adotivo" de todos os órfãos do sistema. Dessa maneira, o Nó // libera o nó utilizado pelo *init* para execução de outros processos.

## 6. Conclusões

O simulador e o sistema operacional já estão funcionando, apesar de não estarem completos. A seguir é apresentada uma lista de pendências:

- No simulador, falta implementar na gerência de memória do XINU a utilização da memória expandida, como foi descrito em 4.3.1.
- No sistema operacional, diversas chamadas de sistema ainda não estão implementadas. No momento, as chamadas implementadas são: *open*, *creat*, *close*, *read*, *write*, *lseek*, *chdir*, *fork*, *exit*, *wait*, *exec*, *brk*, *getpid* e *getppid*. A implementação das chamadas restantes não é difícil, pois quase todas possuem um comportamento semelhante às chamadas resolvidas pelo servidor CRUX, descritas em 5.4.2.
- O programa *shell* está implementado da forma que foi descrita em 5.6. É necessário implementar posteriormente um shell completo conforme o padrão POSIX.2.
- Algumas funções da linguagem C necessitam ser implementadas para poder se compilar programas mais sofisticados. Exemplos são as funções de gerência de memória relacionadas à chamada de sistema *brk*, como a *malloc*, *calloc* e *free*.

O projeto e a realização deste trabalho constituiu uma tarefa extensa que foi realizada em conjunto por duas pessoas da equipe de Sistemas Operacionais (ver [CAM95]). Minha contribuição pessoal no processo de implementação é listada a seguir:

- *Simulador*
  - Alteração no gerente de memória do XINU para simular a memória existente em cada nó (ver 4.3.1).
  - Implementação das chamadas ao XINU através de *traps* (ver 4.3.1).

- *Sistema operacional CRUX*
  - Implementação das camadas de micronúcleo (ver 5.3).
  - Implementação das chamadas de sistema *fork*, *exec* e *getpid* (ver 5.4).
  - Implementação da carga inicial do sistema (ver 5.7).

## 6.1. Contribuições

A realização deste trabalho permitiu comprovar a viabilidade da implantação de um sistema operacional distribuído compatível, a nível da interface de programação com UNIX no multicomputador Nó //. O sistema operacional foi parcialmente implementado antes do término da construção da máquina, o que possibilitou o estudo e o aprimoramento de ambos.

Diversos outros trabalhos que dependem da existência do sistema operacional já estão sendo estudados como, por exemplo, a implementação da linguagem SuperPascal [HAN94] e o transporte de um sistema de arquivos distribuído proposto em [FRÖ94].

A contribuição desse trabalho também se reflete na construção do próprio multicomputador, pois um levantamento da carga de cada um de seus elementos e a detecção de eventuais gargalos são tarefas que já estão sendo realizadas utilizando o simulador construído.

## 6.2. Perspectivas

Após a construção do primeiro protótipo do multicomputador Nó //, colocam-se diversas propostas de melhorias no sistema CRUX para serem implementadas já na máquina real.

### • *Micronúcleo*

Propõe-se a implementação do micronúcleo como uma entidade separada na memória. Ele seria carregado em cada nó do Nó // durante a carga inicial do sistema. Esse enfoque apresenta duas vantagens:

- Não haveria necessidade de carregadores nas chamadas *fork* e *exec*. O próprio micronúcleo receberia e instalaria os processos e programas na memória.
- O sistema poderia ser protegido das aplicações, pois o micronúcleo seria chamado apenas através de uma interrupção de *software (trap)* – gerada pelas chamadas de sistema.

Coloca-se também como proposta a extensão dos serviços do micronúcleo. O número reduzido de seus serviços resulta de um enfoque minimizador que procurou deixar no micronúcleo apenas os serviços estritamente necessários. Propõe-se estender o micronúcleo de forma que ele possa oferecer um maior número de abstrações tais como, por exemplo, caixas postais e *threads*.

A implementação de caixas postais dentro do micronúcleo é um enfoque adotado no sistema CHORUS versão 3 [BRI91]. Uma alternativa viável e mais flexível é a implementação de serviços de caixa postal fora do micronúcleo, utilizando um servidor de comunicações. Essa última alternativa é proposta em [CAM95].



A adoção de *threads* se constitui numa proposta bem interessante. Embora a implementação do sistema CRUX sem esse conceito seja viável, como comprovou este trabalho, percebeu-se durante o seu desenvolvimento que a utilização de *threads* facilitaria muito a programação de eventos paralelos, abstraindo-os como eventos puramente seqüenciais.

Sem dúvida, a implementação de *threads*, complementada com um esquema de endereçamento indireto para as comunicações através das caixas postais, facilitaria em grande parte o desenvolvimento dos algoritmos adotados.

- *Servidor CRUX*

Propõe-se, também, o desdobramento do servidor CRUX em vários servidores. Isso permitiria uma distribuição melhor dos serviços oferecidos pelo sistema operacional. Proposta semelhante, está sendo feita em [CAM95], a qual corresponde a desdobrar os serviços do sistema em três servidores.

- *Chamadas de sistema*

A utilização da chamada *fork* seguida imediatamente de *exec* para a criação de um novo processo é ineficiente, pois implica numa cópia desnecessária da imagem do processo que executou *fork*. No UNIX multiprogramado tradicional, isto não se constitui num problema sério, pois a cópia do processo se faz na memória privativa do próprio processador e ainda conta com os recursos de memória virtual.

Propõe-se, no sistema CRUX a concepção de uma chamada de sistema que combine as operações dessas duas chamadas, criando um novo processo a partir de um arquivo executável.

Uma outra proposta interessante é a extensão da interface de programação do CRUX, de forma a oferecer serviços de comunicação que seriam utilizados nas linguagens paralelas. Essas chamadas de sistema seriam implementadas utilizando os serviços da camada de comunicação entre processos do micronúcleo.

- *Carga de processos*

Os programas a serem carregados no sistema possuem um formato simples conforme visto em 5.2. Uma proposta de melhoria é a carga de programas no formato COFF (*Common Object File Format*) [GIR88] que é um formato padronizado em sistemas UNIX.

## Apêndice A - O Sistema XINU

XINU é um sistema operacional proposto por Douglas Comer em seu livro [COM84]. Todo o código do XINU é apresentado e explicado no livro de forma a facilitar o seu entendimento, alteração, expansão e transporte para outras arquiteturas. O livro é um guia para o projeto e o desenvolvimento de um sistema operacional em camadas. Cada capítulo do livro explica as funções de uma camada do XINU, resultando num sistema capaz de suportar múltiplos processos, comunicação em rede e um sistema de arquivos.

As camadas componentes do XINU são mostradas na figura A-1.

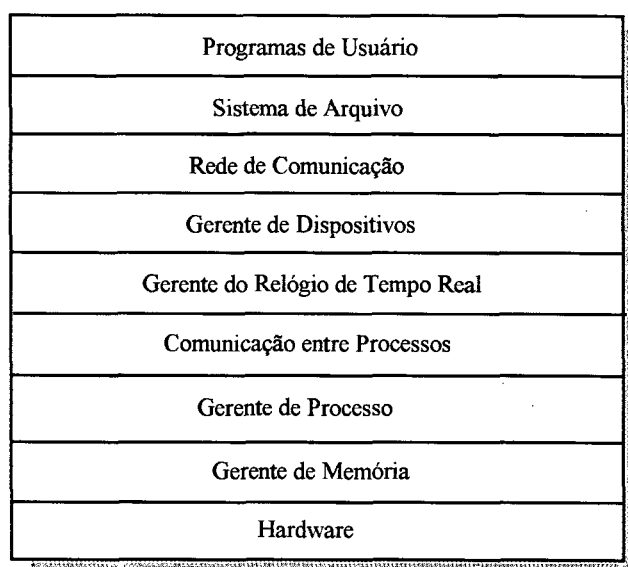


Figura A-1. Camadas do sistema XINU

Programas executados no XINU têm acesso aos serviços oferecidos através de suas chamadas de sistema. A seguir são apresentadas, brevemente, em ordem alfabética as principais chamadas e seus protótipos de função.

**create** - Cria um novo processo.

```
int create ( char * caddr, int ssize, int prio, char * name, int nargs, ... );
```

A função *create* cria um novo processo XINU que começará a ser executado na posição *caddr*, com uma pilha de tamanho *ssize*, prioridade inicial *prio* e nome de identificação *name*. O parâmetro *caddr* deve ser o endereço de um procedimento. Podem ser passado parâmetros adicionais a essa função. Neste caso, *nargs* deve ter um valor maior que 0.

Caso a criação obtenha sucesso é retornado *id* – o identificador do processo criado.

**getmem** - Aloca a memória.

```
char * getmem ( int nbytes );
```

Aloca um bloco de memória de tamanho *nbytes* e retorna o endereço menor do bloco da memória alocado.

**kill** - Termina um processo.

protótipo:

```
int kill( int pid );
```

Essa chamada pára o processo *pid* e o remove do sistema. Um processo XINU pode ser morto em qualquer estado, inclusive quando está suspenso.

***receive*** - Recebe uma mensagem.

```
int receive();
```

Recebe uma mensagem enviada de um processo. Se não houver mensagens, *receive* bloqueia o processo até recebê-la.

***resume*** - Reinicia um processo suspenso.

```
int resume( pid );
```

Essa função tira o processo *pid* do estado de hibernação e permite que ele reinicie sua execução.

***send*** - Envia uma mensagem.

```
int send( pid, msg );
```

Esta função envia uma mensagem *msg* para um processo *pid*.

***suspend*** - Suspende um processo.

```
int suspend( pid );
```

Esta função coloca o processo *pid* em estado de hibernação.

## Referências Bibliográficas

- [AUS91] Austin, P. & Murray, K. & Wellings, K., *The Design of an Operating System for a Scalable Parallel Computing Engine*, Software - Practice and Experience, vol. 21, p. 989-1013, outubro de 1991.
- [BAC86] Bach, M. J., *The Design of the UNIX Operating System*, Prentice-Hall, 1986.
- [BAL87] Bal, H. E. & Tanenbaum, A. S. & Kaashoek, M. F., *Orca A Language for Distributed Programming*, Report IR-40, Depto of Mathematics and Computer Science, Vrije Universiteit, dezembro de 1987.
- [BAL88] Bal, H. E. & Tanenbaum, A.S., *Distributed Programming with Shared Data*, IEEE Conf. on Computer Languages, IEEE, pág. 82-91, 1988.
- [BEN90] Ben-ari, *Principles of Concurrent and Distributed Programming*, Prentice-Hall, 1990.
- [BOO91] Booch, G., *Object Oriented Design with Applications*, Benjamin/Cummings Publishing Company, 1991.
- [BRI91] Bricker, A. & et al., *Architectural Issues in Microkernel-based Operating Systems: the CHORUS experience*, Computer Communications, vol. 14, n. 6, p. 347-357, agosto de 1991.
- [CAM95] Campos, R. A., *Um Sistema Operacional Fundamentado no Modelo Cliente-Servidor e Um Simulador Multiprogramado para Multicomputador*, Dissertação de Mestrado, CPGCC-UFSC, Florianópolis, 1995.

- [COM84] Comer, D., *Operating System Design: The XINU Approach*, Prentice-Hall, 1984.
- [COR93] Corso, T. B., *Ambiente para Programação Paralela em Multicomputador*, Depto de Informática e Estatística, UFSC – Florianópolis, Relatório Técnico, 1993.
- [COU88] Coulouris, G. F. & Dollimore, J., *Distributed Systems: Concepts and Design*, Addison-Wesley, 1988.
- [CRI88] Crichlow, J. M., *An Introduction to Distributed and Parallel Computing*, Prentice-Hall, 1988.
- [CRI94] Crisóstomo, V. L., *Um Mecanismo de Comunicação e Um Método de Ativação de Servidores para Um Sistema Operacional*, Dissertação de Mestrado, CPGCC-UFSC, Florianópolis, 1994.
- [DUN86] Duncan, R., *Advanced MS-DOS*, Microsoft Press, 1986
- [FEN81] Feng, T., *A Survey of Interconnection Networks*, Computer, p. 12-27, dezembro de 1981.
- [FRÖ94] Fröhlich, A. A., *PYXIS – Um Sistema de Arquivos Distribuído*, Dissertação de Mestrado, CPGCC-UFSC, Florianópolis, 1994.
- [GIR88] Gircys, G., *Understanding and Using COFF*, O'Reilly & Associates, 1988.
- [HAN73] Hansen, B., *Operating System Principles*, Prentice-Hall, 1973.
- [HAN89] Hansen, B., *The Joyce Language Report*, Software - Practice and Experience, vol. 19, n. 6, p. 553-578, outubro de 1989.

- [HAN94] Hansen, B. & P., *SuperPascal - a Publication Language for Parallel Scientific Computing*, *Concurrency - Practice and Experience*, vol. 6(5), 461-483, agosto de 1994.
- [HOA78] C. A. R. Hoare, *Communicating Sequential Processes*, *Communications of the ACM*, vol. 21, n. 8, p. 666-677, agosto de 1978.
- [HWA85] Hwang, K & Briggs, F. A., *Computer Architecture and Parallel Processing*, McGraw-Hill, 1985.
- [HWA93] Hwang, K, *Advanced Computer Architecture*, McGraw-Hill, 1993.
- [KIR94] Kirck, O. *The Linux Network Administrator's Guide*, 1994.
- [LEW91] Lewis, D., *POSIX Programmer's Guide*, O'Reilly & Associates, 1991.
- [MER95] Merkle, C. & Boing, H., *Simulação do Nó //*, Depto de Informática e Estatística, UFSC – Florianópolis, Relatório Técnico a ser publicado, 1995.
- [MUL88] Mullender, S. J., *Distributed Operating Systems: State-of-the-Art and Future Directions*, Proc. of the EUTECO 88, Viena, pág. 57-66, 1988.
- [PAS94] Pasin, M., TRIX, *Um Sistema Operacional Multiprocessado para Transputers, com Gerência Distribuída de Processos*, Dissertação de Mestrado, CPGCC-UFRS, Porto Alegre, 1994.
- [POU94] Pountian D, *The Chorus Microkernel*, BYTE, pág. 131-136, janeiro de 1994.
- [REE87] Reed, D. A. & Fujimoto, R. M., *Multicomputer Networks: Message-Based Parallel Processing*, MIT Press, 1994.



- [ROD93] Rodger & Jacquemot & Pillevesse, *COOL: System Support for Distributed Programming*, Communications of the ACM, vol. 36, n. 9, p. 37-46, setembro de 1993.
- [ROZ89] Rozier, M. & et al., *CHORUS Distributed Operating Systems*, Computing Systems, vol. 1, n. 4, fevereiro de 1989.
- [SIL94] Silberschatz, A. P. & Galvin, *Operating Systems Concepts*, Addison-Wesley, 1994.
- [STE92] Stein, B. O., *Projeto do Núcleo de um Sistema Operacional Distribuído*, Dissertação de Mestrado, CPGCC-UFRS, Porto Alegre, 1992.
- [TAN87] Tanenbaum A. S., *Operating Systems: Design and Implementation*, Prentice-Hall, 1987.
- [TAN92] Tanenbaum A. S., *Modern Operating Systems*, Prentice-Hall, 1992.
- [TAN] Tanenbaum A. S. & Mullender, S. J., *An Introduction to Amoeba*, Artigo Técnico.
- [TAY89] Taylor S., *Parallel Logic Programming Techniques*, Prentice-Hall, 1989.
- [VAR94] Varhol P. D., *Small Kernels Hit It Big*, BYTE, pág. 119-128, janeiro de 1994.