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

UM MECANISMO DE COMUNICAÇÃO E UM MÉTODO DE ATIVAÇÃO DE SERVIDORES PARA UM SISTEMA OPERACIONAL

por

Vicente Lima Crisóstomo

Dissertação submetida como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Prof. Thadeu Botteri Corso
Orientador

Florianópolis, Agosto de 1994

UM MECANISMO DE COMUNICAÇÃO E UM MÉTODO DE ATIVAÇÃO DE SERVIDORES PARA UM SISTEMA OPERACIONAL

VICENTE LIMA CRISÓSTOMO

ESTA DISSERTAÇÃO FOI JULGADA 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. Hermann Adolf Harry Lücke, Dr. Ing.
Coordenador do Curso

BANCA EXAMINADORA

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

Prof. Luiz Fernando Jacinto Maia, Dr.

Prof. Philippe Olivier Alexandre Navaux, Dr.

Prof. Simão Sirineo Toscani, Dr.

A meus pais, Maura e José.

AGRADECIMENTOS

Agradeço a todos que contribuiram de alguma forma para que eu pudesse realizar este trabalho.

Meu agradecimento especial ao professor Thadeu Botteri Corso que, mais que orientador, é amigo, pelos ensinamentos seguros e pelo apoio de sempre.

Aos colegas de curso e amigos desde os primeiros momentos desta jornada: Marisa Jordan, a nossa querida Jordan; e Alexandre Moraes Ramos, o velho Brasília, pela amizade e incentivo.

Agradeço a minha amiga Carla Merkle que muito me ouviu, aconselhou e estimulou.

Aos amigos Carlos Montez e Rodrigo, colegas que ocupei inúmeras vezes para esclarecimento de dúvidas.

À amiga Verinha, a dona Vera, pela compreensão e atenção que me dedicou durante o curso.

Aos amigos da Universidade Federal do Ceará, especialmente: Gerson, Ednilce, Hannelore, Ítalo e Marcos França, que me ajudaram e encorajaram desde os primeiros momentos.

E agradeço também ao anônimo trabalhador pagador de impostos de nosso país.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE ABREVIATURAS

FS - FILE SYSTEM

ftp - file transfer protocol

MIMD - Multiple Instruction Stream, Multiple Data Stream

MISD - Multiple Instruction Stream, Single Data Stream

MM - MEMORY MANAGER

OSI - Open Systems Interconnection

SIMD - Single Instruction Stream, Multiple Data Stream

SISD - Single Instruction Stream, Single Data Stream

TCP/IP - Transport Control Protocol/Internet Protocol

UFSC - Universidade Federal de Santa Catarina

RESUMO

A implementação de sistemas operacionais que permitam explorar a capacidade de processamento de sistemas formados por múltiplos processadores é uma necessidade crescente considerando o fato de que a distribuição de processamento vem se constituindo em uma tendência irreversível na informática.

Este trabalho analisa aspectos relacionados com os sistemas operacionais para sistemas computacionais distribuídos, dando ênfase aos tópicos das comunicações entre processos e ao oferecimento de serviços a seus usuários.

São propostos um mecanismo de comunicação entre processos e um método de ativação de servidores por demanda para sistemas operacionais baseados no modelo cliente-servidor. São apresentados aspectos para inclusão dessas propostas em um sistema operacional escolhido como plataforma de experimentação e sua adaptação a um multicomputador específico.

ABSTRACT

The implementation of operating systems which allow exploiting the processing capacity of systems composed by several processors is an increasing necessity considering the fact that the processing distribution has become an irreversible trend in computation.

This work analises aspects related with the operating systems to distributed systems, enphasysing the topics of interprocess communication and the offering of services to its users.

An interprocess communication mechanism and a method of activating servers dynamically based on the client-server model are proposed. Aspects to the inclusion of such proposals in an operating system chosen as platform of experimentation and its adaptation to a specific multicomputer are presented.

1 INTRODUÇÃO

1.1 Panorama Atual

A utilização de sistemas computacionais compostos por múltiplos processadores é hoje uma realidade incontestável decorrente da tendência de descentralização das atividades de processamento de dados das organizações, acompanhada de avanços tecnológicos que permitiram a queda dos preços do *hardware*. Outras motivações que levaram à adoção de tais sistemas foram a existência de aplicações tipicamente distribuídas na área de processamento de dados de organizações empresariais e a necessidade da obtenção de respostas mais rápidas para problemas científicos através de processamento paralelo.

Inúmeras são as arquiteturas surgidas para a construção de equipamentos computacionais formados por vários processadores fazendo ou não uso de memória compartilhada. O desenvolvimento do *software* para elas não tem acompanhado o mesmo ritmo da evolução do *hardware* [REE87]. Várias linhas de pesquisa têm procurado apresentar soluções para a construção de *software* para máquinas com arquitetura distribuída ou paralela. Temas como sistemas operacionais distribuídos e paralelos, linguagens distribuídas e aplicações distribuídas são cada vez mais tratados na literatura.

1.2 Motivações e Objetivos do Trabalho

Este trabalho faz parte do projeto Nó// [COR93] que visa a obtenção de um ambiente para programação paralela e distribuída através da construção de um multicomputador e da implementação de um sistema operacional experimental para ele. O trabalho aborda uma fração do problema relacionado a sistemas operacionais para máquinas com arquitetura distribuída. A motivação foi a busca de alternativas eficientes para o problema das comunicações entre processos, que é crucial em multicomputadores, e de soluções flexíveis para o acesso aos serviços de sistemas operacionais por parte de seus usuários. Fixamos como objetivos para o trabalho, a definição de um mecanismo de comunicação específico que fosse eficiente e de um

método de provimento de serviços em um sistema operacional, visando a incorporação dos mesmos ao sistema operacional previsto para o multicomputador Nó//.

Traçamos como etapas para atingirmos esses objetivos o estudo de arquiteturas distribuídas e de sistemas operacionais para elas; o estudo de um sistema operacional específico e da arquitetura do multicomputador Nó// para serem usados como plataformas de experimentação; e a concepção das propostas fixadas como objetivos, seguidas de um estudo visando a incorporação desses mecanismos às plataformas de experimentação.

1.3 Conteúdo do Trabalho

Esta dissertação está dividida em 6 capítulos cujos conteúdos descrevemos a seguir.

No capítulo 2, apresentamos uma visão geral de sistemas computacionais com arquiteturas compostas por múltiplos processadores, através de classificações e de alguns exemplos de topologias existentes, além de introduzirmos a arquitetura do multicomputador Nó//.

No capítulo 3, apresentamos um estudo sobre sistemas operacionais para máquinas com arquitetura distribuída ou paralela, abordando aspectos relacionados com os mecanismos de comunicação entre processos, além da descrição suscinta de alguns exemplos de tais sistemas.

No capítulo 4, descrevemos o sistema operacional MINIX que foi utilizado como ferramenta de estudo e trabalho para a experimentação das propostas apresentadas nos capítulos 5 e 6. Além disso, fazemos também considerações sobre sua aplicação ao multicomputador Nó//.

No capítulo 5, propomos um mecanismo de comunicação apresentado da seguinte forma: inicialmente mostramos a definição do mecanismo, então fazemos uma proposta para sua implementação no núcleo do sistema MINIX e, por último, consideramos sua implementação no multicomputador Nó//.

No capítulo 6, propomos um método de ativação de servidores apresentado da seguinte forma: inicialmente mostramos a definição do método de ativação de servidores e sua

viabilização por meio da implementação de um servidor de nomes de serviços, então fazemos uma proposta para sua introdução no sistema MINIX e, por último, consideramos sua aplicabilidade ao multicomputador Nó//.

Finalmente, no capítulo 7 apresentamos conclusões do nosso trabalho e perspectivas de sua evolução.

2 MULTICOMPUTADORES

Neste capítulo, introduzimos os sistemas de computação compostos por mais de um processador, as características dos multiprocessadores e multicomputadores e a arquitetura do multicomputador Nó//, que constitui a plataforma material para a qual nosso trabalho se orientou.

2.1 Máquinas Paralelas

Os primeiros equipamentos computacionais que faziam uso de mais de um processador apareceram já na década de 70. Difícil tem sido estabelecer uma classificação estável para tais sistemas que começaram a se desenvolver rapidamente, apresentando a cada momento novas topologias.

Propostas de esquemas de classificação para sistemas com múltiplos processadores vêm surgindo desde que foram implantados os primeiros de tais sistemas. Em [AND75] é apresentada uma taxonomia para sistemas de computadores interconectados, onde já se discute dez arquiteturas de computadores então chamados de "processadores distribuídos", "computadores com função distribuída" ou "redes de computadores". Uma outra abordagem encontra-se em [TAN92], onde é tratado o tema dos sistemas distribuídos levando em consideração tanto os aspectos de *hardware* quanto de *software*.

Dois fatos novos surgidos na década de 80 vieram agilizar o desenvolvimento dos multicomputadores. Primeiro, o desenvolvimento de poderosos microprocessadores que permitiam a máquinas de pequeno porte muitas vezes terem o poder de um *mainframe* por um preço bem menor. Segundo, a criação de redes locais de alta velocidade permitindo que grande número de máquinas interconectadas transferissem dados em alta velocidade entre elas. Muito se investiu nestas duas tecnologias de modo que hoje é possível interconectar-se sistemas computacionais formados por grande número de processadores interligados por uma rede de comunicação de alta velocidade. A denominação "sistema distribuído" para tais ambientes surge então em contraste aos sistemas centralizados tradicionais compostos de um único processador. Alguns autores fazem uma distinção entre **sistemas distribuídos** e **sistemas paralelos**. Os

primeiros seriam projetados para permitir que muitos usuários trabalhem concomitantemente enquanto os outros teriam como objetivo aumentar a velocidade de um determinado programa pela execução paralela. Tanenbaum [TAN92] adota a terminologia "sistema distribuído" para referir-se a qualquer sistema que faz uso de múltiplos processadores.

Sistemas distribuídos requerem *software* bem distinto dos utilizados pelos sistemas centralizados e a implementação de sistemas operacionais para esse tipo de *hardware* é algo que só começou a surgir recentemente.

O uso de sistemas distribuídos tem-se difundido devido a quatro fatores principais, que podem ser considerados vantajosos se comparados aos sistemas centralizados. O fator **econômico** está relacionado com o rápido barateamento dos processadores, obtendo melhor relação preço/desempenho através do uso de processadores agrupados. A existência de **aplicações tipicamente distribuídas**, como por exemplo sistemas de automação bancária, é outro motivo significativo para a utilização de tais sistemas. Uma razão marcante é a possibilidade de **expansão gradual** do sistema distribuído pelo acréscimo de processadores, quando necessário, sem perda da capacidade existente. Pode-se ainda considerar a **confiabilidade** que está relacionada com eventuais falhas do sistema. No caso de um sistema distribuído, a falha de um processador pode ser restrita ao componente defeituoso do sistema.

Além das vantagens já citadas sobre os sistemas centralizados, os sistemas distribuídos também apresentam pontos positivos em relação a conjuntos de computadores pessoais independentes. Uma delas, é o compartilhamento de bases de dados que é essencial em algumas aplicações, como em um sistema de reservas de uma companhia aérea. Outra, é a possibilidade de compartilhamento de dispositivos periféricos. A distribuição da carga de processamento entre os vários componentes do sistema confere a ele maior flexibilidade. Além destes aspectos, os sistemas distribuídos permitem novas formas de comunicação entre seus usuários como o correio eletrônico, por exemplo.

Há porém alguns pontos a serem ponderados com relação a essa nova tecnologia. O principal problema para a utilização de tais sistemas reside na indisponibilidade de *software* para eles uma vez que só agora começam a surgir os primeiros sistemas operacionais para tais arquiteturas, havendo muito a ser explorado. Outro problema é a rede de comunicação entre os

nós do sistema, sendo necessário *software* apropriado para manipulá-la correta e eficientemente. Ainda outro ponto que deve ser levado em consideração é a segurança de dados, uma vez que o *software* para um sistema distribuído deve garantir confidencialidade dos dados pertencentes a seus usuários.

Em [HWA85] são apresentados três esquemas de classificação para arquiteturas de sistemas computacionais em geral. Uma delas é a classificação de Flynn (1966) que se baseia na multiplicidade de fluxos de dados e de instruções no sistema, a qual será abordada a seguir.

O critério de Flynn para classificação de sistemas computacionais leva em consideração a multiplicidade de fluxos de instruções e dados existentes no sistema. A execução de uma seqüência de instruções sobre um determinado conjunto de dados é um processo fundamental em computação. Um fluxo de instruções é uma seqüência de instruções que executam em determinada máquina. Um fluxo de dados é uma seqüência de dados tratados por um fluxo de instruções. De acordo com o sistema provido para servir aos fluxos de instruções e de dados, os sistemas computacionais são, então, classificados em quatro organizações como segue: Fluxo de Instrução Único- Fluxo de Dados Único (**SISD** - *Single Instruction Stream, Single Data Stream*), Fluxo de Instrução Único- Múltiplos Fluxos de Dados (**SIMD** - *Single Instruction Stream, Multiple Data Stream*), Múltiplos Fluxos de Instruções - Fluxo de Dados Único (**MISD** - *Multiple Instruction Stream, Single Data Stream*), e Múltiplos Fluxos de Instruções - Múltiplos Fluxos de Dados (**MIMD** - *Multiple Instruction Stream, Multiple Data Stream*).

Sistemas com organização **SISD** são representados por todos os sistemas monoprocessadores tradicionais, desde computadores pessoais até *mainframes*. Neles, há uma unidade de controle e um processador, sendo as instruções executadas seqüencialmente sobre um único fluxo de dados.

Sistemas com organização **SIMD** têm uma unidade de controle que gere vários processadores. Uma instrução é buscada e executada paralelamente nos processadores sobre fluxos de dados distintos. Em tais sistemas, a memória pode ser dividida em módulos, podendo cada módulo estar associado a um processador. Processadores matriciais e alguns supercomputadores representam esta categoria.

Sistemas com organização **MISD** não existem na prática. No entanto, tais sistemas seriam compostos por várias unidades de controle e processadores executando diferentes fluxos de instruções sobre o mesmo fluxo de dados. Os resultados do processamento sobre os dados em um processador tornam-se os dados de entrada para o próximo.

Sistemas com organização **MIMD** são compostos por um grupo de unidades de controle e de processadores. Cada fluxo de instruções atua sobre um determinado fluxo de dados em um determinado processador. Tal organização corresponde a um grupo de computadores independentes, cada um executando suas próprias instruções sobre seus próprios dados.

2.2 Máquinas MIMD

Os sistemas MIMD podem ser classificados em dois subgrupos [TAN92]: **multiprocessadores** e **multicomputadores**. A diferença entre eles reside no fato de que nos primeiros, os processadores compartilham memória, havendo assim um espaço de endereçamento único para todos eles. Já nos multicomputadores, cada componente do sistema possui um processador com memória privativa e canais de comunicação. A figura 2-1 apresenta tal classificação.

Os sistemas multiprocessadores e multicomputadores são ainda subdivididos conforme a arquitetura da rede de interconexão que poderá estar baseada em um **barramento** ou em um esquema **chaveado**. Os sistemas baseados em **barramento** possuem um meio de comunicação ao qual todos os componentes do sistema estão conectados e é utilizado para a troca de mensagem entre eles. Já no sistema **chaveado** não há esse meio comum mas sim ligações individuais, máquina a máquina, com diversos padrões distintos em uso.

Há ainda uma consideração sobre tais sistemas que está relacionada com o grau de coesão existente entre seus componentes. Assim, os sistemas podem ser **fracamente** ou **fortemente acoplados**. A diferença entre eles reside na velocidade em que dados são transferidos entre os módulos componentes do sistema, sendo significativamente mais alta em sistemas fortemente acoplados. Um exemplo de sistema fortemente acoplado seria um sistema composto por processadores numa mesma placa de circuito impresso compartilhando uma memória única.

Já dois computadores pessoais que se comunicam através da rede telefônica seria um exemplo de sistema com fraca coesão entre seus componentes.

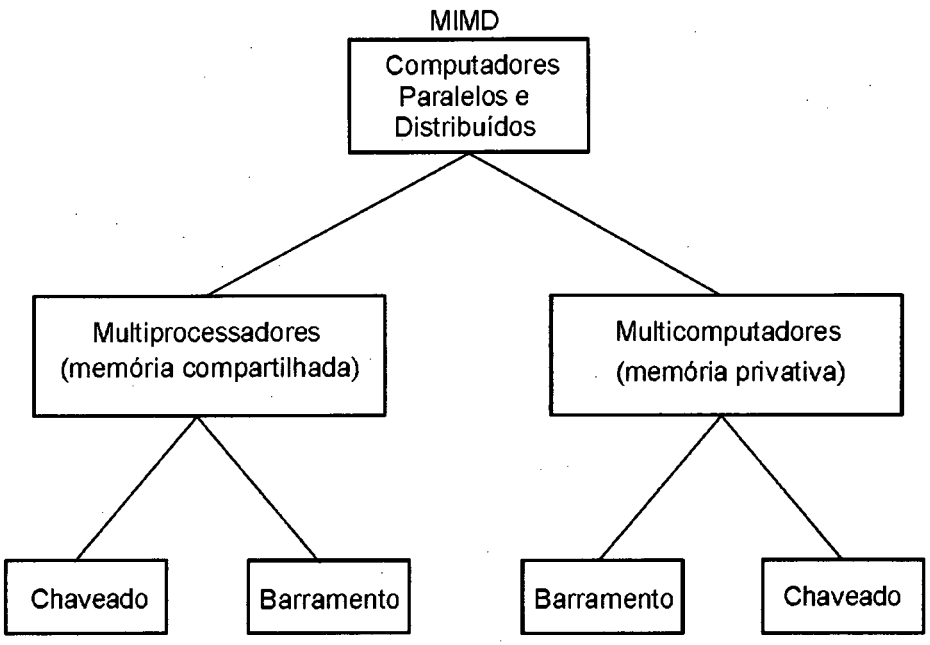

Figura 2-1. Uma taxonomia para sistemas computacionais com múltiplos processadores

2.2.1 Multiprocessadores Baseados em Barramento

Os multiprocessadores baseados em barramento são compostos por vários processadores e um módulo de memória, todos conectados a um barramento comum conforme mostra a figura 2-2.

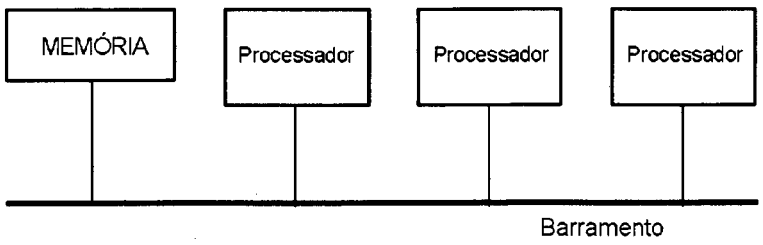

Figura 2-2. Multiprocessador baseado em barramento

Requisições de leitura à memória são feitas com a colocação do endereço da palavra desejada no barramento e um sinal de leitura nas linhas de controle. A memória capta esse endereço e o sinal e põe o conteúdo da palavra referida no barramento para o processador requisitante captá-lo. A gravação de um valor em determinado endereço da memória ocorre de forma análoga: o processador põe o valor a ser gravado e o endereço da palavra no barramento acompanhado do sinal de controle de gravação.

Uma vez modificado o conteúdo de determinada palavra de memória por um determinado processador, qualquer acesso subseqüente por qualquer processador obterá o novo valor daquela palavra. Isto expressa o conceito de memória **coerente**.

O problema desse tipo de multiprocessador reside na capacidade do barramento, que poderá ficar sobrecarregado mesmo com um número reduzido de processadores, observando-se significativo decréscimo de desempenho à medida que cresce o número de processadores. Uma solução possível para amenizar esse fenômeno é a introdução de uma memória *cache* de alta velocidade entre cada processador e o barramento. Nesse caso, tais memórias conteriam cópias das palavras da memória do sistema mais recentemente acessadas e captariam todas as requisições de acesso a memória respondendo diretamente ao processador caso ela já contenha a palavra requisitada, evitando o acesso ao barramento. O problema, nessas situações, é a necessidade de coerência entre os valores da memória do sistema e aqueles contidos em cada *cache*. Modificações em palavras de memória armazenadas na *cache* devem também ser feitas na memória do sistema. Isso pode ser conseguido fazendo com que cada *cache* monitore constantemente o barramento, captando todas as ocorrências de gravações em palavras da memória que ela contenha. A utilização desse esquema tem possibilitado a construção de multiprocessadores com 32 a 64 processadores em um único barramento.

2.2.2 Multiprocessadores Chaveados

Multiprocessadores podem também ser construídos com a memória compartilhada dividida em módulos e uma rede de interconexão que permita que todos os processadores tenham acesso a todos os módulos de memória. Tal rede consiste de um comutador de conexões que pode estabelecer uma total interconexão, havendo uma configuração de chaveamento para cada

par de processador e memória do sistema. Um exemplo desse tipo de rede é o *crossbar* mostrado na figura 2-3. Um ponto de chaveamento do *crossbar* pode ser fechado e aberto em *hardware* durante a operação do sistema. O acesso exclusivo de um processador a uma memória é obtido com o fechamento de um único ponto de chaveamento entre eles.

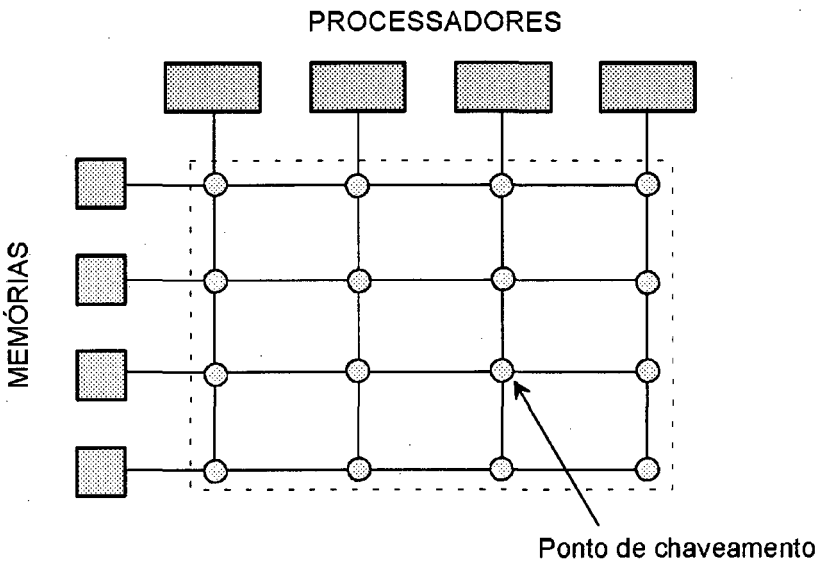

Figura 2-3. Um comutador de conexões *crossbar*

Uma outra alternativa para multiprocessadores chaveados é ter-se uma rede com menos pontos de chaveamento mas que continue garantindo o acesso de todos os processadores a todos os módulos de memória. Esse é o caso, por exemplo, da rede *ômega* mostrada na figura 2-4, onde o número de pontos de chaveamento é bem menor em relação ao *crossbar* para o mesmo número de nós do sistema sendo portanto mais lenta a conexão de processador a memória.

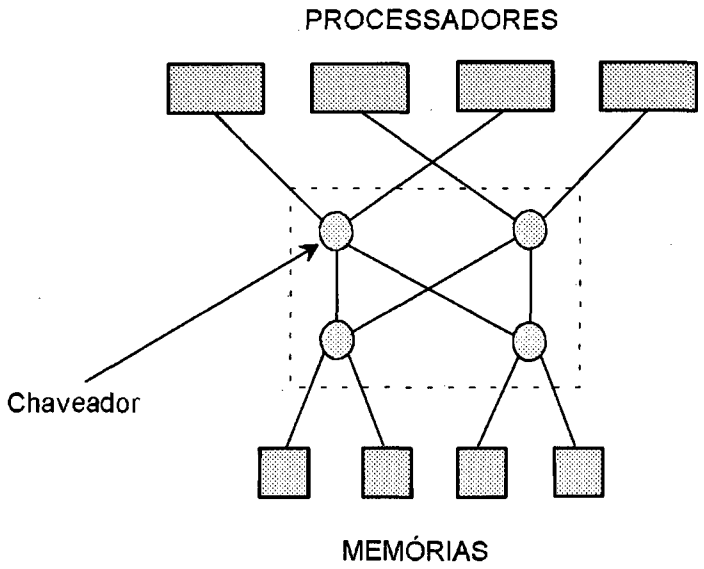

Figura 2-4. Um comutador de conexões ômega

A vantagem dos multiprocessadores que usam comutadores de conexões é permitir o acesso efetivamente simultâneo de processadores a módulos distintos de memória.

Nos multiprocessadores baseados em barramento o número máximo de processadores fica limitado pela capacidade do barramento empregado. O multiprocessadores chaveados apresentam uma possibilidade de aumentar-se o número desses nós componentes, embora por um custo mais elevado. A construção de sistemas computacionais multiprocessadores com grande número de componentes (processadores e módulos de memória) é ainda difícil e cara. A construção de multicomputadores é então uma alternativa viável para a utilização de sistemas com múltiplos processadores.

2.2.3 Multicomputadores Baseados em Barramento

Os multicomputadores necessitam de uma rede eficiente para interconexão de seus nós. No entanto, o volume de tráfego na rede para comunicação entre processadores é bem mais baixo que o necessário para acesso à memória em multiprocessadores uma vez que a rede é utilizada somente para troca de mensagens entre processos de processadores distintos. A rede de interconexão dos nós de um multicomputador pode ser um barramento como mostrado na figura 2-5. Os nós do sistema podem ser estações de trabalho de uma rede local ou uma coleção de processadores agrupados em uma mesma placa.

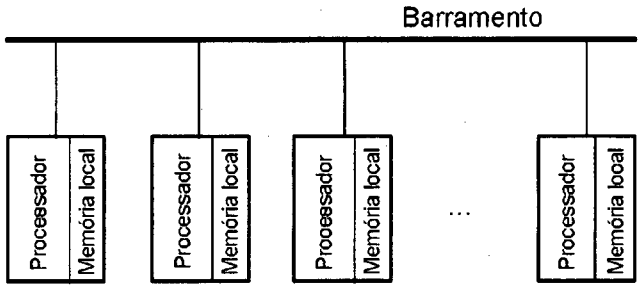

Figura 2-5. Multicomputador com barramento

2.2.4 Multicomputadores Chaveados

Os multicomputadores também podem ser construídos utilizando redes com conexões chaveadas ao invés de barramento. São várias as topologias para as redes de interligação entre nós de um multicomputador. Elas podem ser classificadas como **estáticas** ou **dinâmicas** [FEN81]. As topologias **estáticas** são aquelas nas quais os nós da máquina estão ligados direta e estaticamente a outros. Exemplos de sistemas com topologia estática são a grelha e o hipercubo. A comunicação entre nós não diretamente ligados ocorre por intermédio de outros nós. Nas topologias **dinâmicas**, os nós não estão ligados diretamente uns aos outros mas sim através de um comutador de conexões. Exemplos de máquinas com topologia dinâmica são a rede ômega e o *crossbar*.

A grelha consiste de uma coleção de processadores conectados como na figura 2-6. A grelha elementar é constituída de um nó e cresce pelo acréscimo de linhas e colunas de processadores conectados aos existentes.

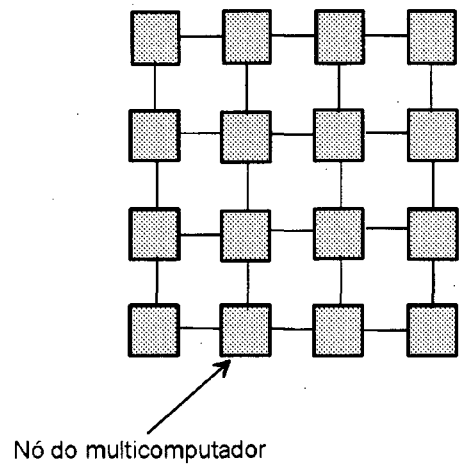

Figura 2-6. Multicomputador com rede grelha

O hipercubo ou n-cubo [REE87] é uma máquina cuja topologia é caracterizada pelo parâmetro n, denominado sua dimensão, que determina o número de ligações de cada nó. O valor de n é uniforme em todo o sistema. Um hipercubo de dimensão 0 é composto por um único nó, um hipercubo de dimensão 1 é composto de dois nós, um de dimensão 2 é formado de quatro nós e assim por diante. Um hipercubo de dimensão n+1 é obtido como a composição de dois hipercubos de dimensão n. A figura 2-7 ilustra o hipercubo de dimensão 3.

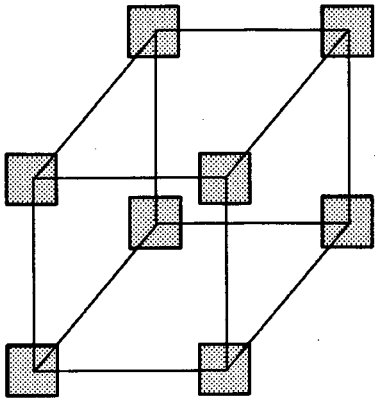

Figura 2-7. O hipercubo de dimensão 3

Tanto na grelha como no hipercubo, uma mensagem pode eventualmente percorrer vários nós para atingir seu destino. No entanto, a rota percorrida entre os nós mais distantes em ambas as topologias para o mesmo número de nós é mais longa na grelha.

Um outro exemplo de topologia de um multicomputador utiliza um comutador de conexões do tipo *crossbar* como na figura 2-8, no qual cada processador tem duas ligações com o *crossbar*.

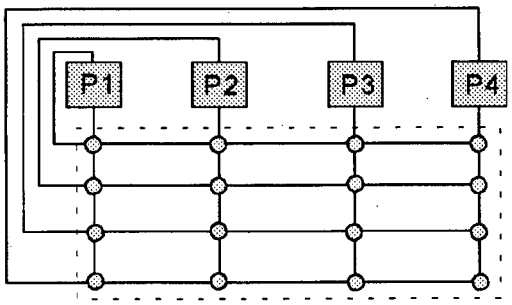

Figura 2-8. Um multicomputador que utiliza um *crossbar*

2.3 A Arquitetura do Nó//

O projeto Nó// [COR93] em desenvolvimento no Curso de Pós-Graduação em Ciências da Computação da UFSC, pelos grupos de sistemas operacionais e arquitetura de computadores, objetiva (1) a construção de um multicomputador com rede de interconexão dinâmica que ofereça grande flexibilidade para a comunicação entre seus nós, (2) a definição de mecanismos de comunicação que permitam expressar programas paralelos como redes de processos comunicantes e (3) a concepção e implementação de um sistema operacional experimental como uma rede de processos comunicantes.

A arquitetura da máquina paralela projetada é a de um multicomputador que faz uso de dois tipos de redes de comunicação para interligação de seus nós como mostra a figura 2-9: um comutador de conexões de tipo *crossbar* e um barramento dito de serviço. Cada nó está conectado ao comutador de conexões e ao barramento de serviço e é formado por um processador e memória própria. O barramento de serviço é utilizado para troca de pequenas mensagens, tipicamente de controle. Os canais estabelecidos no comutador de conexões são utilizados para troca de mensagens de maiores dimensões. Há um nó responsável pela gerência do barramento de serviço e do comutador de conexões chamado **nó de controle**.

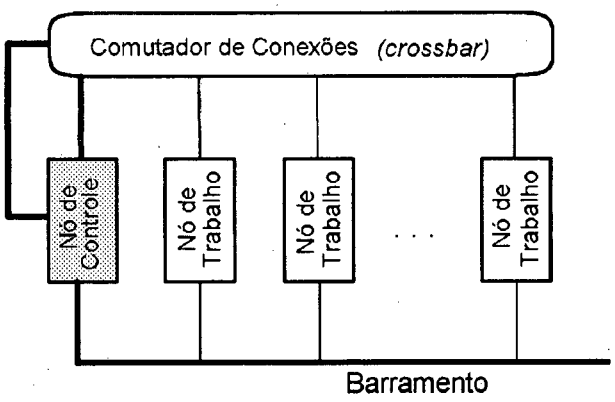

Figura 2-9. Arquitetura da máquina Nó//

Prevê-se a ligação de um dos nós a uma estação de trabalho com o objetivo de fazer uso dos recursos da estação no desenvolvimento do projeto. A denominação Nó// (Nó Paralelo) decorre da intenção final do projeto de incorporar várias dessas máquinas como nós de redes locais.

3 SISTEMAS DISTRIBUÍDOS

Neste capítulo, apresentamos uma classificação para os sistemas operacionais distribuídos bem como o problema fundamental da comunicação entre processos nesses sistemas. Também são discutidos alguns exemplos de sistemas distribuídos atuais.

3.1 Sistemas Operacionais

A construção de sistemas operacionais para ambientes com múltiplos processadores tem sido bastante solicitada, uma vez que a variedade e o número desses ambientes vem crescendo rapidamente. Segundo [TAN92], eles podem ser classificados em: sistemas operacionais para redes e sistemas operacionais distribuídos. A diferença principal entre ambos reside no grau de transparência existente na utilização dos recursos do sistema.

3.1.1 Sistemas Operacionais para Redes

Um sistema operacional para redes é um sistema fracamente acoplado [TAN92] no qual os componentes do sistema em cada nó podem ser distintos [CRI88]. Um exemplo desse tipo de sistema consiste na interligação de estações de trabalho em uma rede local.

Dois aspectos principais são considerados em tais sistemas: a utilização de processadores remotos e a utilização de arquivos remotos.

A utilização de processador remoto deve ser feita explicitamente. O sistema deve fornecer recursos para que o usuário acesse remotamente um nó ao qual tenha direito e lá execute os processos que desejar. Comandos para acesso à máquina remota devem ser providos pelo sistema. Um exemplo de tais comandos é o *telnet* [SUN90]. Outra possibilidade é o sistema oferecer comandos para execução de processo em processador remoto. Nesse caso, o sistema é bem mais flexível e o usuário faz uso de tal comando especificando a máquina que deseja utilizar e o processo que deseja executar.

A utilização de arquivos remotos pode ser feita de duas formas principais. A cópia do arquivo remoto para o nó local, que deve ser feita através de comandos providos pelo sistema nos quais o usuário especifica o nome e a localização do arquivo desejado. Um exemplo desse tipo de comando é o *ftp* [IAB959]. A outra forma considera a existência de um sistema de arquivos global compartilhado em toda a rede. Nesse caso o sistema de arquivos é gerido pelo sistema de arquivos da rede. Um exemplo de tal tipo de sistema é o **NFS** (*Network File System*) da Sun Microsystems.

3.1.2 Sistemas Operacionais Distribuídos

Um sistema operacional distribuído é um sistema fortemente acoplado [TAN92], onde os componentes do sistema que executam em cada nó são os mesmos [CRI88]. A utilização de recursos remotos da rede é gerida pelo sistema, livrando o usuário de tais preocupações (o que não ocorre nos sistemas operacionais para redes).

A **transparência** na utilização de recursos é provavelmente o objetivo principal de um sistema distribuído [TAN92]. Além disso, o projeto de um sistema distribuído deve levar em conta alguns requisitos fundamentais. A **flexibilidade** do sistema de modo que ele seja facilmente adaptável a mudanças futuras. A **confiabilidade** tanto no que se refere à confidencialidade de dados como à sua integridade por ocasião de transferências, além da tolerância a falhas. O **desempenho** para que programas não sejam mais lentos em sistemas distribuídos do que em sistemas centralizados. A **escalabilidade** para suportar crescimento sem prejuízo da capacidade existente.

A alocação de processos a processadores que estejam menos carregados como forma de otimizar a utilização da máquina é uma facilidade desejável para um sistema distribuído. A migração de processos também pode ser um recurso útil, mas de alta complexidade em termos de implementação.

Um sistema distribuído deve apresentar mecanismos unificados de gerência de processos e de comunicação entre processos em todos os nós do sistema, assim como as chamadas de sistema disponíveis.

3.2 Comunicação entre Processos

Nos sistemas monoprocessadores ou mesmo em multiprocessadores, a comunicação entre processos está baseada no uso da memória compartilhada. Já em sistemas multicomputadores, mensagens têm que ser efetivamente transferidas entre áreas de memórias uma vez que processos comunicantes podem estar executando em nós diferentes. Tais sistemas fazem uso de protocolos de comunicação organizados em camadas como OSI e TCP/IP, utilizados em redes de longa distância, ou protocolos mais simplificados aplicados a redes locais [TAN88]. Tais protocolos têm como objetivo somente a transferência de mensagens. Eles são bem aplicados a redes nas quais os nós muitas vezes executam sistemas operacionais diferentes e há pequena interação entre processos de nós distintos. Já em um sistema distribuído real este grau de interação é mais alto, tornando-se interessante a existência de um sistema de comunicação, que leve em consideração a estruturação do sistema e tenha um menor *overhead* do que aquele dos protocolos com muitas camadas.

Para o modelo distribuído é conveniente organizar-se o sistema como uma coleção de processos cooperantes (**servidores**) que oferecem serviços aos processos usuários (**clientes**). Todos os nós do sistema executam o mesmo núcleo podendo haver processos servidores e clientes em cada um dependendo da implementação. O sistema de comunicação é o mais simplificado possível sendo, normalmente, um protocolo do tipo requisição/resposta. Clientes enviam mensagens contendo requisições de serviço a servidores que os realizam e enviam aos clientes mensagens de resposta. A resposta é considerada como confirmação de recepção da mensagem de requisição.

Troca de mensagem é a forma de comunicação predominante em sistemas operacionais distribuídos, os quais normalmente estão estruturados segundo o modelo cliente-servidor.

3.2.1 Troca de Mensagem

No mecanismo de comunicação por troca de mensagens o processo remetente executa a primitiva para enviar mensagem no qual identifica o destinatário e o processo

destinatário executa um comando para receber mensagem no qual identifica o remetente. O núcleo do sistema é responsável por transferir a mensagem entre os processos envolvidos.

A troca de mensagem pode ser implementada de várias formas, havendo alguns fatores a serem considerados no seu projeto, objetivando atender os requisitos desejados para um sistema operacional distribuído, como foi mencionado na seção 3.1.2. Tais fatores são [TAN92]: a forma de identificar os processos no sistema, comunicação direta ou indireta, primitivas de comunicação síncrona ou assíncrona, armazenamento intermediário de mensagem ou não, esquemas com confirmação ou não.

A **identificação de processos** no sistema deve ser a mais flexível possível levando em consideração o pré-requisito de unicidade no sistema. Pode haver esquemas de idenficação por máquina que associe a identificação do processo à máquina onde ele está executando. Assim, mensagens podem ser endereçadas diretamente à máquina correta e o seu núcleo a capta e transmite ao processo. Outra abordagem é ter-se uma identificação global de processos sem levar em consideração a máquina onde ele está executando. Neste caso, pode-se adotar um esquema de difusão da mensagem e o núcleo de cada máquina verifica se a mensagem é destinada a algum processo local; o núcleo da máquina onde o processo destinatário está em execução capta a mensagem e a transmite ao processo.

A comunicação **direta** [SIL91] ocorre quando o processo designa explicitamente o processo que é seu parceiro na troca de mensagem. A comunicação **indireta** ocorre quando o mecanismo de troca de mensagem faz uso de elementos intermediários chamados caixas postais. O emissor deposita suas mensagens na caixa postal do receptor, de onde ele as retira. A comunicação indireta é um instrumento poderoso para conferir ao sistema transparência de localização de processos.

Uma primitiva de comunicação **síncrona** causa o bloqueio do processo que a executa exigindo a participação simultânea dos processos envolvidos na comunicação. Uma primitiva **assíncrona**, por sua vez, permite que o processo remetente não fique obrigatoriamente bloqueado, podendo realizar outras tarefas até que a mensagem seja enviada ou recebida. Nesse caso, o núcleo deve prover o **armazenamento intermediário** de mensagem em *buffers* por ele geridos.

Esquemas com confirmação conferem confiabilidade à comunicação. Mensagens com requisição de serviço podem ser consideradas como recebidas com o retorno da resposta ao serviço ou pela emissão de uma mensagem específica de confirmação. Um temporizador deve ser utilizado pelo cliente para que ele não fique indefinidamente esperando por uma resposta a uma requisição que eventualmente não possa ser atendida, por exemplo quando a máquina do servidor falha por ocasião da realização do serviço.

3.2.2 Chamada de Procedimento Remoto

A estruturação de um sistema operacional distribuído, segundo o modelo cliente-servidor fazendo uso do mecanismo de troca de mensagem, apresenta o desconforto do uso explícito das operações de envio e recepção de mensagens [TAN92]. Esta não é a preocupação principal em um sistema centralizado e também não deve ser em um sistema operacional distribuído que pretende passar a seus usuários a visão de um sistema centralizado.

Uma solução para tal problema é a **Chamada de Procedimento Remoto** [BIR84]. A idéia fundamental desse mecanismo é permitir que programas ativem procedimentos localizados em outras máquinas como se fossem locais.

A chamada de procedimento remoto equivale a uma chamada de procedimento local, diferindo no fato de que a chamada é estendida de modo a prover a transferência de controle e dados através de uma rede de comunicação [BIR84]. Ao chamar um procedimento remoto, o processo chamador é suspenso, os parâmetros são passados através da rede de comunicação para o ambiente remoto onde o procedimento é ativado. Ao terminar sua execução, resultados são transmitidos de volta ao processo ativador e este continua sua execução. A implementação de chamadas de procedimentos remotos faz-se com a inserção de procedimentos intermediários (*stubs*) entre os processos envolvidos e a rede de comunicação. Tais procedimentos intermediários têm a função de esconder dos processos envolvidos a troca de mensagens ocorrida na comunicação real. Os componentes envolvidos numa chamada de procedimento remoto são: cliente, servidor, seus respectivos *stubs* e a rede de comunicação. O funcionamento deste esquema está ilustrado na figura 3-1. O processo cliente realiza uma chamada ao seu *stub* (1) que a transforma numa mensagem que será transmitida (2) através da

rede à máquina onde se encontra o servidor. A mensagem é recebida no destino pelo *stub* do servidor que a transforma numa chamada de procedimento local ao processo servidor (3). O procedimento do servidor é então executado normalmente e retorna a resposta ao serviço (4). Os resultados do retorno normal são transformados pelo *stub* do servidor em uma mensagem que é transmitida à máquina do processo cliente (5). Nesta máquina, o *stub* do processo transforma esta mensagem em um retorno normal de procedimento ao processo cliente (6).

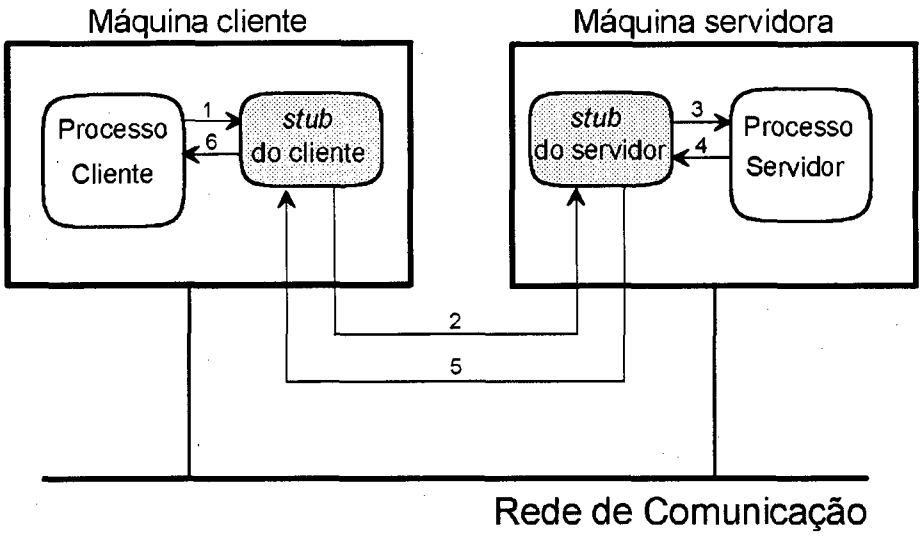

Figura 3-1. Chamada de procedimento remoto

3.3 Alguns Sistemas Distribuídos

O estudo de alguns sistemas operacionais para ambientes com múltiplos processadores, com ênfase nos esquemas de comunicação e provimento de serviços, contribuiu de forma direta ou indireta para as propostas que apresentamos nos capítulo 5 e 6.

Alguns sistemas operacionais modernos suportam o conceito de fluxos de execução (*threads*) [SIL91] e [TAN92]. Fluxos de execução podem ser chamados mini-processos que executam compartilhando o mesmo espaço de endereçamento de um processo. Um fluxo de execução tem seu próprio ponteiro de instrução, sua própria pilha de execução e é escalonado independentemente dos demais fluxos do mesmo processo. A utilização de fluxos de execução

não está associada diretamente a sistemas distribuídos, embora o conceito seja adequado a tais sistemas.

Mach

O núcleo do Mach [YOU87] e [TAN92] foi construído objetivando servir de base para a emulação de outros sistemas operacionais.

Processos no Mach são chamados de tarefas (*tasks*) e são constituídos por um ou mais fluxos de execução (*threads*). Cada tarefa tem controle sobre seus fluxos de modo a designar em quais conjuntos de processadores devem executar.

O núcleo suporta diferentes formas de comunicação de modo flexível e confiável. O sistema permite troca de mensagem assíncrona e chamada de procedimento remoto. A comunicação indireta está baseada no uso de estruturas de dados do núcleo que são caixas postais (*ports*) protegidas por capacidades (*capabilities*). O acesso às caixas postais está restrito a tarefas ou fluxos de execução que tenham autorização para fazê-lo. O fluxo de execução emissor copia a mensagem para a caixa postal que é então lida pelo receptor. A mensagem de resposta enviada por este deve ser passada através de outra caixa postal.

O núcleo mantém para cada fluxo de execução uma tabela de todas as caixas postais a que ele tem acesso. A referência a uma caixa postal é feita pela sua entrada nesta tabela que contém a **capacidade** do fluxo de execução sobre ela. A capacidade determina os direitos que o fluxo tem sobre a caixa. Os direitos referem-se à forma de acesso que pode ser de **receber**, de **enviar** ou de **enviar-uma-vez**. O portador do direito de **receber** é único para cada caixa postal e este fato determina a unidirecionalidade da comunicação sendo este direito transferível. Vários fluxos podem ter o direito de **enviar** para uma determinada caixa postal. O direito de **enviar-uma-vez** confere ao seu portador a possibilidade de enviar uma única mensagem àquela caixa.

Amoeba

O Amoeba [TAN92] é um sistema operacional projetado para uma máquina cuja arquitetura tem duas peculiaridades: grande número de processadores e cada um deles dispõe de dezenas de *megabytes* de memória. A arquitetura está baseada em um modelo no qual a capacidade computacional está localizada em um ou mais agrupamentos de processadores (*processor pool*) que tem seu uso compartilhado por todos os usuários. Os agrupamentos de processadores são constituídos por um número substancial de processadores com memórias próprias. Estes processadores podem ser de arquiteturas diferentes.

O sistema operacional é composto por um micronúcleo e uma coleção de processos servidores. O micronúcleo executa em todas as máquinas do sistema enquanto os servidores podem executar em um ou mais agrupamentos de processadores servindo a grupos de usuários. Assim como o Mach, o Amoeba suporta múltiplos fluxos de execução compartilhando o mesmo espaço de endereçamento de um processo.

Todos os recursos do sistema são considerados objetos sobre os quais determinadas operações podem ser efetuadas. Tais recursos do sistema são geridos pelos processos servidores.

A comunicação provida pelo núcleo suporta chamada de procedimento remoto e comunicação de grupo. Ela está baseada no uso de caixas postais (*ports*) que são endereços lógicos dos seus processos proprietários. Ao requisitar um serviço a um servidor qualquer, o processo cliente reporta-se ao servidor através da sua caixa postal e fica bloqueado até a chegada da mensagem de resposta. Um grupo no Amoeba é uma reunião de processos que cooperam entre si para a realização de determinada tarefa ou provimento de determinado serviço. Grupos no Amoeba são fechados. Assim, um processo cliente comunica-se com o grupo através de uma chamada de procedimento remoto ao seu membro coordenador. Este, por sua vez, comunica-se com os demais membros do grupo para definir as tarefas de cada um, se necessário.

Accent

O Accent é um sistema operacional orientado à comunicação, uma vez que seu princípio organizacional básico é a abstração de comunicação entre processos. Ele provê transparência de utilização de recursos da rede. O núcleo do sistema executa em todos os nós da rede sendo responsável pela gerência de processos e comunicação.

O sistema utiliza o conceito de caixa postal para a transferência de mensagens. Há uma chamada de sistema para a alocação de caixa postal e uma outra para liberação. Ao alocar uma caixa postal, um processo torna-se seu proprietário, sendo o único processo com direito de leitura das mensagens enviadas a ela. O direito de leitura é exclusivo podendo ser transferido a outro processo. Uma caixa pode também ser liberada por seu proprietário se ele não tiver transferido o direito de leitura. A comunicação é assíncrona e assim os processos clientes não são automaticamente bloqueados ao enviar mensagem a um servidor.

Roscoe

O Roscoe [SOL79] é um sistema operacional distribuído projetado para uma rede de microcomputadores objetivando prover aos usuários a aparência de estarem utilizando uma máquina virtual única. Os microcomputadores da rede têm processadores homogêneos. O núcleo executa em todos os nós do sistema e provê serviços que são requisitados através de chamadas específicas. Outros serviços do sistema são providos por processos utilitários. Há rotinas de bibliotecas para facilitar a comunicação entre os processos clientes e os utilitários.

A comunicação entre processos está baseada em dois conceitos: ligações (*links*) e mensagens. Uma ligação é uma conexão lógica unidirecional entre dois processos. Um deles envia e o outro recebe mensagens através da ligação. Um processo pode ter várias ligações identificadas por números inteiros que são índices de uma tabela de ligações do processo mantida pelo núcleo. Processo emissor não precisa ter conhecimento da localização do proprietário da ligação.

Wisdom

O Wisdom [AUS91] é um sistema operacional para uma máquina formada por uma rede de processadores ligados ponto-a-ponto estaticamente. Trata-se de um multicomputador chaveado do tipo grelha no qual os nós são microprocessadores Transputers da INMOS. O Wisdom é um sistema operacional paralelo, o que significa permitir a execução de programas como uma coleção de partes (tarefas) que executam paralelamente comunicando-se por troca de mensagens com transparência de localização.

Um processador é dividido em processadores virtuais que recebem uma fração de memória do processador real que é multiplexado entre os processadores virtuais que hospeda. Todos os transputers da máquina executam o núcleo do sistema. O núcleo serve de base para um sistema paralelo sendo composto de quatro módulos: módulo de roteamento, módulo escalonador, módulo de balaceamento de carga e módulo de nomeação.

O módulo de **roteamento** é responsável pelo transporte de mensagens entre tarefas que executam no mesmo ou em transputers distintos. A identificação de tarefa inclui a coordenada cartesiana na qual a tarefa está executando, que é fixa durante sua existência uma vez que não há migração de tarefas. O módulo de roteamento verifica quais transputers vizinhos estão mais próximos do destinatário. A mensagem é enviada para um deles assim que a primeira ligação estiver livre. O módulo **escalonador** é reponsável por criar e gerir os processadores virtuais de um processador real. Assim, faz o escalonamento de processadores virtuais em um processador real utilizando política de tempo compartilhado. O módulo de **balanceamento de carga** decide sobre a alocação de novas tarefas no sistema. Ele é simplificado pelo pressuposto de que uma tarefa vai querer mais provavelmente comunicar-se com a tarefa que a criou, sendo desta forma os transputers vizinhos ao da tarefa pai os hospedeiros potenciais da nova tarefa. O módulo de **nomeação** tem a função de estabelecer canais de comunicação entre tarefas que trocam mensagens.

4 O SISTEMA MINIX E O MULTICOMPUTADOR Nó//

Neste capitulo, nós introduzimos o sistema operacional MINIX, escolhido como plataforma de experimentação para a realização de nossas propostas de um mecanismo de comunicação e de um método de ativação de servidores por demanda. Também apresentamos um estudo preliminar visando a adaptação do sistema MINIX original à arquitetura do multicomputador Nó//. Grande parte das soluções visualizadas nesse estudo serão utilizadas na alteração do MINIX para o Nó//, conforme propostas que serão descritas nos capítulos 5 e 6 sobre o Nó//.

4.1 O sistema Operacional MINIX

O sistema operacional UNIX tinha seu código aberto, sob licença de seu fabricante, até a chegada ao mercado de sua versão 7, que veio acompanhada da indisponibilidade de código fonte. Com isso, o meio acadêmico perdeu uma poderosa ferramenta de estudo e trabalho e surgiu espaço para o desenvolvimento de recursos substitutos.

Tanembaum, da universidade *Vrije* de Amsterdã (Holanda), produziu, então, o sistema MINIX com o objetivo de suprir tal lacuna. O MINIX, cujo nome deriva de mini-UNIX, é totalmente compatível com o UNIX versão 7 do ponto de vista do usuário, mas apresenta estrutura interna bem distinta. Ele apresenta uma vantagem em termos acadêmicos, pois foi escrito objetivando ser legível enquanto o UNIX prioriza eficiência, uma vez que é um produto comercial.

4.1.1 Estrutura Interna

A estrutura interna do MINIX reflete a tendência atual de minimização do núcleo dos sistemas operacionais, obtida com a descentralização de suas funções, que passam a ser distribuídas em processos do sistema, externos ao núcleo. Tal modelo prevê a existência de um núcleo com um número de funções reduzido e a transferência da maior parte de suas funções tradicionais para processos servidores. Essas funções estão principalmente relacionadas com a

gerência de recursos do sistema. O núcleo permanece com as obrigações essenciais de gerência de processos e de comunicações entre eles, efetivada por troca de mensagens. Essa forma de estruturação caracteriza o modelo cliente-servidor. Nesse modelo, processos usuários fazem uso dos recursos do sistema através de processos servidores. Essa estruturação é adequada tanto a sistemas monoprocessadores como a sistemas distribuídos.

O MINIX é estruturado funcionalmente em conjuntos de componentes dispostos em quatro camadas, como esboçado na figura 4-1.

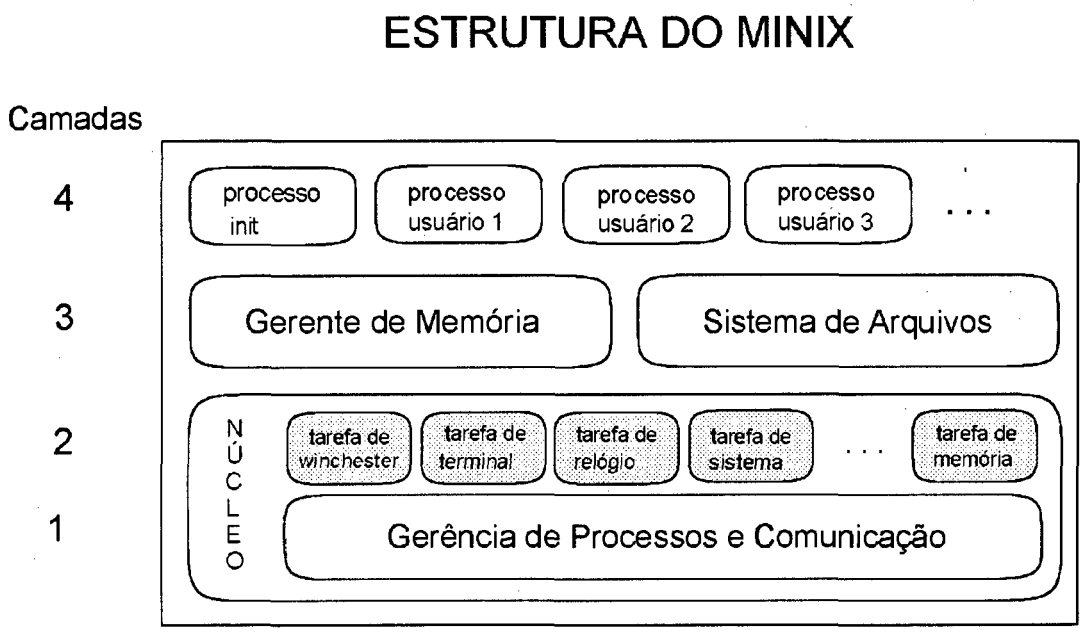

Figura 4-1. Estrutura do sistema operacional MINIX

A camada 1 é responsável pela gerência de processos e pela comunicação entre eles. Essa camada provê aos processos das camadas superiores um mecanismo de comunicação baseado na troca síncrona direta de mensagens.

A camada 2 é composta pelos controladores de periféricos responsáveis pelas operações de entrada e saída. Estes controladores são implementados como processos, um para cada tipo de dispositivo, chamados tarefas (*tasks*) de entrada e saída. Há um processo especial nesta camada que não trata de nenhum dispositivo específico mas é funcionalmente equivalente às

tarefas. Ele é chamado tarefa do sistema (*system task*) que existe para realizar a comunicação entre os processos servidores presentes na camada três e o núcleo do sistema.

A camada 3 é formada por dois processos que provêem serviços aos processos usuários. São eles o gerente de memória (*Memory Manager* - MM) e o sistema de arquivos (*File System* - FS).

Por fim, a camada 4 é composta por um processo especial chamado *init* e por todos os processos usuários como compiladores, editores, interpretadores de comandos além dos processos usuários do sistema.

O sistema pode ser visto (figura 4-2) segundo o modelo cliente-servidor sendo formado por uma coleção de processos clientes e servidores que se comunicam através de troca de mensagens.

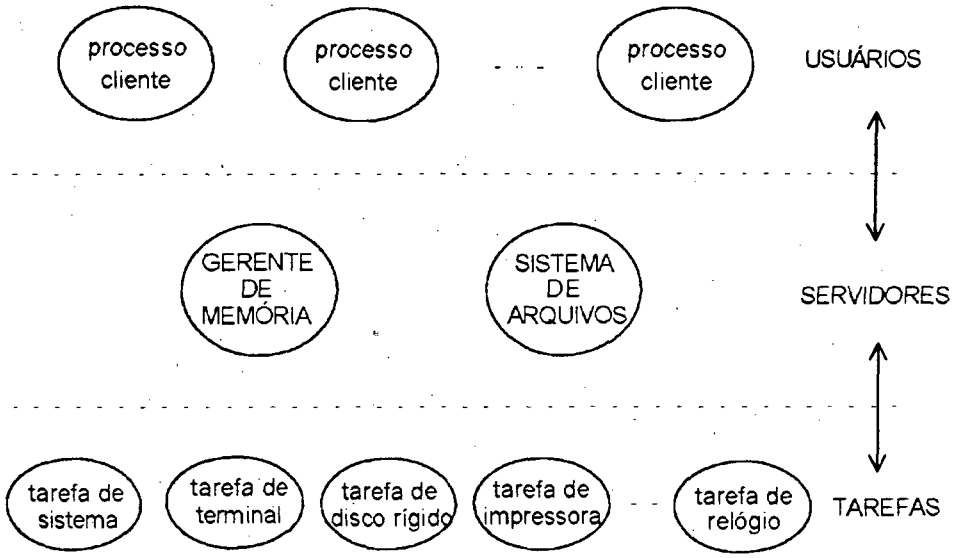

Figura 4-2. Estrutura do MINIX como uma coleção de processos comunicantes

4.1.2 Processos

O procedimento de carga do MINIX é iniciado com a transferência para memória e execução de um programa de inicialização (**bootstrap**) contido no primeiro setor da primeira

trilha do disco de carga do sistema. Esse programa carrega em memória todo o sistema operacional que desabilita as interrupções, inicializa alguns registradores e chama a rotina principal do núcleo. Essa rotina faz a inicialização da tabela de processos e ativa os processos referentes às tarefas de entrada e saída, gerente de memória, sistema de arquivos e um processo especial chamado *init* colocando-os nas suas respectivas filas de escalonamento. As tarefas de entrada e saída, o gerente de memória e o sistema de arquivos ficam bloqueados à espera de requisição de serviço.

Processos no MINIX são criados dinamicamente através da chamada de sistema *fork*. Cada novo processo pode, por sua vez, criar outros processos produzindo assim uma hierarquia de processos no sistema. O processo *init* coloca o sistema à disposição dos usuários, criando para cada terminal um processo de conexão (*login*) que permite o acesso de um usuário ao sistema. A função do *login* é ler a identificação e eventualmente a senha (*password*) de um usuário para verificar a validade dessas informações no arquivo etc/passwd que contém um cadastro de todos os usuários do sistema. Quando um usuário é bem sucedido na tentativa de conexão, o processo *login* do terminal em uso executa um programa interpretador de comandos (*shell*) que recepta os diversos comandos do usuário gerando, na maioria das vezes, um novo processo para cada comando. Dessa forma, obtém-se a árvore de processos da figura 4-3. Os processos *login* são filhos do processo *init*. Os comandos são executados como processos filhos do *shell* que por sua vez podem criar outros processos filhos, estendendo a árvore.

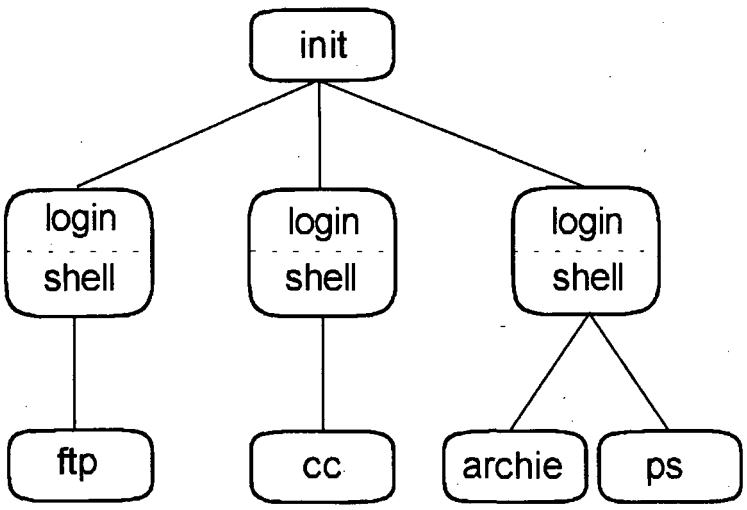

Figura 4-3. Árvore de processos usuários no MINIX

4.1.3 Núcleo

A camada 1 do sistema é o núcleo propriamente dito. Suas funções estão ligadas ao tratamento de interrupções, escalonamento de processos e a gerência de comunicações. O núcleo do sistema MINIX segue a tendência moderna de redução do número de funções.

Os programas da camada 2 são compilados conjuntamente com os da camada 1, formando um único programa objeto, para facilitar a portabilidade do sistema para máquinas que operam nos modos supervisor e usuário, onde só no modo supervisor é permitido realizar operações de entrada e saída. Apesar disso, as tarefas são processos completamente independentes uns dos outros e comunicam-se através de troca de mensagens.

4.1.3.1 Tratamento de Interrupções

As interrupções são captadas por rotinas do núcleo, sendo cada tipo de interrupção atendida por uma rotina específica. Tais rotinas apresentam uma estrutura de execução genérica que consiste, inicialmente, na suspensão do processo corrente e chaveamento de contexto para o núcleo. A partir de então, uma mensagem é composta contendo a informação causadora da interrupção e o procedimento *interrupt* é ativado para transmití-la á tarefa que fará seu tratamento. Após o tratamento, o controle retorna para a rotina captadora da interrupção que ativa o procedimento *restart* responsável por escalonar um novo processo.

Há duas formas distintas de composição da mensagem de interrupção. Em alguns casos ela é composta no próprio procedimento captador. Em outros, há necessidade de algum processamento adicional para compô-la. Esse processamento é feito em rotinas de tarefas que são ativadas antes da chamada de *interrupt*.

A função do procedimento *interrupt* é enviar a mensagem recém composta para a tarefa adequada. Além disso, ele reabilita o controlador de interrupções para permitir o tratamento de novas interrupções.

4.1.3.2 Comunicação

O MINIX usa o método de comunicação síncrona direta sem armazenamento intermediário de mensagem. Todos os processos do sistema comunicam-se através desse mecanismo. As mensagens são de tamanho fixo, podendo variar de acordo com a arquitetura utilizada.

O mecanismo de troca de mensagens é realizado por três operações primitivas, disponíveis através de três procedimentos especiais que fazem parte das bibliotecas do MINIX. Tais primitivas são chamadas **send**, **receive** e **sendrec**, tendo os formatos a seguir especificados:

send(*dest*, *&message*);

receive(*source*, *&message*);

sendrec(*src_dst*, *&message*);

A primitiva **send** tem como parâmetros a identificação do destinatário (*dest*) e o endereço da mensagem (*&message*) e tem a função de enviar a mensagem cujo endereço é *&message* para o processo *dest*.

A primitiva **receive** tem como parâmetros a identificação do processo emissor (*source*) e o endereço da mensagem (*&message*) e tem a função de receber do processo *source* uma mensagem que será armazenada no endereço *&message*.

A primitiva **sendrec** é uma dupla operação especificando como parâmetros a identificação do processo parceiro da comunicação (*src_dest*) e o endereço da mensagem (*&message*). Esse último parâmetro refere-se também ao endereço no qual a mensagem de retorno deverá ser armazenada, havendo então sobreposição da mensagem enviada.

As três operações bloqueiam o seu chamador caso o processo correspondente não esteja apto a participar da comunicação. O desbloqueio do processo só ocorrerá quando o correspondente se tornar apto a realizar a troca síncrona de mensagem.

O procedimento que implementa as operações primitivas faz uso dos registradores **cx**, **bx** e **ax** para armazenar respectivamente a **operação** a ser realizada (SEND, RECEIVE ou

SENDREC), o **endereço da mensagem** e a **identificação do processo** remetente ou destinatário de mensagem. Feita a atualização dos registradores, é então executada uma interrupção de software (SYS_VECTOR = 32), direcionando o fluxo de execução para a rotina *s_call*. Ela faz o tratamento da interrupção como já mencionamos, ou seja, bloqueia o processo chamador da primitiva (processo em execução) transferindo o controle para o núcleo, compõe uma mensagem com os registradores ax, bx e cx, e transfere controle para a rotina *sys_call*. A rotina *sys_call* verifica a validade dos parâmetros recebidos e executa a rotina *mini_send* para atender à operação primitiva **send**, *mini_rec* para atender a um **receive** ou ambas para atender à primitiva **sendrec**.

As rotinas *mini_send* e *mini_rec* gerem no núcleo as trocas de mensagens. A rotina *mini_send* recebe como parâmetros: o apontador do processo emissor da mensagem (caller_ptr), o destinatário (dest) e o endereço da mensagem a ser enviada (m_ptr). Sua execução consiste inicialmente em verificar a validade do destinatário, uma vez que processos usuários só podem enviar mensagem para servidores (gerente de memória e sistema de arquivos). Em seguida, é verificado se a mensagem cabe no segmento de dados do destinatário. Neste ponto exite um teste para prevenção de *deadlock* que consiste em verificar se o destinatário está bloqueado na execução de um **send** para o processo emissor. Caso esteja, a operação **send** corrente é suspensa e um aviso é devolvido ao seu chamador. Depois disso, verifica o estado do destinatário que pode estar bloqueado na execução de um **receive** de qualquer processo (ANY) ou desse especificamente. Caso esteja, a rotina *cp_mess* do programa usr/src/kernel/klib.s é ativada para realizar a efetiva cópia física da mensagem entre as áreas de memória de ambos os processos e o destinatário é reescalonado (desbloqueado) pela rotina *ready*. Caso contrário, o emissor é bloqueado via rotina *unready*, o processo é marcado como desejando enviar mensagem, e a identificação registrada na fila de processos que desejam enviar mensagens para aquele destinatário.

A rotina *mini_rec* recebe também três parâmetros: seu chamador (caller_ptr) que deseja receber mensagem, o processo remetente do qual ele deseja receber tal mensagem (src) e m_ptr que é o endereço de sua área de memória onde a mensagem recebida deverá ser gravada. O segundo parâmetro refere-se a um processo específico ou a um processo qualquer que é especificado pela utilização da constante ANY. Sua execução consiste em verficar se o processo especificado como fonte está na fila de processos que desejam enviar mensagem para o receptor.

Esse processo pode ser aquele especificado como parâmetro ou o primeiro da fila se o parâmetro for ANY. Caso haja, a mensagem é copiada pela rotina *cp_mess* da área de memória do emissor para a do receptor. Caso não haja, o processo receptor é bloqueado via rotina *unready* sendo salvos o endereço do *buffer* de mensagem (m_ptr) e a identificação do processo fonte (src) na sua entrada na tabela de processos. O último passo de *mini_rec* é verificar se há sinais gerados pelo núcleo (SIGINT, SIGQUIT, SIGALRM) para serem tratados. Em caso positivo, o procedimento *inform* é chamado, o qual envia uma mensagem para o gerente de memória comunicando o sinal.

4.1.3.3 Escalonamento

O MINIX implementa um algoritmo de escalonamento baseado em um conjunto de 3 filas de prioridades distintas. O escalonador mantém as filas de processos aptos como mostra a figura 4-4, sendo uma para os processos de cada camada. Há dois vetores, *rdy_head* e *rdy_tail*, que são compostos por três ponteiros cada. Estes ponteiros apontam respectivamente para os processos das cabeças e caudas de cada fila. Ao tornar-se apto, um processo é colocado na cauda da fila de processos do seu grupo. Os processos controladores de periféricos (camada 2) formam a fila de maior prioridade (prioridade 0), os processos servidores (camada 3) formam a fila de prioridade intermediária (prioridade 1) e os processos usuários (camada 4) formam a fila de menor prioridade (prioridade 2). De acordo com o esquema mencionado, um processo servidor só ocupará o processador quando não houverem tarefas de entrada e saída aptas e processos usuários, por sua vez, somente quando não houverem processos servidores nem tarefas de entrada e saída aptas. A política de primeiro a chegar, primeiro a ser atendido (*First Come First Served* - FCFS) é aplicada às filas das tarefas e dos servidores. A política de *round-robin* é aplicada à fila dos processos usuários, com fatia de tempo de 100 ms.

Os procedimentos *ready* e *unready* do escalonador, que têm como parâmetro o endereço de um processo, são usados respectivamente para inserir e retirar processos da sua respectiva fila de escalonamento. O procedimento *pick_proc* é responsável pela escolha do próximo processo a ocupar o processador.

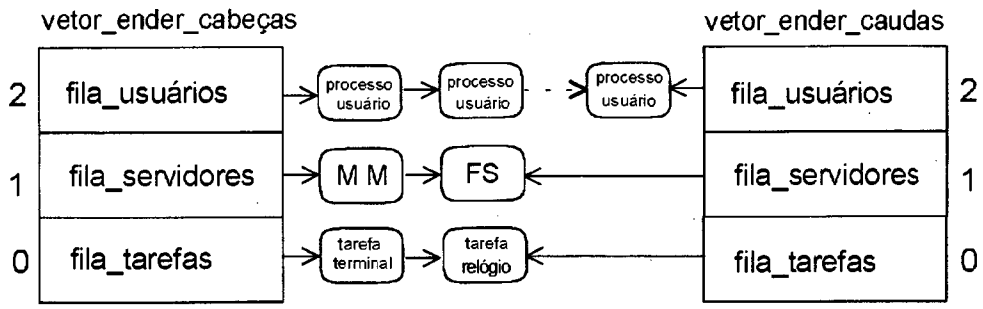

Figura 4-4. Filas de Escalonamento do MINIX

4.1.4 Tarefas de Entrada e Saída

Cada classe de dispositivo periférico de entrada e saída gerida pelo MINIX dispõe de um processo tratador específico chamado de tarefa. As tarefas comunicam-se com os servidores do sistema (processos da camada 3) usando o mesmo mecanismo de troca de mensagens dos demais processos do sistema. O que distingue as tarefas dos demais processos é o fato de serem ligadas conjuntamente com os programas da camada 1 formando um único programa objeto acessando o mesmo espaço de endereçamento. Essa composição é feita para facilitar a portabilidade do MINIX para máquinas que apresentem modos supervisor e usuário onde operações de entrada e saída só são permitidas em modo supervisor. O fato das tarefas serem processos independentes que se comunicam com os demais processos do sistema por troca de mensagens é uma diferença marcante do UNIX no qual os tratadores de dispositivos são procedimentos do núcleo [TAN87].

Todas as tarefas no MINIX são estruturadas de forma similar, segundo o modelo de um servidor, sendo formadas por um programa principal e vários procedimentos correspondentes aos seus diversos serviços.

4.1.5 Servidores

A camada 3 do MINIX é composta pelos processos servidores do sistema: o gerente de memória e o sistema de arquivos. Cada servidor é um processo seqüencial de comportamento semelhante ao das tarefas: eles esperam por mensagens contendo requisição de

serviço, executam o procedimento responsável pelo serviço solicitado, e devolvem mensagens contendo resposta ao serviço.

Os servidores utilizam formatos de mensagens específicos que devem ser conhecidos e usados por seus clientes para que a interação seja correta. Os processos servidores da camada 3 têm como clientes os processos usuários da camada 4 e são, por sua vez, clientes dos processos da camada 2 (tarefas). O mecanismo de comunicação é o mesmo para todos os processos. As trocas de mensagens, no entanto, estão restritas a processos pertencentes a camadas adjacentes.

4.1.5.1 Gerente de Memória

A existência do gerente de memória como processo servidor independente do núcleo do sistema permite maior flexibilidade no que se refere a eventuais mudanças na política de gerência de memória. Alterações de tais políticas são feitas no servidor sem afetar a tarefa de sistema (*system task*) responsável pela execução do mecanismo de instanciação de mapas de memória para cada processo.

O MINIX adota uma política simples para gerência de memória. A memória é gerida através da utilização de uma lista que contém todos os espaços livres de memória classificados em ordem crescente de endereços que são alocados e liberados à medida que processos são criados e finalizados. A simplicidade desse esquema está presa a três fatores: a idéia do sistema ser feito para computadores pessoais e não para grandes sistemas de tempo compartilhado, ou seja, o sistema lidará com um pequeno número de processos em execução; o desejo de fazer o sistema funcionar em um microcomputador IBM-PC (processador 8088) que não possui mecanismos de proteção de memória; e a vontade de ver o sistema facilmente portável para outros computadores pessoais sem precisar fazer grandes modificações para adequá-lo a cada um deles.

Há três situações que causam mudanças nos espaços livres de memória. Uma modificação ocorre quando da execução da chamada de sistema *fork*, pela alocação de memória ao processo filho o qual recebe uma cópia da memória do processo pai. A outra situação ocorre

quando um processo substitui sua imagem de memória por um outro programa através da chamada de sistema *exec*. Neste caso, a área alocada para a antiga imagem é retornada para a lista de espaços livres e uma nova área é alocada para a nova imagem. Ainda outra situação consiste na liberação de área de memória sempre que um processo termina, seja pela sua finalização normal ou por ter sido destruído por um sinal.

4.1.5.2 Sistema de Arquivos

O sistema de arquivos do MINIX, assim como o gerente de memória, é implementado como um servidor independente do núcleo do sistema. Ele é o maior, o mais estável e o mais independente da máquina entre os componentes do MINIX. Um sistema de arquivo MINIX é uma entidade lógica com índices de nodos de arquivos (i-nodos), diretórios e blocos de dados que pode estar armazenado em algum dispositivo de bloco, como um disco flexível ou parte de um disco rígido. Ele é composto de um **bloco de carga**, de um **super bloco**, de um **mapa de bits de i-nodos**, de um **mapa de bits de zonas**, dos **i-nodos** e dos **blocos de dados**. A figura 4-5 apresenta o esboço de um sistema de arquivo MINIX para um disco flexível de 360K.

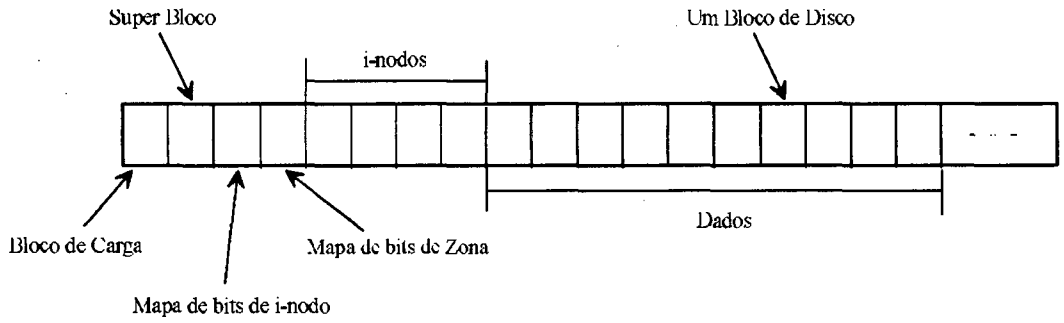

Figura 4-5. Um sistema de arquivo para um disco flexível de 360 K

O **bloco de carga** é lido para a memória quando o sistema é inicializado e depois não é mais utilizado. Nem todo *drive* de disco pode ser utilizado para carga do sistema, mas todos têm o bloco de carga por questão de uniformidade. O **super bloco** é composto de informações que descrevem a disposição dos componentes de um sistema de arquivo. O mapas de *bits* guardam informação sobre os i-nodos e as zonas do sistema de arquivo que estão livres. O i-nodo

de um arquivo existe de forma estática no disco e, entre outras informações, guarda as identificações dos blocos do disco que contêm os dados do arquivo. O i-nodo de um arquivo é carregado em memória na tabela de i-nodos quando o arquivo é aberto, sendo ali mantido até o seu fechamento.

O sistema de arquivos mantém três estruturas de dados para gerir o acesso aos arquivos do sistema: a tabela de processos *fproc*, a tabela *filp* e a tabela de i-nodos em memória. Essas três tabelas são mostradas na figura 4-6 onde esboçamos também a relação entre elas.

A tabela *fproc* representa a parte da tabela de processos mantida pelo sistema de arquivos. Ela contém, para cada processo, entre outros campos, um vetor de descritores dos arquivos utilizados. Cada descritor contém o índice de uma entrada na tabela *filp* correspondente a um determinado arquivo.

A tabela *filp* é uma tabela única compartilhada pelos processos onde cada entrada corresponde a um arquivo em uso no sistema. Entre as informações mantidas nessa tabela, para cada arquivo, estão o índice de uma entrada para a tabela de i-nodos, um contador do número de usuários do arquivo e a posição no arquivo que indica o próximo *byte* a ser lido ou gravado.

A tabela de i-nodos em memória possui uma entrada para cada arquivo aberto no sistema. Cada entrada nessa tabela corresponderá ao i-nodo de um arquivo no momento da sua abertura, a partir das informações contidas em disco. Se um mesmo arquivo for aberto mais de uma vez, haverá somente uma entrada para seu i-nodo nessa tabela.

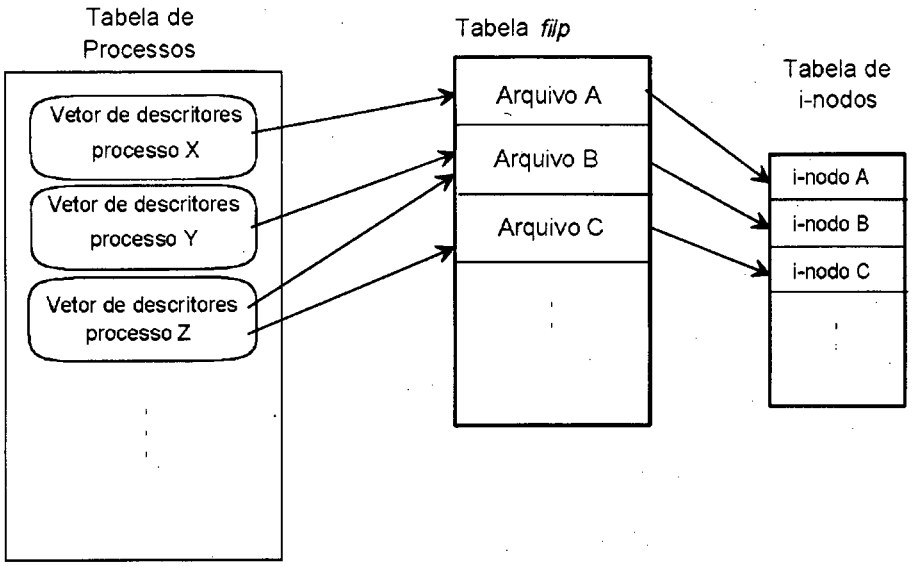

Figura 4-6. Relação entre estruturas utilizadas para acesso a arquivos

O acesso a um arquivo por um processo ocorre com a utilização destas três estruturas da seguinte forma. O processo usuário recebe o descritor de arquivo como retorno da chamada de sistema *open* e, a partir de então, utiliza este descritor para referir-se ao arquivo. O descritor de arquivo recebido pelo processo é o índice de entrada no vetor de descritores de arquivo da sua tabela *fproc*. Esta entrada possui um ponteiro para a tabela *filp* que contém o índice de entrada da tabela de i-nodos em memória para o arquivo especificado. Se um arquivo é compartilhado por mais de um processo usuário, cada um deles compartilha a mesma entrada na tabela *filp* como ilustrado na figura 4-6. A entrada na tabela de i-nodos em memória contém o endereço utilizado para acessar os blocos de dados do arquivo.

4.2 A Realização do MINIX sobre o Nó//

A realização do sistema MINIX sobre o Nó// se apóia em dois modelos de implementação básicos: um para os processos e outro para as comunicações.

Os processos podem ser implementados de forma simples pela atribuição de apenas um processo por nó com o triplo objetivo de explorar o paralelismo real, de aumentar a disponibilidade dos processos individuais e de simplificar a gestão de recursos. A adoção do nó

como unidade de alocação corresponde a uma solução intermediária entre duas possibilidades extremas. Em alguns ambientes, a unidade de alocação pode ser o próprio multicomputador, quando então ele é usado para a execução de algoritmos específicos [REE87]. Em outros, a unidade de alocação pode corresponder às unidades de processos independentes multiprogramados sobre nós individuais como acontece, por exemplo, no sistema Helios [GAR87].

Para a implementação de processos segundo este modelo, o barramento de serviço pode ser utilizado na transmissão das solicitações de alocação e liberação de nós.

A arquitetura do Nó// permite que canais diretos sejam estabelecidos por demanda através da rede de interconexão dinâmica. O estabelecimento de conexões entre processos por demanda é um procedimento completamente geral uma vez que a duração dos canais (até sua desconexão, também por demanda) pode variar desde o tempo necessário para o transporte de uma única mensagem até a execução completa de um programa paralelo.

Para a realização desse mecanismo, o barramento de serviço deve ser utilizado na transmissão das solicitações de conexões e desconexões de canais.

As chamadas de sistema são realizadas em MINIX por dois processos servidores: o sistema de arquivos e o gerente de memória. O sistema de arquivos é o elemento mais estável do MINIX por ser funcionalmente autônomo e largamente independente da máquina. Ele realiza uma coleção coerente de chamadas de sistema que pode ser harmoniosamente gerida por um componente único. Por outro lado, o gerente de memória do MINIX é extremamente dependente da máquina. As chamadas de sistema que se referem à criação e destruição de processos exigem a atenção dos dois servidores porque elas afetam o estado global do sistema. A maioria do trabalho é realizado pelo gerente de memória mas é necessário que o sistema de arquivos tenha conhecimento da criação ou retirada de um processo.

Para a implementação do MINIX sobre o Nó//, essas características nos induzem a adotar um processo servidor de arquivos equivalente ao sistema de arquivos e, considerando ainda a alocação de um único processo de usuário por nó, reduzir o papel do gerente de memória ao de alocador de nós.

Além dos servidores, o MINIX comporta ainda as tarefas controladoras de periféricos e os processos usuários. As tarefas podem ser realizadas como processos independentes que trocam mensagens exclusivamente com os servidores. Todos os processos usuários são criados dinamicamente segundo o esquema da chamada de sistema *fork*, em resposta às necessidades dos usuários do sistema. As tarefas de entrada e saída podem ser combinadas de modo a serem alocadas aos nós da máquina que estejam conectados aos dispositivos periféricos geridos. A tarefa de sistema (*sys_task*()) responsável por atender as chamadas especiais para comunicação com o núcleo deve ser alocada ao nó de controle.

As únicas restrições de colocação do Nó// são devidas a existência de um nó específico: o nó de controle.

A colocação do gerente de memória sobre o nó de controle é obrigatória. Por outro lado, o sistema de arquivos pode ser colocado sobre qualquer nó de trabalho sem que a escolha seja importante. As tarefas podem ser colocadas sobre os nós de trabalho. Os processos usuários são alocados dinamicamente aos nós de trabalho segundo um critério completamente arbitrário, uma vez que as comunicações se fazem através de canais diretos.

Resta ainda considerar a função de estabelecimento de canais de comunicação entre nós. Uma vez que a criação de um processo deve ser acompanhada da conexão imediata do nó alocado ao do pai, para a operação de duplicação da imagem de memória, uma solução eficaz é a de incluí-la no gerente de memória.

5 PROPOSTA DE UM MECANISMO DE COMUNICAÇÃO

Neste capítulo, descrevemos inicialmente a proposta de um mecanismo de comunicação flexível, simples e eficiente. Na seqüência, são examinadas as condições exigidas para sua implementação no sistema operacional MINIX e na máquina Nó//.

5.1 O Mecanismo de Comunicação

O mecanismo de comunicação que propomos visa atender, especificamente, às interações do modelo cliente-servidor para prover suporte às comunicações síncronas inerentes às chamadas de procedimentos remotos. A flexibilidade do mecanismo proposto deve-se à liberdade de formato e de tamanho de mensagens além da transparência de localidade de processos conferida pela referência a caixas postais interpostas entre processos clientes e servidores. Além disso, um outro aspecto que caracteriza flexibilidade e principalmente eficiência é a inexistência de cópias intermediárias de mensagens, liberando o núcleo da tarefa de gerência de *buffers* para armazenamento temporário.

Há três componentes envolvidos no mecanismo proposto: caixas postais, requisições de comunicação e mensagens. No mecanismo, requisições de comunicação transitam através de caixas postais enquanto que as mensagens propriamente ditas são transmitidas diretamente entre os espaços de endereçamento dos processos envolvidos na comunicação. Uma requisição de comunicação é um aviso de um cliente para um servidor de que ele precisa fazer uso de seus serviços. As mensagens são transmitidas somente quando ambos os processos estão simultaneamente preparados para que isso ocorra.

A figura 5-1 apresenta um esboço desse mecanismo. Ao necessitar de um serviço de determinado servidor, o processo cliente envia uma requisição de comunicação para a caixa postal do processo servidor (1) e fica bloqueado até receber uma resposta ao serviço solicitado. O processo servidor, por sua vez, quando está apto a atender uma requisição de serviço consulta sua caixa postal (2). Não havendo requisição nela armazenada, ele fica bloqueado até que alguma

seja depositada. Caso existam requisições em sua caixa postal, a mais antiga é atendida. Neste momento ocorre uma comunicação síncrona entre os processos cliente e servidor, com a mensagem efetiva sendo copiada diretamente da área de memória do cliente para a área de memória do servidor (3). O servidor então interpreta a mensagem, executa o serviço e envia uma nova mensagem para o cliente contendo a resposta ao serviço solicitado (4).

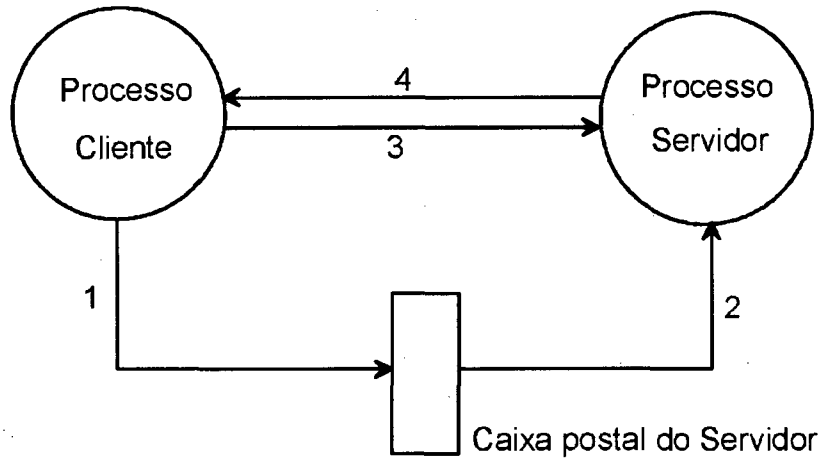

Figura 5-1. Esboço do mecanismo de comunicação

Podemos dizer que o mecanismo proposto representa, de certa forma, uma combinação da flexibilidade da comunicação assíncrona indireta com a simplicidade da comunicação síncrona direta. Transparência de localização entre processos que se comunicam é conferida ao mecanismo pela utilização de caixas postais e otimização de trânsito de mensagens é alcançada pela eliminação do armazenamento intermediário de mensagens.

A seguir definimos cada um dos componentes envolvidos no mecanismo de comunicação proposto.

5.1.1 Requisições de Comunicação

Uma requisição de comunicação constitui-se de um aviso oriundo de um processo cliente para um processo servidor indicando que o primeiro necessita fazer uso de serviços oferecidos pelo segundo. A requisição de comunicação fica registrada temporariamente na caixa

postal até que o servidor a atenda. É desejável que uma requisição deste tipo tenha um tamanho reduzido para economizar espaço de armazenamento mas que, ao mesmo tempo, contenha todas as informações necessárias para a transação completa do modelo cliente-servidor. Assim, propomos uma requisição composta pela identificação do processo cliente (cli_id), o endereço (&mess1) e o tamanho da mensagem (size_mess1) a ser enviada pelo cliente, o endereço (&mess2) e tamanho da área de memória (size_mess2) reservada pelo cliente para recepção da mensagem de resposta, além de um ponteiro (p_next_req) para o encadeamento das requisições. Estes campos estão representados na figura 5-2.

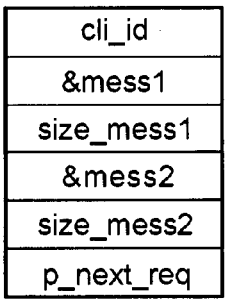

Figura 5-2. Uma requisição de comunicação

O campo de identificação do cliente é utilizado no procedimento de envio da resposta do servidor. Os campos seguintes (referentes a endereços e tamanhos de áreas de memória do cliente) são combinados com as informações complementares fornecidas pelo servidor através das primitivas de comunicação por ele executadas para permitir que o núcleo possa realizar as cópias de mensagens. O campo do ponteiro permite que o núcleo mantenha a fila de requisições de comunicação.

5.1.2 Caixas Postais

Somente processos servidores são propretários de caixas postais e têm direito de leitura sobre elas. Cada processo servidor possui uma única caixa postal. Processos clientes têm direito de escrita sobre as caixas postais, uma vez que o núcleo registra suas requisições de comunicação nelas. As requisições de comunicação são atendidas pelos servidores na ordem de chegada.

As caixas postais, mantidas em uma tabela pelo núcleo do sistema, são identificadas pelos índices de suas entradas nesta tabela. Quando um processo servidor é criado, o núcleo aloca uma entrada nessa tabela e registra sua identificação no registro descritor do processo servidor criado.

A caixa postal de um servidor é composta pela fila de requisições de comunicação a ela dirigidas. A fila é registrada por dois ponteiros: um para a primeira requisição (first_req) e outro para a última (last_req). A figura 5-3 apresenta um esboço da tabela de caixas postais com uma fila de requisições de comunicação. As primeiras entradas da tabela de caixas postais são alocadas estaticamente aos processos controladores de periféricos e aos servidores permanentes do sistema. Um dos servidores permanentes é o servidor de nomes de serviços que será apresentado no capítulo 6. Os demais servidores são chamados servidores não permanentes ou dinâmicos. As demais entradas da tabela são associadas a outros servidores admitidos dinamicamente no sistema. Essas entradas podem então estar alocadas a diferentes processos servidores em momentos distintos durante a execução do sistema. A forma de inclusão de servidores no sistema por demanda será apresentada no capítulo 6.

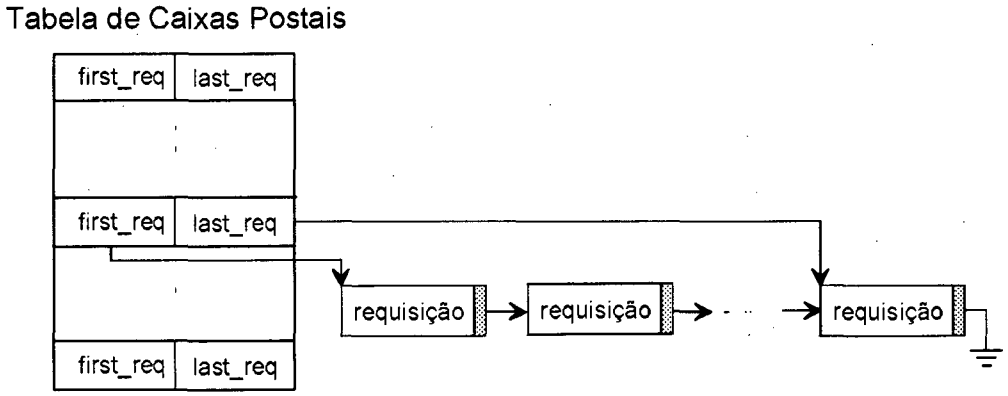

Figura 5-3. Tabela de caixas postais com uma fila de requisições de comunicação

As identificações das caixas postais dos processos controladores de periféricos somente são conhecidas por seus proprietários e pelos servidores permanentes que são seus clientes. As caixas postais dos servidores permanentes do sistema são públicas e acessíveis para escrita a partir de todos os processos usuários. O núcleo associa a cada processo, durante sua

criação, uma tabela contendo ponteiros para as caixas postais dos servidores a que ele tem direito de acesso para escrita.

5.1.3 Mensagens

Mensagem é a unidade lógica de informação trocada entre processos de um sistema [ATH88]. As mensagens podem ter tamanho fixo ou variável e seus formatos podem ser estabelecidos pelo sistema ou definidos pelos processos que as utilizam.

No mecanismo proposto, os tamanhos das mensagens ficam a cargo exclusivamente dos processos envolvidos, uma vez que não dependem de áreas de armazenamento intermediárias. O formato das mensagens é transparente para o núcleo, que se responsabiliza simplesmente por gerir as transmissões de blocos de dados entre as áreas de memória dos processos envolvidos em uma determinada comunicação.

Cada processo servidor tem autonomia para definir o formato das mensagens que está apto a receber e enviar. Desta forma, cada processo cliente que deseja obter serviços de determinado servidor deve conhecer o formato da mensagem com a qual aquele servidor trabalha.

5.1.4 Primitivas de Comunicação

O mecanismo que propomos dispõe de três operações de comunicação orientadas ao modelo cliente-servidor: **sendrec**, usada por clientes, **send** e **receive** usadas pelos servidores. Elas são implementadas no núcleo do sistema e acessadas através de procedimentos de biblioteca.

A primitiva **sendrec** é usada por processos clientes que estejam requerendo serviço de algum servidor. Sua função é a de enviar uma mensagem contendo uma solicitação de serviço e bloquear o processo cliente até que ele receba uma mensagem do servidor contendo a resposta ao serviço solicitado. A chamada dessa primitiva tem o seguinte formato:

sendrec(sd, &mess1, size_mess1, &mess2, size_mess2).

Os parâmetros da chamada **sendrec**, exceto **sd**, são usados na composição da requisição de comunicação estabelecida em 5.1.1. O parâmetro **sd** representa o descritor do servidor ao qual a mensagem é destinada. Este descritor do servidor é o índice da entrada na tabela de servidores associada ao processo cliente, que contém ponteiros para as caixas postais a que ele tem direito de escrita. O parâmetro **&mess1** é o endereço da área de memória do processo cliente onde a mensagem de requisição de serviço está armazenada e **size_mess1** é o seu tamanho. O parâmetro **&mess2** é o endereço da área reservada pelo cliente para armazenar a mensagem de resposta do servidor e **size_mess2** é o seu tamanho. O núcleo armazena na caixa postal epecificada uma requisição de comunicação correspondente à chamada executada pelo processo cliente.

A chamada **receive** é executada por processo servidor quando este está apto a atender algum serviço de cliente. Sua função é receber uma mensagem contendo uma solicitação de serviço de um cliente. Após a realização desta chamada, o servidor pode executar as tarefas necessárias para o atendimento do serviço requisitado. A chamada da primitiva **receive** tem o seguinte formato:

cli_id = **receive**(&mess, size_mess).

A caixa postal é determinada implicitamente através do registro descritor do processo servidor. O endereço e o tamanho da área de memória do cliente aonde a mensagem contendo a resposta do serviço será armazenada são obtidos a partir da requisição de comunicação armazenada na caixa postal do servidor. Os parâmetros **&mess** e **size_mess** especificam respectivamente o endereço e o tamanho da área de memória aonde a mensagem do cliente deverá ser recebida. Ao executar tal chamada, o servidor fica bloqueado caso não haja requisição de comunicação armazenada em sua caixa postal. Caso haja alguma requisição, a mensagem é transmitida diretamente da área de memória do cliente para a área de memória do servidor. Caso o tamanho da mensagem enviada seja maior que o tamanho da área reservada para seu armazenamento, uma mensagem de erro é retornada pelo núcleo a ambos os processos envolvidos e a operação é interrompida. O processo servidor recebe como retorno a identificação do processo cliente (cli_id) ao qual prestará serviço.

A primitiva **send** é executada por processo servidor após a realização de um serviço solicitado por um cliente. Sua função é enviar uma mensagem contendo a resposta ao serviço solicitado por um cliente. Esta chamada não bloqueia o processo servidor, uma vez que o cliente estará sempre apto a receber imediatamente a mensagem o que é garantido pela primitiva **sendrec**. A chamada dessa primitiva tem o seguinte formato:

send(&mess, size_mess).

A identificação do cliente, assim como o endereço e o tamanho da área de memória onde a mensagem oriunda do servidor será gravada, são obtidas a partir da requisição de comunicação armazenada na caixa postal do servidor. Os parâmetros **&mess** e **size_mess** especificam o endereço da área de memória do servidor onde a mensagem de resposta está armazenada e o seu tamanho. Caso o tamanho da mensagem seja maior do que o da área reservada pelo cliente para o seu armazenamento, a operação é interrompida e uma mensagem de erro é enviada a ambos os processos envolvidos. Caso contrário, o núcleo realiza a cópia da mensagem da área de memória do servidor para a área de memória do cliente.

5.2 Proposta de Implementação no MINIX

Apresentamos a seguir a proposta de implementação do mecanismo de comunicação utilizando o sistema operacional MINIX. Propomos a substituição do mecanismo de comunicação original do sistema apontando as funções e estruturas a serem alteradas e/ou incluídas.

Para a substituição do mecanismo de comunicação atual do MINIX, devem ser alteradas funções do núcleo, chamadas de sistema e o procedimento interpretador das primitivas de comunicação. Além disso, novas estruturas devem ser inseridas e outras alteradas. Primeiramente descreveremos as estruturas a serem alteradas e inseridas, depois as funções a serem alteradas e criadas.

5.2.1 Estruturas Novas ou Modificadas

Duas novas estruturas são criadas: a fila de requisições de comunicação e a tabela de caixas postais. Uma outra estrutura, a tabela de processos do sistema, é modificada. As novas estruturas são inseridas no arquivo usr/src/kernel/proc.h por serem estruturas manipuladas somente pelo núcleo. A fila de requisições de comunicação é uma lista na qual cada nó é composto por três inteiros e três ponteiros. Os campos inteiros correspondem à identificação do cliente (cli_id) e aos tamanhos da mensagem a ser enviada (size_mess1) e da área reservada para recepção da resposta (size_mess2). Dois ponteiros são os endereços da mensagem a ser enviada (&mess1) e da área reservada para armazenamento da resposta (&mess2). O outro ponteiro serve para permitir o encadeamento de requisições de comunicação da fila. Uma nova constante (#define NIL_REQ ((struct queue_req *) 0)) é criada para auxiliar na manipulação da fila de requisições de comunicação.

A tabela de caixas postais é um vetor que tem uma entrada alocada para cada servidor, permanente ou não, ativo no sistema, incluindo as tarefas de entrada e saída. Cada entrada é composta por dois ponteiros. Um para a primeira requisição de comunicação da fila (first_req) e o outro para a última (last_req). A tabela de processos do sistema contém uma entrada para cada processo existente, quer seja tarefa, servidor ou processo usuário. A tabela de caixas está limitada a um número razoavelmente menor do que o da tabela de processos. Desta forma, definimos mais uma constante no arquivo usr/include/minix/const.h (#define MAX_DYN_SERVERS 8) para limitar o número máximo de servidores não permanentes em execução. Cria-se uma constante (#define FREE_MBOX (struct queue_req *) -1) que é atribuída ao ponteiro para a primeira requisição de comunicação quando a caixa postal é liberada. A alocação de uma caixa postal é feita pela atribuição de NIL_REQ aos ponteiros de primeira e última requisição.

A tabela de processos (struct proc) no arquivo usr/src/kernel/proc.h que contém o registro descritor de cada processo é modificada com o acréscimo do campo inteiro (int server_mb) usado para identificar a caixa postal na qual o processo recebe suas requisições de comunicação. Desta forma, esse campo só é usado para processos servidores e tarefas de entrada e saída. Os servidores permanentes e as tarefas de entrada e saída terão este campo definido ao serem ativados durante a inicialização do sistema (as identificações das caixas postais de

servidores permanentes e tarefas de entrada e saída são constantes). Os servidores não permanentes terão este campo atualizado quando de cada ativação pela chamada de sistema *fork*. A diferença entre um processo qualquer e um processo servidor não permanente está na identificação do processo pai. Só os servidores não permanentes terão o servidor de nomes de serviços como pai. Neste caso, a função *do_fork* do arquivo usr/src/kernel/system.c aloca uma caixa postal livre para o processo filho e atualiza o campo server_mb do registro descritor do novo processo. Uma outra alteração na estrutura da tabela de processos é a inserção do vetor de descritores de servidores. Este vetor contém em cada entrada um campo inteiro que é utilizado para armazenar a identificação da caixa postal de um servidor que o processo pode acessar. A função *do_fork* faz a inicialização deste vetor para cada novo processo com as identificações das caixas postais dos servidores a que ele tem acesso ao ser criado.

5.2.2 Programas e Funções

Devem ser alterados o programa usr/src/kernel/main.c responsável pela inicialização do sistema, o programa usr/src/kernel/proc.c responsável pela gerência de processos e comunicações, além do programa usr/src/kernel/system.c que contém a função *do_fork* que realiza a parte da chamada de sistema *fork* no núcleo. Além disto, devem ser modificados os programas captadores de cada chamada de sistema e o programa usr/src/lib/other/call.c, que transforma a chamada em mensagem para o servidor que a tratará.

O programa usr/src/kernel/main.c passa a conter no código da sua função principal (PUBLIC main()) a inicialização da tabela de caixas postais através da atribuição da constante NIL_REQ a todos os ponteiros de primeira e última requisição das caixas postais das tarefas de entrada e saida e dos servidores permanentes. As demais entradas da tabela, utilizadas pelos servidores não permanentes são registradas como caixas postais livres pela atribuição da constante FREE_MBOX aos seus ponteiros de primeira requisição. Esta inicialização é feita logo após a definição do endereço inicial do gerente de memória e antes da execução do laço onde há a ativação dos processos referentes às tarefas de entrada e saída e aos servidores permanentes.

O programa usr/src/kernel/proc.c sofre alterações nas funções relacionadas com a implementação das primitivas de comunicação: *sys_call*, *mini_send* e *mini_rec*. Além disto,

cria-se uma nova função *m_send_rec* para prover ao sistema uma primitiva real de envio e recepção de mensagem. A função *sys_call* é totalmente modificada não só pela adição de mais alguns parâmetros mas também porque ela tem um novo comportamento. Ela recebe como parâmetros o código da função de comunicação a ser realizada (SENDREC, RECEIVE ou SEND), um descritor de servidor, o endereço e o tamanho da mensagem a ser enviada e o endereço e o tamanho da área reservada para recepção da mensagem de resposta. Inicialmente, *sys_call* verifica algumas consistências a respeito dos parâmetros recebidos. Por exemplo, se o processo ativador é cliente, a função chamada só pode ser SENDREC. Após a crítica dos parâmetros, uma função associada à operação é ativada: *m_send_rec* caso a operação seja SENDREC, *mini_rec* caso a operação seja RECEIVE e *mini_send* caso a operação seja SEND. As funções *mini_rec* e *mini_send* adequam os parâmetros recebidos e transferem o controle de execução para a rotina *cp_mess* do programa usr/src/kernel/klib.s para que seja feita a cópia física da mensagem. Esta última função (*cp_mess*) é preservada sem modificacões.

O MINIX original implementa **sendrec** como um *mini_send* seguido imediatamente por um *mini_rec* sendo a função *sys_call* responsável por gerir essa seqüência. Propomos a função *m_send_rec* com o objetivo de prover ao sistema uma primitiva verdadeira de envio e recepção de mensagem. Ela recebe como parâmetros a identificação do processo cliente, o descritor do servidor, o endereço e o tamanho da mensagem a ser enviada, e o endereço e o tamanho da área reservada pelo cliente para armazenar a resposta. A caixa postal onde a requisição de comunicação será armazenada está contida na entrada do vetor de descritores de servidores do processo cujo índice é o descritor do servidor. Inicialmente é alocada uma estrutura de requisição de comunicação atualizando seus campos e inserindo-a na fila de requisições associadas à caixa postal. Se a fila está vazia, o servidor deve ser desbloqueado. Por último, a função bloqueia o processo cliente que só será desbloqueado quando receber a mensagem de resposta ao serviço solicitado.

A função *mini_rec* recebe como parâmetros a identificação do processo servidor que ativou a primitiva **receive**, o ponteiro para a área reservada para armazenamento da mensagem a ser recebida e o tamanho desta área. A caixa postal a ser consultada é aquela identificada no registro descritor do processo ativador da primitiva cuja identificação é recebida como parâmetro. A primeira atividade da função é consultar a fila de requisições de comunicação. Caso não haja requisição de comunicação (first_req == NIL_REQ), o processo servidor é

bloqueado. Caso contrário, é feita uma crítica sobre a compatibilidade do tamanho da mensagem e o da área reservada para seu armazenamento. Se o tamanho da mensagem for menor ou igual ao da área, a cópia da mensagem é feita através da função *cp_mess* e a identificação do processo cliente é retornada. Caso a mensagem tenha tamanho maior que o da área reservada, a cópia não é realizada e um código de erro é retornado.

A função *mini_send* recebe como parâmetros a identificação do processo servidor que ativou a primitiva **send**, o endereço da mensagem a ser enviada e o seu tamanho. Quando esta função está sendo executada, há obrigatoriamente um cliente bloqueado esperando por esta mensagem e desta forma não é necessário haver bloqueio do servidor. A função então faz a crítica de compatibilidade entre o tamanho da mensagem e o da área reservada para seu armazenamento. Se forem compatíveis a cópia efetiva de mensagem através de *cp_mess* é realizada e o processo cliente é desbloqueado. Caso contrário, um código de erro é retornado ao cliente antes do seu desbloqueio.

A função *s_call* do arquivo usr/src/kernel/mpx.s e o procedimento interpretador das primitivas de comunicação (usr/src/lib/ibm/sendrec.s) devem adaptar-se aos três novos parâmetros devido à nova sintaxe da primitiva **sendrec**. O procedimento interpretador deve fazer uso de mais registradores para viabilizar a passagem dos parâmetros adicionais e a função *s_call* deve adequá-los para serem passados como parâmetros para a função *sys_call*.

O programa usr/src/kernel/system.c que contém o código da tarefa de sistema (*sys_task*()) sofre algumas modificações. O código que realiza a chamada de sistema *fork* no núcleo encontra-se na função *do_fork* que é ativada após o gerente de memória ter realizado sua parte da chamada de sistema na função *do_fork* do programa usr/src/mm/forkexit.c. O gerente de memória faz uma cópia da imagem do processo pai em memória e aloca uma entrada para o novo processo na sua parte da tabela de processo. Então, cria um identificador de processo (*pid*) e o atribui ao processo recém criado. Feito isto, comunica ao núcleo e ao sistema de arquivos a criação do novo processo para que estes façam suas partes. Por último, comunica ao núcleo o mapa de memória do novo processo e retorna o identificador do processo criado ao processo pai e 0 ao processo filho. O núcleo inicializa os campos do registro descritor do novo processo. Esse procedimento é alterado para acrescentar um teste visando a correta inicialização do campo caixa postal (server_mb) no qual um processo servidor deverá atender requisições de serviço. Este

campo deverá ser nulo caso o processo pai seja qualquer processo, exceto o servidor de nomes de serviços que será visto no capítulo 6. Caso este seja o processo pai do que está sendo criado, a função deve procurar uma caixa postal disponível. O índice de entrada desta caixa será a identificação da caixa postal do novo processo. A inicialização do vetor de descritores de servidores do processo é inserida, após a atualização do campo caixa postal, com as primeiras posições preenchidas com as identificações das caixas postais dos servidores aos quais todos os processos têm acesso ao serem criados que são: gerente de memória, sistema de arquivos e servidor de nomes de serviços, além dos servidores não permanentes herdados do pai.

As alterações nas ativações de chamadas de sistema devem ser feitas de modo que os programas captadores delas passem a se referir ao descritor do servidor que atenderá à chamada. Originalmente, a referência é feita diretamente ao identificador do processo servidor que é constante. Agora, os descritores dos servidores permanentes e suas caixas postais são constantes mas os servidores não permanentes, que serão introduzidos no capítulo 6, podem utilizar diferentes caixas postais a cada nova ativação.

5.3 Proposta de Implementação no Nó//

A proposta original de um ambiente para programação paralela para o Nó// prevê um mecanismo de comunicação hierárquico realizado em duas camadas [COR93]:

- a camada das comunicações de baixo nível que permite a troca de mensagens compactas que transitam pelo barramento de serviço, e

- a camada das comunicações de alto nível que permite a troca de mensagens volumosas através do *crossbar*.

As primitivas da camada de baixo nível servem exclusivamente para suportar a realização da camada de alto nível. Em ambos os níveis, o mecanismo está orientado a comunicações síncronas diretas.

Na implantação da interface de programação MINIX através de chamadas de procedimentos remotos, as trocas de mensagens entre processos dependem do serviço de

comunicação provido através do *crossbar*. O mecanismo de comunicação que propomos deve, então, substituir a camada de alto nível original. Entretanto, ele deve apoiar-se sobre o mecanismo original de baixo nível sem alterá-lo.

5.3.1 A Camada das Comunicações de Baixo Nível

As comunicações pelo barramento de serviço têm sempre o nó de controle como origem ou destino e todas elas se desenvolvem sob o comando desse nó. Primitivas específicas para o nó de controle e para os nós de trabalho evidenciam essa assimetria de comportamento:

W_SendReceive (request, reply)

C_ReceiveAny (node, request)

C_Send (node, reply)

Um processo executando em um nó de trabalho qualquer solicita através da primitiva **W_SendReceive** o envio ao nó de controle de uma mensagem de requisição de serviço *request* e a recepção de uma mensagem de resposta em *reply*. O processo executando no nó de controle solicita através da primitiva **C_ReceiveAny** a recepção de uma mensagem de requisição de serviço em *request* de um nó de trabalho qualquer cuja identificação é devolvida em *node*. O processo executando no nó de controle solicita através da primitiva **C_Send** o envio de uma mensagem de resposta *reply* a um nó de trabalho identificado por *node*.

Essas primitivas são orientadas à disciplina de comunicação do tipo cliente-servidor. Assim, um cliente executa a primitiva **W_SendReceive** (para o envio de uma requisição de serviço e a recepção da resposta) e um servidor executa a primitiva C_ReceiveAny (para a recepção de uma requisição de serviço) e C_Send (para o envio da resposta).

5.3.2 A Camada das Comunicações de Alto Nível

O mecanismo de comunicação apresentado na seção 5.1 compõe a camada das comunicações de alto nível. As requisições de comunicação trafegam pelo barramento de serviço

e são armazenadas nas caixas postais que são geridas pelo núcleo no nó de controle. Já as mensagens contendo solicitações de serviços e suas respostas transitam pelo *crossbar* quando a conexão já está estabelecida entre os processos comunicantes.

O núcleo no nó de controle é responsável pela localização dos nós envolvidos na comunicação e pela associação de caixas postais a seus servidores proprietários.

Esta camada utiliza serviços providos pela camada de comunicações de baixo nível. Assim, as primitivas de comunicação utilizadas nesta camada são transformadas nas primitivas utilizadas na camada das comunicações de baixo nível.

A primitiva **receive** executada por um processo servidor alocado a um determinado nó de trabalho é transformada em um **C_ReceiveAny** que é captado pelo núcleo no nó de controle. Ao ser estabelecido o canal de comunicação entre o servidor e o cliente ao qual servirá alocado em um outro nó, uma primitiva de recepção de baixo nível é executada. A primitiva **send** é transformada em **C_Send** que é captada pelo núcleo. Após o estabelecimento da conexão, uma primitiva de envio de baixo nível é então executada.

6 PROPOSTA DE UM MÉTODO DE ATIVAÇÃO DE SERVIDORES POR DEMANDA

Neste capítulo, apresentamos a proposta de um método de ativação de servidores por demanda de seus clientes. Também discutimos as condições exigidas para a introdução do método no sistema MINIX e no multicomputador Nó//.

6.1 O Servidor de Nomes de Serviços

Um sistema operacional pode contar com um número fixo de servidores permanentes que estão sempre disponíveis, mas pode necessitar eventualmente de outros servidores não permanentes ou dinâmicos que são adicionados durante sua operação. Se, em determinado momento, um novo serviço passa a ser útil a muitos usuários, pode-se pensar na possibilidade de elaboração de um processo servidor para atender aquela carência comum. A existência destes novos serviços deve ser divulgada a todos os usuários do sistema. Uma alternativa de tornar os servidores disponíveis consiste em divulgar seus endereços. Contudo, essa solução apresenta o incoveniente de conferir rigidez ao sistema, por exemplo, com relação à localização de servidores. Outra, consiste em divulgar o nome do servidor. Neste caso, surge a necessidade de oferecer um servidor de nomes para mapear os nomes dos servidores em seus endereços. Assim, um usuário pode obter o endereço atual de um servidor específico através do servidor de nomes que serve, então, como intermediário entre clientes e servidores, conferindo independência de localidade aos servidores.

Considerando a importância da independência de localidade para os sistemas distribuídos, optamos pela última alternativa, incluindo um método de ativação de servidores por demanda.

Alguns sistemas distribuídos utilizam um esquema de **ligação dinâmica** [TAN92] para tornar servidores disponíveis a clientes, nos quais um servidor ao ser ativado comunica-se com um processo chamado **ligador** que faz seu registro junto ao sistema e torna-o público. Para

registrar-se, o servidor comunica ao **ligador** seu nome, seu identificador único e seu endereço. Ao realizar uma chamada que é atendida por um servidor, o procedimento intermediário do cliente verifica se o servidor que a atende já está disponível para ele. Caso não esteja, uma mensagem é enviada ao **ligador** requisitando a sua disponibilidade. O **ligador** verifica se aquele servidor já está disponível no sistema. Caso esteja, ele comunica ao cliente a sua identificação e localização. Caso contrário a chamada do cliente falha. Neste esquema, os servidores são ativados explicitamente por um agente externo como o super-usuário, por exemplo, independentemente de serem requisitados e permanecem ativos mesmo que não haja usuários para eles. A pior situação ocorre quando alguma chamada para um servidor não é realizada pelo fato dele não estar ativo.

O método que propomos resolve estes problemas: um servidor só é ativado e permanece ativo se houver cliente para ele. Além disso, a forma de ativação de servidores no método que propomos fica a cargo de um servidor do sistema e não de algum agente externo. O método de ativação de servidores é viabilizado por um processo servidor de nomes de serviços que é um servidor permanente do sistema, disponível a todos os usuários. Sua função é tornar servidores não permanentes disponíveis aos processos clientes que os requeiram.

O servidor de nomes de serviços funciona como um agente intermediário entre processos clientes e processos servidores não permanentes do sistema. Ele fornece apenas dois serviços a clientes: tornar um serviço disponível a cliente que o solicitar e tornar um serviço não mais disponível a cliente quando este não mais desejar utilizá-lo.

A interação entre o servidor de nomes de serviços, seus clientes e os servidores dinâmicos pode ser entendida como segue. Num primeiro momento, o processo cliente requer através de uma mensagem ao servidor de nomes a utilização de um determinado serviço provido por um servidor não permanente. A requisição é feita por referência ao nome do serviço. O servidor de nomes de serviços, então, torna disponível o servidor desejado ao cliente, retornando a ele o descritor do servidor. Esse descritor de servidor deve ser referido pelo cliente a partir de então para acessá-lo. Quando não mais necessitar daquele serviço, o cliente libera o servidor através de uma mensagem enviada ao servidor de nomes de serviços.

A inclusão de um novo serviço no sistema consiste de um procedimento realizado pelo super-usuário. O código executável do servidor correspondente é gravado em um diretório

determinado e eventuais procedimentos de biblioteca que facilitem seu uso podem ser adicionados a bibliotecas específicas. O arquivo com o cadastro dos serviços é atualizado e o nome do novo serviço pode então ser divulgado.

A utilização de serviços conforme propomos guarda alguma analogia com a utilização de arquivos no sistema UNIX. A similaridade diz respeito à forma como o servidor dinâmico é referido. Quando requerido, o serviço é referido pelo nome assim como um arquivo é referido pelo nome por ocasião de sua abertura. Durante sua utilização e na liberação, o cliente refere-se ao serviço através de um descritor de servidor (índice de uma tabela associada ao cliente). Da mesma forma, no sistema de arquivos as referências ao arquivo após sua abertura e até seu fechamento também são feitas por um descritor e não mais por seu nome. As chamadas de sistema para alocação e liberação de serviço atendidas pelo servidor de nomes de serviços serão apresentadas na seção 6.1.3.

6.1.2 Estruturas de Dados

O mecanismo de ativação de servidores proposto se apóia sobre o uso de cinco estruturas de dados. Três delas são implementadas no servidor de nomes de serviço: a **tabela de servidores dinâmicos existentes** no sistema, a **tabela de servidores dinâmicos que estão ativos** no sistema e uma **tabela de servidores em uso por cliente**. As outras duas são implementadas no núcleo: uma **tabela de servidores em uso por cliente** e a **tabela de caixas postais** dos servidores do sistema definida na seção 5.1.2.

6.1.2.1 Estruturas de Dados do Servidor de Nomes de Serviços

A **tabela de servidores dinâmicos existentes** no sistema é composta pelos nomes de todos os servidores não permanentes cadastrados no sistema por responsabilidade do super-usuário. A tabela é carregada a partir de um cadastro armazenado em um arquivo que contém os nomes pelos quais os serviços são conhecidos e os nomes dos arquivos onde seus códigos executáveis podem ser encontrados. A figura 6-1 apresenta um esboço dessa tabela.

Nome do Serviço	Nome do Arquivo Contendo o Programa Servidor
Serviço L	Arquivo do Servidor L
Serviço M	Arquivo do Servidor M
Serviço N	Arquivo do Servidor N
⋮	⋮

Figura 6-1. Tabela de servidores não permanentes do sistema

Outra estrutura manipulada pelo servidor de nomes de serviços é uma **tabela composta pelos nomes dos servidores não permanentes que estão ativos**, ou seja, que estão sendo utilizados por algum processo cliente. Além dos nomes dos servidores, cada entrada na tabela contém a identificação do processo servidor e um contador do número de usuários que o estão utilizando. Esse contador permite verificar quando não há mais clientes para aquele serviço. Quando isto ocorre, o servidor de nomes de serviços libera a entrada do servidor nesta tabela e desativa o processo servidor fazendo uso da identificação do processo. A figura 6-2 apresenta um esboço desta tabela. Esta tabela seria análoga à tabela de arquivos abertos no sistema de arquivos UNIX que permite também verificar quando não há mais usuários utilizando um determinado arquivo.

Nome Serviço	Identificação do Processo Servidor	Contador de Usuários do Serviço
Serviço L	pid Servidor L	Contador de usuários de L
Serviço M	pid Servidor M	Contador de usuários de M
Serviço N	pid Servidor N	Contador de usuários de N
⋮	⋮	⋮

Figura 6-2. Tabela de servidores não permanentes ativos

Cada **tabela de servidores em uso por cliente** (ou vetor de descritores de servidores mantido pelo servidor de nomes de serviços) possui registros de todos os servidores disponíveis aos clientes (uma tabela por cliente). Suas entradas contêm os índices da **tabela de servidores não permanentes ativos**. O núcleo mantém tabelas correspondentes a estas, uma por processo, como será visto a seguir. A figura 6-3 apresenta um esboço desta tabela.

Figura 6-3. Tabela de descritores de servidores em um uso por cliente

A figura 6-4 apresenta uma visão lógica do relacionamento entre a **tabela de servidores em uso por cliente** e a **tabela de servidores não permanentes ativos**. Cada entrada da tabela do cliente contém o índice de entrada na tabela de servidores ativos mantida pelo servidor de nomes de serviços. Na situação mostrada na figura 6-4, há 3 três servidores não permanentes ativos. O processo X está utilizando dois deles, L e M. O servidor M está sendo utilizado por dois processos, X e Y. O servidor N, por sua vez, está sendo utilizado somente pelo cliente Z.

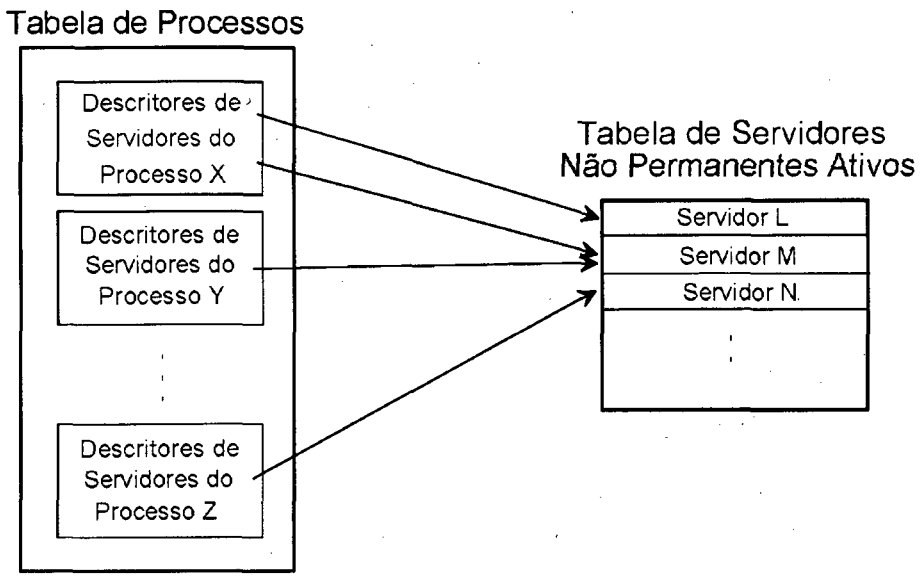

Figura 6-4. Tabela de servidores em uso pelo cliente e a tabela de servidores não permanentes ativos

6.1.2.2 Estruturas de Dados do Núcleo

A **tabela de servidores em uso por cliente** mantida pelo núcleo esboçada na figura 6-5 faz parte do registro descritor de cada processo. Sua estrutura é análoga à da tabela de descritores de arquivos em uso por processo no sistema UNIX. Sua finalidade é manter um registro de cada servidor que está sendo utilizado pelo processo da mesma forma que a tabela de descritores de arquivo por processo mantém um registro de cada arquivo que está sendo utilizado pelo processo. O índice de uma entrada nesta tabela é retornado pelo servidor de nomes de serviços por ocasião do provimento do serviço ao cliente. Este índice é o mesmo da tabela correspondente mantida pelo servidor de nomes de serviços. Este índice é o descritor de servidor (sd) que deve ser usado como parâmetro da primitiva **sendrec** (ver seção 5.1.4) para identificação do servidor destinatário da mensagem. Cada entrada desta tabela contém um ponteiro para a caixa postal na qual o servidor recebe suas requisições de comunicação.

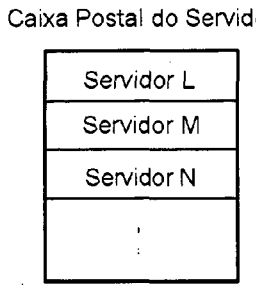

Figura 6-5. Tabela de servidores em uso por um processo

A **tabela de caixas postais** de servidores dinâmicos do sistema faz parte da mesma estrutura que é utilizada para as caixas postais do servidores permanentes (ver seção 5.1.2). Os servidores permanentes possuem caixas postais fixas enquanto os servidores dinâmicos podem estar associados a diferentes caixas postais a cada nova ativação.

Quando um processo é criado, sua tabela de servidores em uso (figura 6-5) é inicializada com ponteiros tanto para as caixas postais dos servidores permanentes do sistema como para os servidores não permanentes herdados do processo pai. Esta herança de servidores não permanentes em uso exige que o servidor de nomes de serviços atualize os respectivos contadores de usuários de cada um destes servidores.

A figura 6-6 apresenta um esboço do relacionamento entre a tabela de servidores em uso pelo cliente mantida pelo núcleo e a tabela de caixas postais. Cada entrada da primeira tabela contém um índice de entrada na tabela de caixas postais correspondente à caixa postal do servidor.

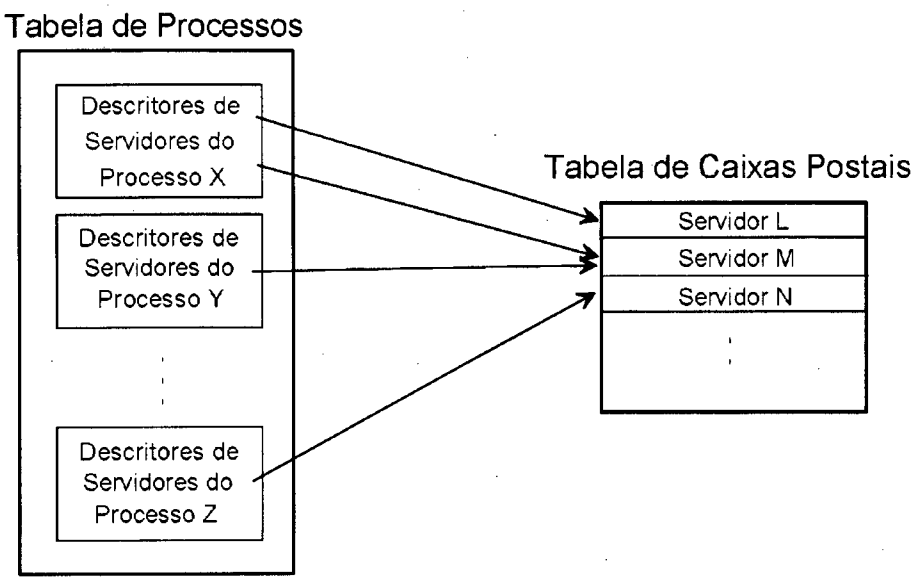

Figura 6-6. Tabela de servidores em uso pelo cliente e a tabela de caixas postais

6.1.3 Chamadas de Sistema para o Servidor de Nomes de Serviços

Definimos duas chamadas de sistema para o servidor de nomes de serviços: uma para alocação e outra para liberação de servidor não permanente do sistema. Elas são implementadas da mesma forma que as demais chamadas de sistema do MINIX, ou seja, como chamadas de procedimentos remotos.

A chamada de sistema ***get_service*** permite que um processo cliente passe a utilizar os serviços de um determinado servidor não permanente do sistema. Seu formato é o seguinte:

sd = ***get_service***(nome_serviço);

A chamada ***get_service*** recebe como parâmetro o nome do serviço que o processo cliente está requisitando e retorna o índice da sua **tabela de servidores em uso** que chamamos **descritor de servidor** (sd - *server descriptor*). O acesso ao servidor requisitado é feito, então, indiretamente através do **descritor de servidor**, assim como o acesso a um arquivo aberto é feito

pelo descritor do arquivo retornado pela chamada *open* do sistema de arquivos do UNIX. Caso ocorra algum problema que impeça o servidor de nomes de serviços tornar o servidor requerido disponível, um código de erro é retornado.

O servidor de nomes de serviços realiza a chamada ***get_service*** verificando, inicialmente, em sua **tabela de servidores não permanentes ativos** se o servidor requisitado está presente. Caso esteja, atualiza a **tabela de servidores em uso do cliente** (e requisita ao núcleo a realização da atualização da tabela correspondente por ele mantida) e incrementa o contador de usuários do servidor. Caso o servidor não esteja ativo, ele faz uma consulta à **tabela de servidores dinâmicos do sistema** verificando se o nome do servidor corresponde a algum servidor existente. Caso exista, o servidor de nomes de serviços faz sua ativação (cria um novo processo), aloca uma entrada na **tabela de servidores não permanentes ativos** e inicializa os seus campos. Além disto, ele atualiza a tabela de servidores em uso do cliente, requisita ao núcleo a realização da operação correspondente sobre suas tabelas e faz o contador de usuários do servidor igual a 1.

A forma de alocação de uma caixa postal ao servidor ativado por demanda é realizada por uma alteração da chamada de sistema *fork*, que passa a executar essa função adicional quando trata de processos criados pelo servidor de nomes de serviços.

A chamada ***free_service*** é executada por um processo cliente quando não mais deseja utilizar os serviços de um servidor anteriormente requisitado. Seu formato é o seguinte:

free_service(sd);

A chamada ***free_service*** tem como parâmetro o índice da **tabela de servidores em uso do cliente** (descritor do servidor). Ela é responsável por tornar o servidor indisponível para o processo cliente que a executa. O servidor de nomes de serviços realiza a chamada ***free_service*** liberando, inicialmente a entrada na **tabela de servidores em uso pelo cliente** indicada pelo **descritor do servidor** (sd) e solicita ao núcleo que o mesmo seja feito na estrutura correspondente que faz parte da tabela de processos. Caso o contador de usuários do servidor liberado seja igual a 1, a entrada correspondente da **tabela de servidores ativos** é desalocada e o servidor de nomes de serviços envia uma mensagem ao núcleo para desativar o processo servidor, uma vez que não há mais usuários para ele. Caso contrário, apenas o contador é decrementado.

Caso um processo cliente seja finalizado sem liberar alguns servidores não permanentes aos quais ele tinha acesso essa liberação é feita pela chamada *exit*, de modo análogo ao que ocorre com arquivos abertos e não fechados no sistema UNIX. A chamada *exit* deverá consultar a **tabela de servidores em uso pelo cliente** e comunicar ao servidor de nomes de serviços quais servidores dinâmicos devem ser liberados.

A criação de novos processos no sistema deve ser feita de modo que processos criados herdem os serviços providos por servidores não permanentes que estão sendo utilizados por seus processos pais.

6.2 Proposta de Implementação no MINIX

A implementação do servidor de nomes de serviços exige não só a codificação dos programas do novo servidor como também algumas alterações na estrutura do sistema para que ele passe a conviver com seus pares (gerente de memória e sistema de arquivos). O código das estruturas criadas assim como os programas componentes do servidor de nomes de serviços é inserido no novo diretório usr/src/ns. Além disto, o sistema deve ter seu processo de inicialização modificado para que esse novo servidor permanente seja ativado juntamente com o gerente de memória e o sistema de arquivos, compondo a mesma fila de escalonamento deles. Na seção 4.1.1 apresentamos a estrutura interna do sistema MINIX e aqui na figura 6-7 reapresentamos a estrutura com o servidor de nomes de serviços fazendo parte da estrutura do novo sistema. No novo sistema, mais processos comporão a coleção de processos servidores. O servidor de nomes de serviços será um servidor permanente assim como o sistema de arquivos e o gerente de memória. Outros servidores comporão a coleção de processos servidores não permanentes ou dinâmicos, como é o caso do servidor de compilação apresentado na figura 6-7.

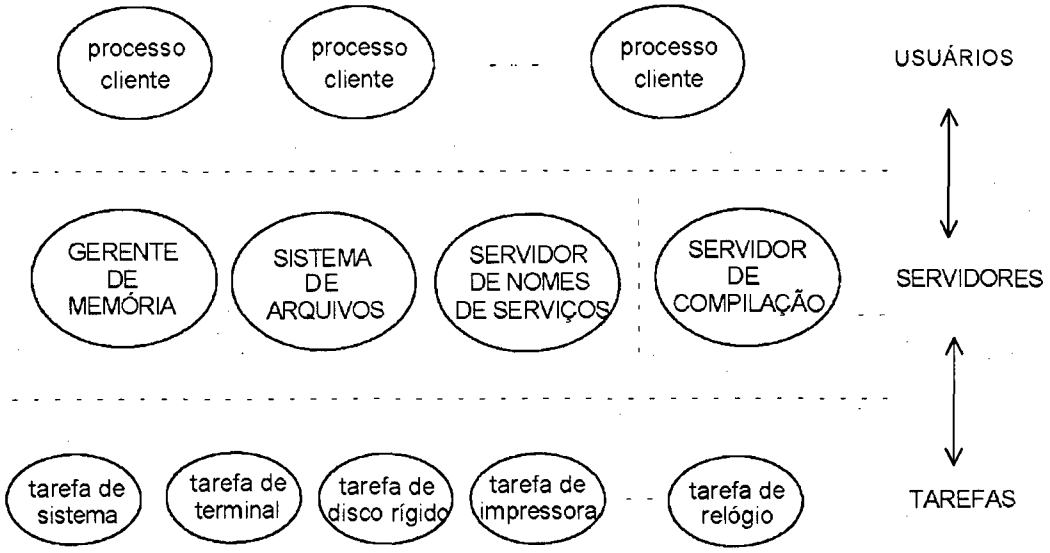

Figura 6-7. Estrutura do sistema com o servidor de nomes de serviços

6.2.1 Estruturas de Dados

Três estruturas são criadas para serem geridas pelo servidor de nomes de serviços: a **tabela de servidores não permanentes existentes** no sistema, a **tabela de servidores não permanentes ativos no sistema** e uma parte do **registro descritor de cada processo** que contém o vetor de descritores de servidores do processo (ver seção 6.1.2.1).

A **tabela de servidores não permanentes existentes** é uma lista na qual os nós são formados por dois campos *string*: um para o nome do serviço (char *service_name) e outro para o nome do arquivo que contém o código executável do servidor que provê aquele serviço (char *progr_server). Além destes, há um terceiro campo que é um ponteiro para a estrutura que permite o encadeamento da lista.

A **tabela de servidores não permanentes ativos** no sistema conterá MAX_DYN_SERVERS entradas, sendo cada uma alocada a um processo servidor não permanente que esteja ativo. Cada entrada da tabela é composta por um campo *string* para armazenar o nome do serviço correspondente (char *name_server) e dois campos inteiros

correspondentes à identificação do processo servidor (int server_pid) e o contador do número de usuários daquele serviço (int n_users). Uma entrada desta tabela será alocada a um novo processo servidor não permanente quando da sua ativação.

O arquivo usr/src/ns/name_server.h contém os códigos referentes às duas estruturas supracitadas que são incluídas no código do servidor de nomes de serviços por ocasião de sua compilação.

Analogamente ao que existe no gerente de memória e no sistema de arquivos, cria-se uma parte da tabela de processos na área do servidor de nomes de serviços no novo arquivo usr/src/ns/proc.h. Esta parte da tabela de processo contém os vetores de descritores de servidores de cada processo do sistema. Assim como os vetores correspondentes na parte da tabela de processos do núcleo, estes vetores são inicializados com os servidores permanentes a que os processos têm acesso por ocasião da sua criação. A tabela será uma estrutura com NR_PROCS entradas sendo cada entrada composta por um vetor de MAX_DYN_SERVERS entradas. Cada entrada deste vetor contém um ponteiro para uma entrada da tabela de servidores não permanentes ativos. As entradas desse vetor de descritores de servidores de um processo correspondem às mesmas entradas do vetor do registro descritor gerido pelo núcleo.

6.2.2 Programas e Funções

Alguns programas são alterados para que se possa incluir o servidor de nomes de serviços no sistema: o programa usr/src/kernel/main.c de inicialização do sistema, o programa usr/src/lib/other/syslib.c que capta as chamadas especiais para comunicação entre os servidores e o núcleo através da tarefa de sistema (*sys_task*()), além da própria tarefa *sys_task*() que é implementada no programa usr/src/kernel/system.c. O programa usr/src/kernel/main.c é alterado para ativar o servidor de nomes de serviços e inicializar o seu mapa de memória juntamente com os mapas do gerente de memória, do sistema de arquivos e do processo especial *init*. Além destas alterações, os programas que implementam as chamadas de sistema para criação (*fork*) e destruição (*exit*) de processos também sofrem alterações. Além disto, o programa que constrói o disco de carga do sistema usr/src/tools/build deve ser modificado para que seja adicionado o novo servidor.

O programa usr/src/lib/other/syslib.c captador das chamadas especiais é alterado com a inclusão de três novas entradas para atender às chamadas especiais provenientes do servidor de nomes de serviços para comunicação com o núcleo. Estas entradas são três novas rotinas de syslib.c com os nomes *sys_service_got*, *sys_service_freed*, e *sys_mbox_freed* que enviam mensagem ao núcleo via programa call.c que transforma a chamada em mensagem e a envia através primitiva **sendrec**. Os três novos tipos de operações que o servidor de nomes de serviços utiliza para comunicar certos eventos ao núcleo são: SYS_SGOT para comunicar que um determinado servidor foi tornado disponível a um cliente, SYS_SFREED para comunicar que um servidor foi tornado indisponível para um determinado cliente e SYS_MBFREED para comunicar que um determinado servidor não tem mais clientes e desta forma sua caixa postal deve ser desalocada.

Para atender a estes três tipos de requisição, três novas funções são implementadas em *sys_task*(): *do_service_got*, *do_service_freed* e *do_mbox_freed*. As duas primeiras são responsáveis pela alocação e liberação de entrada no vetor de descritores de servidores em uso do processo cliente no núcleo e a última pela liberação de caixa postal de determinado servidor. A função *do_service_got* recebe como parâmetros a identificação do processo cliente (cli_id), o descritor de servidor (sd) a ele alocado recentemente pelo servidor de nomes de serviços e a identificação do processo servidor (server_pid) que foi alocado ao cliente. Com estes parâmetros, a entrada de índice (sd) é alocada no vetor de descritores de servidores do processo no núcleo, sendo preenchida com a identificação da caixa postal do respectivo servidor (server_id). A função *do_service_freed* recebe como parâmetros a identificação do processo cliente (cli_id) e o descritor do servidor (sd) que está sendo liberado. Sua função é desalocar a entrada de índice sd do vetor de descritores de servidores do processo cliente. Por último, a função *do_mbox_freed* recebe como parâmetro a identificação do processo servidor (server_pid) que deve ter sua caixa postal liberada junto ao núcleo. Esta função atribui a constante FREE_MBOX ao ponteiro para a primeira requisição de comunicação da caixa postal daquele servidor.

O servidor de nomes de serviços que tem seu programa principal (usr/src/ns/main.c) esboçado de forma simplificada na figura 6-8 ativa a primitiva de comunicação **receive** recebendo a identificação do cliente. A mensagem que ele espera receber é uma estrutura formada por um campo inteiro (int option) que contém a opção de serviço, e uma união de estruturas que pode ser de três tipos: um campo ponteiro para um *string* que contém o nome do

serviço sendo requisitado, um inteiro contendo o descritor do servidor, ou ainda dois inteiros que contêm as identificações dos processos pai e filho. A mensagem de resposta é uma estrutura formada por um campo inteiro que conterá o descritor do servidor no caso de provimento de serviço. As estruturas referentes às mensagens são criadas no novo arquivo usr/src/ns/type.h que é incluído no arquivo usr/src/ns/main.c por ocasião de sua compilação. O servidor faz uso de quatro funções que são implementadas para atender às suas chamadas de sistema. As funções são *do_get_service*, *do_free_service*, *do_init_process* e *do_ns_exit* que tratam respectivamente do provimento e liberação de serviço, inicialização da entrada da parte da tabela de processo gerida pelo servidor do processo recém criado e da liberação de entrada nesta tabela quando da finalização do processo.

```
main ()
{
  struct message mess1, mess2;
  while ( TRUE ) {
        cli_id = receive( &mess1, size_mess1 );
        switch ( mess1.option ) {
          case ALLOCATE_SERVICE: mess2.sd = do_get_service(cli_id, mess1.server_name);
          case DEALLOCATE_SERVICE: mess2.sd = do_free_service(cli_id, mess1.sd);
          case INITIALIZE_PROCESS: do_init_process(mess1.parent_pid, mess1.child_pid);
          case EXIT_PROCESS: do_ns_exit(mes1.process_pid);
          default: mess2.sd = -1;
        }
        send( &mess2, size_mess2 );
  }
}
```

Figura 6-8. Servidor de Nomes de Serviços

As funções *do_get_service* e *do_free_service* recebem requisições de processos clientes enquanto *do_init_process* e *do_ns_exit* atendem requisições oriundas do gerente de memória.

A função *do_get_service* tem a finalidade de tornar o servidor disponível para o cliente. Ela recebe como parâmetros a identificação do cliente (cli_id) e o nome do serviço requisitado (service_name). O descritor do servidor é retornado ao cliente se o servidor pode ser tornado disponível; caso contrário, um código de erro é retornado. Para realizar suas atribuições, inicialmente a função verifica se o servidor que provê aquele serviço está ativo, consultando a tabela de servidores não permanentes ativos. Caso ele esteja presente na tabela, ela aloca uma entrada no vetor de descritores de servidores em uso daquele cliente, cujo índice passa a ser o descritor do servidor retornado ao cliente. Antes do retorno do descritor do servidor (sd), a função comunica ao núcleo, via tarefa de sistema (*sys_task*()), usando a rotina *sys_service_got*, a alocação ocorrida no vetor de descritores de servidores do processo, para que o núcleo faça o mesmo na parte do registro descritor de processo que lhe pertence. Caso o servidor não esteja ativo, a tabela de servidores não permanentes existentes é consultada para verificar se o serviço requisitado existe no sistema. Caso exista, o servidor a ele associado é ativado e seu registro descritor é inicializado na área do servidor de nomes de serviços, após o retorno da chamada *fork*, pela ativação da função *do_init_process*. Então, é alocada uma entrada na tabela de servidores ativos e é executado o procedimento correspondente ao caso do servidor já estar ativo. Caso o serviço não exista, um código de erro é retornado ao cliente.

A função *do_free_service* tem a finalidade de tornar o servidor indisponível ao cliente. Ela recebe como parâmetros a identificação do processo cliente (cli_id) e o descritor do servidor (sd) que está sendo liberado. Inicialmente, a função libera a entrada de índice sd do vetor de descritores de servidores do cliente. Em seguida, comunica ao núcleo, via tarefa de sistema (*sys_task*()), usando a rotina *sys_service_freed*, a liberação ocorrida para que o núcleo faça o mesmo na parte do registro descritor do cliente que fica sob seu controle. Então, a função verifica o valor do contador de usuários do servidor que está sendo liberado: se ele for igual a 1, o servidor de nomes de serviços solicita ao núcleo, através da tarefa de sistema, usando a rotina *sys_mbox_freed*, que a caixa postal do servidor seja desalocada, depois do que libera a entrada associada ao servidor na tabela de servidores ativos e, finalmente, envia uma mensagem ao núcleo

para que o servidor seja desativado; caso o contador de usuários seja maior que 1, a função somente o decrementa.

A função *do_init_process* tem a finalidade de inicializar o vetor de descritores de servidores da parte do registro descritor de um processo recém criado pertencente ao servidor de nomes de serviços. Ela recebe como parâmetros as identificações dos processos pai e filho. Uma nova entrada na tabela de processos do servidor de nomes de serviços é alocada ao novo processo e o conteúdo do vetor de descritores de servidores do processo pai é copiado para a área do filho que então torna-se herdeiro dos servidores utilizados por seu pai. O vetor de descritores de servidores do novo processo é então pesquisado. Para cada servidor não permanente a ele alocado, ou seja, herdado do seu processo pai, o contador de usuários deste servidor é incrementado de um.

A função *do_ns_exit* tem a finalidade de desalocar a entrada da tabela de processos referente a um processo que está sendo finalizado. Ela recebe como parâmetro a identificação do processo que está terminando. O vetor de descritores de servidores em uso do processo é pesquisado. Se houver alguma entrada alocada a algum servidor não permanente, ou seja, se houverem servidores não permanentes que não tenham sido liberados pelo processo, estes são liberados agora (para cada um deles, é executada a função *do_free_service*).

As chamadas de sistema para criação (*fork*) e término (*exit*) de processos, que são atendidas pelo gerente de memória, também sofrem modificações em função do novo servidor do sistema. A chamada *fork* causa a alocação de uma nova entrada na tabela de processos na área do gerente de memória, do sistema de arquivos e do núcleo, e a chamada *exit* realiza a liberação de tais entradas. A partir das alterações propostas, uma entrada na parte da tabela de processos gerida pelo servidor de nomes de serviços é também alocada para cada novo processo criado.

A chamada *fork* é atendida no gerente de memória pela função *do_fork* do programa usr/src/mm/forkexit.c que realiza sua parte (ver seção 5.2.2) e envia mensagem ao núcleo e ao sistema de arquivos comunicando a criação do novo processo. A partir daí, uma mensagem deve também ser enviada ao servidor de nomes de serviços, conforme veremos. A comunicação com o núcleo é feita pela chamada especial *sys_fork*, via tarefa de sistema. O núcleo faz sua parte através da função *do_fork* do programa usr/src/kernel/system.c. Esta função é

alterada com a inclusão de uma rotina que copia o vetor de descritores de servidores do processo pai para o registro descritor do processo filho. Assim, o vetor de descritores de servidores do filho é inicializado com o mesmo conteúdo do vetor do pai. Operação similar deve ser feita também no servidor de nomes de serviços como veremos a seguir. A comunicação do gerente de memória ao servidor de nomes de serviços para notificar a criação de um novo processo é feita de forma análoga àquela ocorrida com o sistema de arquivos. Uma nova chamada especial (*tell_ns*) é implementada no programa syslib.c de forma análoga à chamada *tell_fs* para a comunicação com o sistema de arquivos. Esta nova chamada especial é também transformada em mensagem no programa call.c e é enviada ao servidor de nomes de serviços que faz a inicialização do novo processo na sua parte do registro descritor pela função *do_init_process*. Esta comunicação só ocorrerá quando o processo pai do processo que está sendo criado não for o servidor de nomes de serviços. Isto porque durante a execução da chamada *fork* o servidor de nomes de serviços que está criando um novo processo, está bloqueado à espera da finalização da chamada. Ele próprio fará a inicialização do novo processo em sua área, pela ativação da função *do_init_process*, após receber a identificação do novo processo como retorno da chamada *fork*.

A chamada *exit* é atendida no gerente de memória pela função *do_mm_exit* do programa usr/src/mm/forkexit.c. Entre outras atividades, a função libera o espaço de memória ocupado pelo processo e comunica ao núcleo através da tarefa *sys_task*() a finalização do processo para que ele tome providências. Tais providências são efetivadas pela função *do_xit* que libera a entrada na tabela de processos, atribuindo a constante P_SLOT_FREE ao campo p_slot_free do registro descritor do processo alocado àquela entrada da tabela. A alteração da chamada *exit* consiste em inserir uma comunicação entre o gerente de memória e o servidor de nomes de serviços analogamente ao que é feito no sistema de arquivos. Nesta comunicação com o servidor de nomes de serviços, o gerente de memória comunica qual processo está sendo finalizado. Esta mensagem é atendida pela função *do_ns_exit*.

Após a eventual liberação de servidores não permanentes pendentes, realizada pela função *do_ns_exit*, a entrada da tabela de processos que era utilizada pelo processo finalizado é marcada como livre e o controle volta ao gerente de memória que então comunica ao núcleo, via tarefa de sistema, a desativação do processo.

As chamadas de sistema do MINIX têm os códigos de seus programas captadores no diretório usr/src/lib/posix. As duas novas chamadas criadas para serem atendidas pelo servidor de nomes de serviços terão seus programas captadores no mesmo diretório. As chamadas de sistema a serem atendidas pelos servidores ativados por demanda serão também colocadas neste diretório, funcionando de forma similar às demais. A diferença entre elas e as chamadas atendidas pelos servidores permanentes reside no fato delas terem como parâmetro o descritor do servidor (sd) além daqueles peculiares ao serviço solicitado. Cada captador de chamada de sistema para servidor não permanente direciona a mensagem para o servidor através do descritor de servidor recebido como parâmetro.

O servidor de nomes de serviços é ativado na inicialização do sistema juntamente com o gerente de memória e o sistema de arquivos. Todos eles são escalonados a partir da fila de prioridade intermediária. O programa kernel/main.c é então alterado para fazer esta ativação. Assim também o programa *build* que monta o disco de carga do sistema deverá preparar o disco com a adição do código binário do servidor de nomes de serviços.

6.3 Proposta de Implementação no Nó//

A inserção do servidor de nomes de serviços na máquina Nó// ocorre naturalmente sendo o servidor um processo adicional alocado a um nó de trabalho quando da ativação do sistema. As suas estruturas de dados são mantidas no nó onde ele executa e são manipuladas por ele. A inclusão de servidores por demanda ocorrerá da mesma forma que a inclusão de um processo qualquer, através da criação de um novo processo pela chamada de sistema *fork*. Desta forma, servidores demandados serão alocados a nós de trabalho havendo um controle de alocação de servidores no nó de controle para a associação de cada um deles a uma caixa postal. As estruturas de dados do núcleo relacionadas com o mecanismo de ativação de servidores (vetor de descritores de servidores por processo e tabela de caixas postais) são geridas no nó de controle.

7 CONCLUSÕES

7.1 Contribuições

Nosso trabalho concentrou-se no estudo de sistemas operacionais distribuídos ou paralelos para sistemas computacionais compostos por múltiplos processadores, com destaque para os aspectos de comunicação entre processos e provimento de serviços. Suas contribuições primordiais residem na concepção de um mecanismo original de comunicação, simples e eficiente, e na concepção de um modelo original de ativação de servidores por demanda em um sistema operacional. O mecanismo de comunicação proposto provê ao sistema a transparência de localidade de processos que é um elemento importante e desejável em sistemas operacionais distribuídos. O servidor de nomes de serviços proposto, por sua vez, provê os serviços do sistema a seus usuários baseado na transparência de localidade. Uma contribuição adicional foi o estudo de implementação do sistema MINIX sobre o Nó// através da distribuição efetiva de seus componentes.

O estudo bibliográfico serviu para detectar carências, as quais foram atacadas e resolvidas. Embora nossas propostas apresentem características que as diferenciam das citadas na literatura, acreditamos que o ponto forte das mesmas seja a utilidade prática dos resultados obtidos.

7.2 Sugestões de Trabalhos Futuros

As propostas de nosso trabalho poderão ser efetivamente realizadas quando for concretizada a construção do primeiro protótipo do multicomputador Nó//.

Como sugestões para trabalhos futuros, sugerimos a utilização de nossas propostas no novo sistema operacional voltado ao modelo cliente-servidor que está sendo concebido para o multicomputador Nó//.

BIBLIOGRAFIA

[AND75] ANDERSON, George A.; JENSEN, E. Douglas; **Computer Interconnection Structures: Taxonomy, Characteristics, and Examples**. Computing Surveys, Vol. 7, No. 4, Dezembro de 1975.

[ARI90] BEN-ARI, M.; **Principles of Concurrent and Distributed Programming**. Prentice Hall, Englewood Cliffs, 1990.

[ATH88] ATHAS, William C.; SEITZ, Charles L.; **Multicomputers: Message-Passing Concurrent Computers**. 0018-9162/88/0800-0009 IEEE, Agosto de 1988.

[AUS91] AUSTIN, Paul; MURRAY, Kevin; WELLINGS, Andy; **The Design of an Operating System for a Scalable Parallel Computing Engine**. York (Inglaterra), SOFTWARE-PRACTICE AND EXPERIENCE, vol. 21(10), 989-1013, outubro de 1991.

[BIR84] BIRREL, Andrew D.; NELSON, Bruce Jay; **Implementing Remote Procedure Calls**. ACM Transactions on Computers Systems, Vol. 2, No 1, Páginas 39-59, fevereiro de 1984.

[COR93] CORSO, Thadeu B.; **Ambiente para Programação Paralela em Multicomputador**. Relatório Técnico No 1. Depto de Informática e Estatística - UFSC, Florianópolis, Dezembro de 1993.

[CRI88] CRICHLOW, Joel M.; **An Introduction to Distributed an Parallel Compunting**. Prentice Hall, Englewood Cliffs, 1988.

[DUN90] DUNCUN, Ralph; **A Survey of Parallel Computer Architectures**. Control Data Corporation, IEEE 0018-9162/90/0200-0005, Fevereiro de 1990.

[FEN81] FENG, Tse-yun; **A Survey of interconnection Networks**. The Ohio State University, IEEE, EH0217-0/84/0000/0005, 1981.

[GAR87] GARNET, N. H.; **Helios - An Operating System for the Transputers**, VII Conference Occan Users Group, pp. 411-419, Setembro de 1987.

[FLY72] FLYNN, M. J.; **Some Computer Organizations and Their Effectiveness**. IEEE Trans. on Computers, vol. C-21, pp. 948-960, Setembro de 1972.

[FRI89] FRICKS, Ricardo M.; **Projeto de um Sistema Operacional Tempo-Real Baseado no MINIX**. Dissertação de Mestrado, CPGCC-UFRGS, Porto Alegre, Abril de 1989.

[HAN87] HANSEN, Brinch; **A Joyce Implementation**, Computer Science Department, University of Copenhagen, Universitetsparken I, SOFTWARE-PRACTICE AND EXPERIENCE, vol. 17(4), 267-276, Abril de 1987

[HOL78] HOLT, R.C.; GRAHAM, G. S; LAZOWSKA, E. D.; SCOTT, M.A.; **Structured CONCURRENT PROGRAMMING with Operating Systems Applications**. Addison-Wesley Publishing Company, Massachusetts, 1978.

[HWA85] HWANG, Kai; BRIGGS, Fayé A.; **Computer Architeture and Parallel Processing**. McGraw-Hill Book Company, Singapore, 1985.

[IAB959] IAB - RFC 959 - **File Transfer Protocol**. Network Working Group; Postel J., Reynolds J., ISI, October, 1985.

[KER78] KERNIGHAN, Brian W.; RITCHIE, Dennis M.; **The C Programming Language**. Englewood Cliffs, Prentice Hall, 1978.

[NOR88] NORTON, Peter; Socha, John; **Linguagem Assembler para IBM PC**. Editora Campus, Rio de Janeiro, 1988.

[OLI 92] STEIN, Benhur de Oliveira; **Projeto do Núcleo de um Sistema Operacional Distribuído**. Dissertação de Mestrado, CPGCC-UFRGS, Porto Alegre, Fevereiro de 1992.

[OUS79] OUSTERHOUT, John K.; SCEIZA, Donald A.; SINDHU, Pradeep S.; **Medusa: An Experiment in Distributed Operating Structure (Summary)**. ACM 0-89791-009-5/79/1200/0115, 1979.

[PAZ91] PAZZINI, Marcelo; **Especificação do TRIX: Um Sistema Operacional Multiprocessado para Transputers**. Trabalho Individual, CPGCC-UFRGS, Porto Alegre, Outubro de 1991.

[RED80] REDELL, David D.; DALAL, Yogen K.; HORSLEY, Thomas R.; LAUER, Hugh C.; LYNCH, William C.; McJONES, Paul R.; MURRAY, Hal G.; PURCELL, Stephen C.; **Pilot: An Operating System for a Personal Computer**. Xerox Business Systems; ACM 0001-0782/80/0200-0081, 1980.

[REE87] REED, D. A., FUJIMOTO, R. M.; **Multicomputer Networks: Message-Based Parallel Processing**. MIT Press, 1987.

[SIE79] SIEGEL, Howard J.; MCMILLEN, Robert J.; MUELLER, Philip T. Jr; **A Survey of Interconnection Methods for Reconfigurable Parallel Processing Systems**. Puudue University, West Lafayette, Indiana, National Computer Conference, 1979.

[SIL91] SILBERCHATZ, Abraham; PETERSON, James L.; GALVIN, Peter B.; **Operating System Concepts**. Addison-Wesley Publishing Company, USA, 1991.

[SOL91] SOLOMON, Marvin H.; FINKEL, Rafael A.; **The Roscoe Distributed Operating System**. University of Wisconsin - Madison, ACM 0-89791-009-5/79/1200/0108, 1979.

[SUN90] Sun Microsystems; **SunOS Reference Manual**, vol I, pp. 573-576, USA, 1990.

[TAN87] TANENBAUM, Andrew S.; **Operating Systems: Design and Implementation**. Prentice Hall, Englewood Cliffs, 1987.

[TAN88] TANENBAUM, Andrew S.; **Computer Networks**. Prentice Hall, Englewood Cliffs, 1988.

[TAN92] TANENBAUM, Andrew S.; **Modern Operating Systems**. Prentice Hall, Englewood Cliffs, 1992.

[TOS87] TOSCANI, Simão; **Introdução aos Sistemas Operacionais**. Curso de Pós-Graduação em Ciências da Computação - UFRGS, Porto Alegre, Março de 1987.

[WIT81] WITTIE, Larry D.; **Communication Structures for Large Networks of Microcomputers**. EHO217-0/84/0000/0291, IEEE, 1981.

[YOU87] YOUNG, Michael; TEVANIAN, Avadis; RASHID, Richard; GOLUB, David; EPPINGER, Jeffrey; CHEW, Jonathan; BOLOSKY, William; BLACK, David; BARON, Robert; **The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System**. Carnegie-Mellon University, ACM 089791-242-X/87/0011/0063, 1987.