UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TRIX, Um Sistema Operacional Multiprocessado para Transputers, com Gerência Distribuída de Processos

por

Marcelo Pasin

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

Prof. Philippe Navaux
Orientador

Prof. Thadeu Botteri Corso
Co-orientador

Porto Alegre, janeiro de 1994

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Pasin, Marcelo

TRIX, Um Sistema Operacional Multiprocessado para Transputers, com Gerência Distribuída de Processos / Marcelo Pasin. – Porto Alegre: CPGCC da UFRGS, 1994.

118 p. : il.

Dissertação (mestrado)—Universidade Federal do Rio Grande do Sul, Curso de Pós-Graduação em Ciência da Computação, Porto Alegre, 1994. Orientador: Navaux, Philippe; Co-orientador: Corso, Thadeu Botteri

Dissertação: Sistemas Operacionais Distribuídos
Sistemas operacionais, transputers, sistemas distribuídos, gerência de processos

AGRADECIMENTOS

O assunto desenvolvido neste texto é o resultado de alguns anos de trabalho, durante os quais contei com o apoio de diversas pessoas. Gostaria de agradecer, de maneira sincera:

- Ao professor Philippe Navaux pela seu total apoio ao projeto TRIX, à sua confiança nele depositada, pelo suporte técnico, pela paciência na espera de resultados e pelo apoio psicológico nos momentos difíceis;
- Ao professor Thadeu Botteri Corso, por seu suporte técnico no desenvolvimento do sistema e por sua acolhida em seu grupo de trabalho;
- Ao professor Hermann Lücke, que incondicionalmente forneceu recursos de seu laboratório para que este trabalho pudesse ser feito;
- À minha querida Simone, pela sua insubstituível companhia nos melhores momentos que vivi durante a confecção deste trabalho, contando sempre com muita paciência nos outros momentos, nos quais me fiz ausente por causa do trabalho;
- Ao colega Benhur Stein pela sua constante e impagável parceria nos momentos de discussão sobre assuntos do trabalho bem como nos de descontração, sem os quais este trabalho não seria possível;
- Aos amigos Jáder Brilhante e Alceu Frigeri, que em tempos diferentes propiciaram agradável companhia, tornando muito mais humana a minha vida fora do laboratório;
- Ao Departamento de Eletrônica e Computação da Universidade Federal de Santa Maria, pela oportunidade que me deu de aprimorar os meus conhecimentos e avançar em minha carreira docente;
- À CAPES pelo auxílio através de bolsa PICD.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

O trabalho em torno do sistema TRIX visa desenvolver um sistema operacional multiprocessado experimental, para servir de base a futuros trabalhos de pesquisa em sistemas operacionais e processamento paralelo. Como características essenciais do sistema têm-se simplicidade, desempenho e compatibilidade com UNIX. Com o sistema operando em vários processadores, pode-se fazer experiências de muitos tipos em processamento paralelo, como por exemplo, ensaios de distribuição de carga, avaliações de desempenho, implementação de linguagens paralelas ou distribuídas, bancos de dados, sistemas dedicados, etc.

Ao construir o sistema TRIX, decidiu-se usar o código fonte de um sistema já pronto e funcionando, para encurtar o tempo de desenvolvimento. Optou-se pelo MINIX, que é um sistema bem estruturado, dividido em camadas de processos seqüenciais que se comunicam por troca de mensagens. Mais que isso, o MINIX é compatível com o UNIX versão 7 e o seu código fonte pode ser usado para fins acadêmicos sem ferir o seu *Copyright*. Foi necessário porém dotar o MINIX de características distribuídas, para o controle de uma arquitetura multiprocessada. O projeto e implementação destas características é o assunto deste trabalho.

Inicialmente são avaliados os sistemas operacionais multiprocessados encontrados na literatura, de maneira a auxiliar no projeto do TRIX. Entre os detalhes estudados estão as diferenças entre sistemas paralelos e distribuídos, a comunicação entre seus processos e sua forma de distribuição.

São apresentados os microprocessadores do tipo transputer, com seus mecanismos de criação e comunicação de processos implementados em *hardware*. Também é detalhado o sistema operacional MINIX, com sua construção em camadas, com processos servidores e clientes e suas estruturas de controle.

O sistema foi projetado tentando maximizar a sua flexibilidade, escalabilidade e disponibilidade e minimizar sua complexidade. Os processos servidores são divididos de forma fixa, controlando as funções de memória localmente e as funções de arquivos remotamente. Os processos de usuário são distribuídos tentando se equalizar a taxa de ocupação dos processadores do sistema.

É implementado um mecanismo de identificação global de processos juntamente com um método transparente de comunicação. Isso é feito junto com a criação de um *driver* suplementar ao sistema, responsável pela comunicação entre processos de processadores distintos.

O núcleo original do MINIX recebeu uma série de alterações para suportar as características distribuídas do novo sistema, de maneira a identificar processos locais e remotos. Foram extraídas do núcleo as funções executadas pelo escalonador em *hardware* do transputer. Foram inseridas algumas funções novas para tratar de algumas idiosincrasias do transputer, como por exemplo, sua falta de gerência de memória.

Finalmente, foi criado um mecanismo distribuído para controlar a árvore de processos do sistema, mantendo a semântica do UNIX. A escolha do processador para a execução de novos processos é feita pelo processador que o estiver criando, através de informações recebidas sobre processadores menos ocupados que estiverem mais próximos.

PALAVRAS-CHAVE: Sistemas operacionais, transputers, sistemas distribuídos, gerência de processos.

TITLE: "TRIX, A TRANSPUTER BASED MULTIPROCESSOR OPERATING SYSTEM, WITH DISTRIBUTED PROCESS MANAGEMENT."

ABSTRACT

The TRIX system has been defined as an experimental operating system, to be used in future research on operating systems and parallel processing. The essential characteristics of the system are simplicity, performance and UNIX compatibility. With the system running on several processors, several topics can be studied, such as load balancing, performance evaluation, implementation of distributed or parallel programming languages, distributed data bases and embedded systems.

While designing the TRIX system, it was decided to use the source code of an existing system, to shorten the development time. MINIX was chosen, because it is a well structured system, divided into layers of sequential processes, which communicate via messages. MINIX is compatible with UNIX version 7 and its source code can be used for academic purposes without any copyright infringiment. It had to be extended to control a multiprocessor architecture. The implementation of these characteristics is the main subject of this work.

Initially some existing multiprocessor operating systems are evaluated, in order to guide in the development of TRIX. The studied issues are differences between parallel and distributed systems, interprocess communication features and the process distribution policy.

The transputer microprocessors are presented, with their hardware implemented devices for process creation and communication. The MINIX operating system layered structure is described, with client and server processes and their control structures.

The system is projected as an attempt to maximize flexibility, scalability and availability and to minimize complexity. The server processes are distributed in a fixed layout, controlling the memory and processes locally, and the files remotely. The user processes are distributed with load balancing among the processors.

Global identification and transparent communication are implemented. This is achieved with an additional device driver, responsible for communication between processes running on different processors.

The original MINIX kernel was changed to support the new distributed features of the system, identifying processes globally. The scheduler functions were stripped off, as they are now performed by the transputer hardware. Some functions were inserted to deal with some transputer problems, as for example, the lack of a memory management unit.

Finally, a mechanism to control the global system process tree was created, maintaning strict UNIX semantics. The choice of which processor will hold a newly created process is made by the processor creating it, based on information received about the nearer processor with a low load.

KEYWORDS: Operating systems, transputers, distributed systems, process management.

1 INTRODUÇÃO

Nos dias de hoje é possível encontrar sistemas digitais programáveis instalados em quase qualquer atividade de nossa sociedade. Tais sistemas funcionam de forma bastante independente e desconexa. É fácil porém prever que cada vez mais serão interligados estes sistemas de computação, de forma a melhor aproveitar a informação neles armazenada. Em um sistema totalmente interligado, podem ser compartilhados todos os seus dados e recursos, diminuindo de forma organizada a repetição de informações hoje existentes.

As sofisticações tecnológicas necessárias para aproveitar os recursos distribuídos de forma otimizada são extremamente complexas, pois dependem de mecanismos de comunicação rápida, de tolerância a falhas e de acesso controlado às informações. Não obstante, são necessários computadores de muito alto desempenho em certos pontos do sistema, para que os tempos de resposta de aplicações computacionais complicadas gerem resultados em tempo hábil para serem aproveitados.

A construção de sistemas de alto desempenho é um dos principais ramos da pesquisa em arquitetura de computadores. Pode-se notar hoje a existência de diversas formas sofisticadas de construção de computadores, como computadores de conjunto reduzido de instruções (RISC) e processadores vetoriais. Embora altos investimentos sejam feitos a nível de eletrônica, o que faz estas novas arquiteturas alcançarem desempenhos tão altos é a execução de mais de uma parte das tarefas ao mesmo tempo, ou seja, em paralelo.

Os dois paradigmas que regem os estudos para construção dos complexos sistemas de computação que já começam a aparecer em nossa sociedade são: *processamento paralelo* e *processamento distribuído*. Ambos possuem uma vasta área de intersecção, pois estudam o acontecimento de mais de um evento por

unidade de tempo. A diferença principal entre os dois paradigmas é o grau de interconexão entre as atividades de processamento.

Pensando em contribuir para a construção dos futuros sistemas de computação, o Instituto de Informática da Universidade Federal do Rio Grande do Sul possui um grupo de pesquisa em processamento paralelo e distribuído. Deste grupo fazem parte mais de vinte pessoas, entre professores, pesquisadores associados, alunos de mestrado e auxiliares de pesquisa, produzindo trabalhos e publicações em diversos níveis e sobre diversos tópicos.

1.1 Sistema Operacional TRIX

Com o aparecimento de máquinas com características de processamento paralelo no Instituto de Informática se notou a falta de um sistema operacional que tirasse proveito de sua capacidade. Foi assim com os Projetos M3P [CAR89] e DIX [BAR90], se repetindo com a aquisição de módulos processadores baseados em transputers. Para estes últimos, existiam dois ambientes de programação, um baseado na linguagem occam [INM88b] e outro com um compilador C, e nos dois as primitivas de sistema operacional eram executadas por processo servidor no computador hospedeiro dos módulos.

Foi iniciado em meados de 1990 um projeto de nome TRIX, com o objetivo de construir um sistema operacional para máquinas baseadas em transputers, que executasse nestes e não no hospedeiro as tarefas mais importantes. No projeto deste sistema [PAS91], foi decidido que o mesmo seria compatível com o UNIX e teria recursos para distribuição da carga de processamento aos processadores sob controle do sistema. Com um sistema deste tipo, onde existe conhecimento sobre seu funcionamento interno e se tem acesso ao seu código fonte, é possível desenvolver vários estudos nas áreas de sistemas operacionais para máquinas paralelas ou distribuídas.

Uma das premissas básicas deste projeto foi que o sistema operacional devia ser compatível com um sistema popular, para se ter maior disponibilidade de *software*. Dos sistemas operacionais em uso hoje, é incontestável a popularidade do MS-DOS, do VAX/VMS ou do VM/SP, além de outros. Porém, todos eles são atrelados a arquiteturas específicas, respectivamente IBM-PC, DEC-VAX e IBM/370.

O UNIX foi escrito em linguagem C e possui seu código portado para inúmeras arquiteturas, inclusive as já citadas, como nos sistemas XENIX, ULTRIX e AIX. É um sistema multiusuário, multitarefa e de propósito geral, extremamente popular no meio acadêmico. A opção pelo UNIX é hoje um senso comum, porém seu código é extremamente extenso e caro. Embora fosse possível escrever um sistema como o UNIX a partir do nada, isso não traria muita contribuição à pesquisa neste assunto, além de necessitar de muito tempo e esforço humano.

A solução mais viável foi usar o MINIX, que é um sistema operacional compatível com o UNIX. O código fonte do MINIX está publicado em um livro [TAN87], e pode ser usado para fins acadêmicos sem ferir o seu *Copyrigth*. É um sistema baseado em camadas de processos que se comunicam pela troca de mensagens, à imagem dos sistemas operacionais mais modernos da atualidade [RAS86, MUL90, POU91a]. Isto é uma vantagem inclusive sobre o próprio UNIX, pois tais técnicas de programação de sistemas não foram usadas quando este foi escrito.

Foi implementado então um sistema operacional multiprocessado para controlar um grupo de processadores do tipo transputer, baseado no MINIX. Supõe-se que, graças à arquitetura do transputer, este sistema deve ter sensíveis ganhos em desempenho em relação ao MINIX original. Isto se deve à capacidade do transputer escalonar processos e enviar mensagens através de instruções específicas para tal fim. Estas características também permitem que a parte escrita em linguagem *assembly* do MINIX desapareça quase que por completo, o que facilita sua alteração e depuração.

1.2 Organização deste Volume

O texto que segue nos próximos capítulos tem a intenção de descrever como foi implementado o sistema. Detalhadamente aparecem todos os mecanismos de distribuição do sistema, com relação a processos, mensagens e demais recursos do sistema. A seguir será dada uma breve descrição de cada um dos capítulos que compõem este trabalho, de forma a auxiliar a sua consulta no futuro.

No capítulo 2 é feito um estudo sobre sistemas operacionais multiprocessados, comparando-os entre si na medida do possível. Dentro deste estudo são discutidos os termos que definem sistemas operacionais como paralelos ou distribuídos, ressaltando suas diferenças. Após isto é feito um tratado de como a comunicação entre os processos é implementada nos diversos sistemas existentes. Por fim, são estudados tópicos sobre as formas de distribuição dos processos.

Microprocessadores do tipo transputer são apresentados no terceiro capítulo. São mostrados alguns detalhes de sua arquitetura interna para facilitar as explicações dos capítulos finais. Em detalhe aparecem as instruções de criação, término e comunicação de processos do escalonador do transputer.

No capítulo 4 aparece uma breve descrição do sistema operacional MINIX. É discutido o fato de ser um sistema construído em camadas, seu principais processos servidores, juntamente com algumas estruturas de controle.

O projeto do sistema TRIX é apresentado no quinto capítulo. Baseado nos objetivos do sistema e na arquitetura a ser suportada, são estipulados quais processos devem executar em quais processadores. Logo após, é descrito o identificador global de processos e um método transparente de comunicação. Também são descritas as alterações necessárias aos processos do MINIX para suportar o novo sistema. Aparece ainda o projeto de um processo para comunicação interprocessador e de um mecanismo para distribuir processos de usuário pelo sistema, na tentativa de equalizar a taxa de ocupação dos processadores.

O sexto capítulo descreve em detalhes como foi implementado o sistema. O sétimo e último capítulo faz uma análise dos resultados obtidos e aponta algumas direções nas quais o sistema pode evoluir no futuro.

2 SISTEMAS OPERACIONAIS MULTIPROCESSADOS

Dominar o conhecimento sobre sistemas de computação baseados em diversos processadores é hoje um grande desafio. Há tratados recentes em quase qualquer área da Ciência da Computação que defrontam esse novo paradigma, desde a arquitetura de computadores até inteligência artificial ou bancos de dados. Este capítulo faz uma discussão sobre as principais características dos sistemas multiprocessados, dando uma conceituação teórica básica para os próximos capítulos.

Os sistemas de computação multiprocessados podem ser classificados de diversas formas, dependendo de qual característica é levada em consideração. A mais tradicional é a proposta por Flynn [FLY66], onde o que caracteriza os sistemas é o número de fluxos de instruções e de dados. Através desta classificação, os sistemas podem ter um ou vários fluxos de instruções (*Single* ou *Multiple Instruction stream*) e um ou vários fluxos de dados (*Single* ou *Multiple Data stream*), sendo denominados como SISD, SIMD, MISD ou MIMD. Sistemas monoprocessados tradicionais são facilmente reconhecidos como sendo do tipo SISD. Os exemplos mais comuns de processadores SIMD são os processadores vetoriais. Raramente um sistema se encaixa na classificação MISD, pois é muito raro um processador executar vários fluxos de instruções sobre um único fluxo de dados. Os sistemas MIMD são bastante abrangentes, perfazendo todos os sistemas com diversos processadores independentes, processadores superescalares, etc.

Existe uma classificação mais abrangente [TAN92], onde os sistemas MIMD são divididos em **multiprocessadores** e **multicomputadores**, cada um dos quais subdivididos em **barramento** ou **comutado**, dependendo da forma de ligação entre os elementos processadores. Um sistema multiprocessador é formado por diversos elementos processadores independentes e interligados em seu

interior. Em contrapartida, um sistema multicomputador é formado por diversos computadores, cada qual com um elemento processador, interligados através de uma rede local.

A diferença principal entre estes dois sistemas é a capacidade de transmitir rapidamente dados de um elemento processador para outro, ou o nível de acoplamento destes elementos. Assim, em um sistema multiprocessador diz-se que o *hardware* é fortemente acoplado (*tightly coupled*) e em um sistema multicomputador é fracamente acoplado (*loosely coupled*).

Os sistemas operacionais multiprocessados podem ser paralelos ou distribuídos, dependendo do grau de acoplamento entre os elementos processadores [TAN92]. No caso de sistemas operacionais comerciais que controlam multiprocessadores simétricos, o acoplamento é máximo pois os processadores compartilham toda a sua memória. Já sistemas operacionais como o Amoeba [MUL90] ou Mach [RAS86], que controlam um grupo de processadores ligados por uma rede, o acoplamento é mínimo. Ambos exemplos são fáceis de classificar como paralelo e distribuído respectivamente.

Os sistemas operacionais multiprocessados em que o meio de acoplamento entre os processadores é uma rede podem ainda ser classificados como **sistema operacional de rede** ou **sistema operacional distribuído** [STA84]. A diferença está na maneira como o sistema é construído. No primeiro caso, o sistema é composto por diversos computadores individuais, conectados através da adição de mecanismos de comunicação em uma camada sobre seu sistema original. Exemplos deste tipo de sistema são DACNOS [GEI90], Unix United [BLA86] e Sprite [OUS88].

No segundo caso, o sistema operacional que controla todos os processadores é um só, sendo concebido com o propósito de interconectar processos em processadores diferentes de forma transparente. A rede de computadores parece ser um computador único e as fronteiras existentes entre as máquinas são

escondidas. Programas e arquivos são alocados ou movidos dinamicamente, sem interferência no seu funcionamento. Exemplos de sistemas distribuídos são o Locus [POP81, WAL83], Mach [RAS86] e Amoeba [MUL90].

2.1 Sistemas Paralelos × Distribuídos

Os sistemas operacionais que controlam máquinas multiprocessadas, como na figura 2.1, são chamados de Sistemas Operacionais Paralelos [TAN92]. Nesses sistemas, uma fila de processos em execução é sistematicamente distribuída entre os processadores disponíveis. Cada vez que tiver seu processo bloqueado, um processador recebe um novo processo, retirado de uma única fila. Enquanto existirem processos prontos, todos os processadores serão mantidos ocupados.

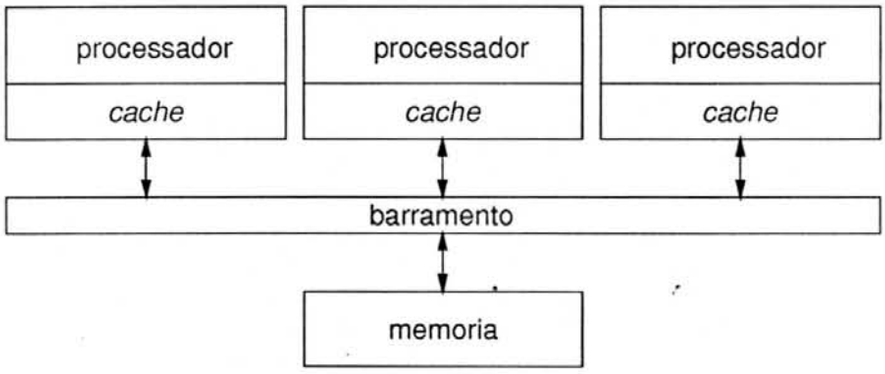

Figura 2.1: Sistema multiprocessado

Não é necessário alocar de maneira fixa um processo a um dado processador porque todos processadores têm acesso à mesma memória. Quando um processador receber um processo qualquer da fila, passa a executá-lo em sua própria memória, que, ao mesmo tempo, é a memória de todos os processadores do sistema. Esta memória compartilhada permite que exclusão mútua seja facilmente implementada com semáforos ou variáveis de controle.

A comunicação entre os processos é implementada usando trechos de memória compartilhados ou cópias de memória. Quando um processo desejar enviar uma mensagem, deverá acionar o núcleo com um pedido. O núcleo mantém tal processo bloqueado até que a mensagem seja entregue, ou copiada para um *buffer*. Quando o processo interlocutor desejar receber uma mensagem, também aciona o núcleo. O mesmo bloqueia o receptor se o transmissor ainda não houver enviado nenhuma mensagem. Se já houver, a mensagem é copiada para o receptor, que não bloqueia.

Como todos processadores têm acesso a toda memória de forma comum (ou mesmo parte dela) nenhuma mensagem é transportada explicitamente de um processador para outro. Isto na realidade é feito copiando a mensagem do espaço de endereçamento de memória de um processo para outro.

Os processadores de um sistema distribuído não possuem nenhum trecho de memória em comum e estão conectados por algum tipo de rede de comunicação, conforme a figura 2.2. Este tipo de sistema é chamado de **multi-computador** [TAN92]. Um sistema operacional pode ser considerado distribuído quando cada processador possuir gerência de memória e de processador locais, mas processos, proteção e comunicação são controlados globalmente [CHA90].

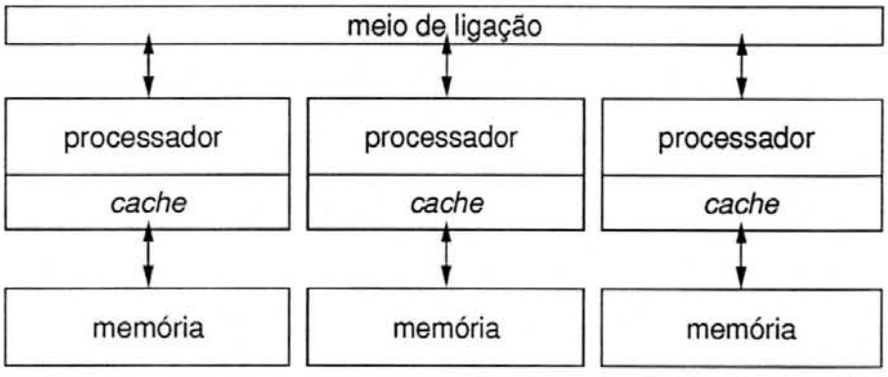

Figura 2.2: Sistema distribuído

Pela rede de conexão trafegam todas as mensagens entre os processos dos diversos processadores. A rede de conexão usualmente insere uma incerteza

na transferência das mensagens, por estar exposta a problemas físicos e a acessos não autorizados. Para tanto, são necessários protocolos de comunicação que identifiquem origem e destino de mensagens, façam verificação de erros e possuam controle de acesso [MAS87].

Um processo deve ser alocado a um dado processador de maneira mais perene que em sistemas paralelos. Trocar o processador que executa um processo significa usar a rede de comunicação que existe entre os processadores para transferir todo o conteúdo da memória do processo de um lado para outro.

O sistema operacional implementado no decorrer deste trabalho tem algumas características de sistemas paralelos e distribuídos ao mesmo tempo. Todos os processadores estão bastante próximos fisicamente e a rede de comunicação entre eles é confiável e segura. Sua conexão é feita fisicamente, processador a processador. Além disso, em caso de falhas no sistema, como panes elétricas, por exemplo, provavelmente todo o conjunto de processadores deixe de funcionar ao mesmo tempo. O argumento mais forte para classificá-lo como um sistema distribuído é que não há memória comum entre os processadores do sistema. Como se pode identificar características dos dois tipos de sistema, foi usada para o TRIX a denominação de sistema operacional **multiprocessado**.

2.2 Comunicação entre Processos

Nos sistemas operacionais modernos, a interação entre processos se dá através de mensagens [TAN92]. Dentro de uma mensagem um processo pode enviar ou receber dados de outro, sem que este precise acessar o espaço de endereçamento daquele. Normalmente existem primitivas do núcleo do sistema operacional através das quais um processo expressa a sua vontade de enviar ou receber alguma mensagem. O núcleo do sistema toma o controle do processador

neste instante e faz a cópia da mensagem do espaço de endereçamento do processo remetente para o receptor.

Como a comunicação entre processos através de mensagens tem a reputação de ser lenta, devido a cópias de memória e trocas de contexto, o sistema V propõe uma maneira de enviar mensagens através de registradores [CHE88]. Microprocessadores do tipo transputer possuem instruções implementadas em seu próprio *hardware*, também com o intuito de acelerar a troca de mensagens [INM87].

Em sistemas distribuídos, os processos que não executam no mesmo processador não possuem memória em comum. Para que uma mensagem seja enviada corretamente é necessária a sua transferência através da rede de interconexão dos módulos processadores [TAN85]. Isto pode ser feito pelo próprio núcleo do sistema ou por um processo dedicado especificamente para transferência de mensagens.

Para que o núcleo do sistema saiba se a mensagem tem destino local ou deve ser transferida pela rede, o processo destinatário precisa ser identificado como sendo local ou remoto. Para isso, os identificadores de destinatários de mensagens precisam de alguma forma carregar a informação de **onde** está o processo. Esta informação pode estar dentro do próprio identificador ou em tabelas do núcleo do sistema. A última opção normalmente é mais utilizada por proporcionar transparência de localidade, permitindo mudar a localização de processos sem que os outros que com ele se comunicam precisem ser alterados.

No sistema operacional Helios [HEL89], as mensagens são enviadas através de portas e têm 3 partes: cabeçalho, vetor de controle e vetor de dados, enviados de endereços separados e em tempos distintos. Na versão para transputers, mensagens locais são trocadas diretamente entre os dois processos e mensagens remotas são enviadas pela rede de interconexão. Há um processo chamado *Guardian*, que recebe mensagens da rede de interconexão. Se o processador

local estiver entre a mensagem e o processo destinatário a mensagem é retransmitida na direção do mesmo. Portas remotas são mapeadas por *surrogate ports*, que representam portas em um processador vizinho. Se a mensagem precisar ir adiante, haverão subseqüentes *surrogate ports*. Quando uma mensagem chega em um *Guardian*, cria-se uma *surrogate port* no sentido contrário, para uma resposta poder voltar. Ao enviar uma resposta, os *surrogate ports* podem ser apagados caso se deseje.

O sistema operacional Amoeba é baseado em objetos nomeados através de capacidades, que são controlados por servidores [MUL90, MUL88, TAN90]. Tais capacidades, além de dar nomes aos objetos, contêm as suas permissões de acesso e o número de uma porta para comunicação com seu servidor. A comunicação entre os processos é feita por RPCs (*Remote Procedure Calls*), usando três primitivas: `do_operation`, `get_request` e `send_reply`. A primeira é usada por clientes e as outras duas por servidores. Se no momento de uma RPC o núcleo não conhecer o servidor que atende à porta dada, é feito um *broadcast*. O núcleo não interfere na comunicação, deixando ao servidor a tarefa de verificar as capacidades enviadas por clientes de maneira a restringir seu acesso.

Já a comunicação no sistema Charlotte não bloqueia os processos interlocutores nas trocas de mensagens [ARS87]. Ainda assim, o núcleo do sistema não possui *buffers* para mensagens não entregues, sob o pretexto de que isto pode levar a *deadlocks*. Isto deixa o encargo de controlar a transmissão das mensagens aos próprios processos que estão se comunicando. O mecanismo de comunicação acaba por ser extremamente flexível, permitindo até que se use *rendezvous*, que mensagens sejam canceladas ou recebidas seletivamente.

2.3 Distribuição da Carga de Processamento

Para que se possa fazer uma distribuição equânime da carga de processamento do sistema, é necessário antes poder-se medir a carga de cada processador. Para isso normalmente se usam medidas percentuais, relativas ao máximo que cada sistema individualmente suporta. Embora a carga do processador possa ser medida de diversas maneiras [TAN92], a forma mais usual é a taxa de ocupação do processador, medida em intervalos fixos de tempo.

Quando um processador estiver sobrecarregado, precisa transferir um (ou mais) processo para outro processador. Escolher um processador para receber tal processo não é uma tarefa simples. A maneira mais direta e intuitiva para descobrir qual processador deve receber um processo seria perguntar a **todos** os processadores do sistema qual é a sua carga e esperar todas as respostas. O processador que se disser menos carregado deveria receber o processo. Estudos [EAG86] mostram que tal mecanismo por si só introduz uma elevada sobrecarga ao sistema devido a quantidade de mensagens trocadas, vindo de encontro ao resultado esperado.

Ao fazer a distribuição da carga de processamento pode ser levado em consideração o custo da comunicação entre os processos. Para equilibrar a carga de processamento em multiprocessadores tipo *mesh* ou hipercubo, pode-se, basear em uma solução de um problema de mínimo custo [BOK93]. Para escolher um caminho com menor custo para migração de um processo, atribui-se um custo a cada conexão interprocessador, um para para cada passagem por um processador e ainda um custo adicional para trocar a direção nesta passagem. Para tomar a decisão com rotas sem coincidências, evitando problemas de contenção que alteram o custo das conexões, há necessidade de se saber o estado global do sistema.

O processo de migração de processos pode ser iniciado por processadores sobrecarregados ou sub-utilizados. Os processo que inicia em um processador sobrecarregado normalmente é chamado de *sender initiated* e aquele que inicia em processadores sub-utilizados, *receiver initiated* [TAN92]. Em um mecanismo *sender initiated*, um processador que se der por sobrecarregado tenta localizar algum processador sub-utilizado que possa receber parte de seu trabalho. Por outro lado, quando um processador se descobrir sub-utilizado, pode usar métodos *receiver initated* para encontrar processadores sobrecarregados capazes precisando repartir sua carga.

Dependendo de onde for iniciar a transferência, diferentes aspectos devem ser levados em consideração, como por exemplo, como fazer o levantamento de processadores em questão e por quanto tempo continuar tal levantamento. Uma proposta híbrida para o início do processo é encontrada em [SHI92], onde mensagens são trocadas entre os processadores em várias ocasiões e as informações sobre processadores remotos estarem sobrecarregados, normais ou sub-utilizados são guardadas no início de três filas. Cada processador que entrar no estado sobrecarregado ou sub-utilizado troca mensagens com os processadores de suas filas, varrendo-as na ordem conveniente.

Normalmente os sistemas distribuem sua carga de processamento fazendo uso de migração de processos. A maneira mais simples de migrar um processo é congelar o processo no seu computador original, transportar todo o seu estado (incluindo registradores, memória, arquivos abertos, etc.) para outro computador e descongelá-lo em seu novo local, para que possa executar [OUS88]. Porém, nem sempre é possível reproduzir todo o ambiente original de um processo após a migração, porque alguns recursos dos sistemas não são distribuídos e não podem ser simulados em todos os processadores.

Leiss descreve um algoritmo distribuído para equilibrar a carga de processadores fracamente acoplados [LEI91]. O algoritmo possui um critério local, onde são somente levadas em consideração a carga do processador local e a

de seus vizinhos. Para isso, periodicamente os processadores trocam informações sobre a sua carga de processamento com os vizinhos. Se a carga de um certo processador for muito maior que a média da sua vizinhança, um de seus processos deve ser migrado. O processador que estiver com a menor carga desta mesma vizinhança é que recebe tal processo.

No modelo do gradiente, os processadores também somente interagem com seus vizinhos [LIN87]. Um processador ocioso avisa seus vizinhos que está à distância **zero** de um processador ocioso, desencadeando todo o mecanismo de equilíbrio. Cada processador, ao receber a informação de ociosidade, repassa tal informação incrementado a informação de distância. Todos os processadores sobrecarregados migram um de seus processos na direção do processador ocioso. O mecanismo deve se repetir até que não existam mais processadores ociosos ou processos para migrar, o que ocorre quando o equilíbrio for alcançado.

A maioria dos algoritmos citados assumem algumas restrições para diminuir o número de mensagens trocadas. Eles também procuram descentralizar e distribuir ao máximo a informação de carga, de forma que cada processador possa tomar a decisão de transferir um processo a outro de maneira autônoma. Através destas restrições, nem sempre é possível se fazer uma distribuição perfeitamente equilibrada da carga de processamento. Atualmente, o desafio à pesquisa nesse assunto é conseguir equilibrar a carga inserindo um mínimo de sobrecarga ao sistema.

2.4 Outros Trabalhos Relacionados

Esta seção apresenta alguns trabalhos que se relacionam mais intimamente com o TRIX. Três deles foram feitos no Instituto de Informática da UFRGS e desencadearam em dissertações de mestrado [STE92], [BEL93] e [SCH92]. Há também um trabalho, chamado DISTRIX, desenvolvido na Universidade da Cidade

do Cabo, que tem um objetivo semelhante ao do TRIX, porém com abordagens diferentes para os problemas encontrados.

2.4.1 Sistema M3P

M3P significa Minix Multi-MicroProcessador, e é um sistema que possui um processador mestre e diversos processadores escravos, com um mecanismo de DMA entre os mesmos [CAR89]. É montado com processadores i8086 em placas de expansão de um IBM PC. Este projeto trouxe ao grupo de Processamento Paralelo do Intituto de Informática da UFRGS bastante experiência com o MINIX em multiprocessadores, desencadeando em várias dissertações de mestrado, em assuntos relacionados.

A parte principal do *software* do M3P é uma versão alterada do MINIX, com uma única cópia do servidor de arquivos sendo executada no mestre [SCH92]. Há servidores de processos no mestre e nos escravos, porém o servidor que é executado no mestre é diferente dos demais e faz o controle centralizado dos processos em execução nos escravos. Este servidor faz migração de processos entre os escravos, no intuito de equilibrar a carga de processamento.

A identificação de processos no sistema é implementada dividindo o identificador em duas partes: processador e processo. O núcleo ao perceber que uma mensagem não é local, entrega-a para uma *task* de comunicação, que se encarrega de fazê-la chegar ao processador destinatário. Para fazer a transferência de mensagens entre os processadores existe um microcontrolador dedicado, chamado **ouvidor** [STE89].

2.4.2 Sistema Operacional Distribuído DIX

Extendendo o trabalho do projeto M3P, foi criado um sistema operacional distribuído de nome DIX [BAR90]. Este é baseado no MINIX e executa em estações de trabalho especiais, que possuem dois processadores — hospedeiro e principal — conectadas em uma rede. O processador hospedeiro executa as camadas mais baixas do MINIX, pois serve somente para controlar os dispositivos. O processador principal executa um sistema operacional distribuído, com duas camadas a mais que o Minix original. Tais camadas provêm a distribuição do sistema tornando-a transparente aos processos dos usuários e fazendo uso dos servidores originais do Minix.

A implementação de um núcleo que suportasse a sua distribuição foi feita em uma dissertação de mestrado [STE92]. Tal núcleo fornece um mecanismo de comunicação com independência de localização e suporte à migração de processos. Foi construída também um rede em barramento para conexão dos equipamentos, usando a interface paralela do hospedeiro, com altas taxas de transferência.

2.4.3 O Sistema DISTRIX

Foi desenvolvido um sistema operacional com características semelhantes às do presente trabalho, no departamento de Ciência da Computação da Universidade da Cidade do Cabo, África do Sul. Este sistema, chamado DISTRIX [SMI89], possui processadores centrais para gerência de processos, para o sistema de arquivos e para controle de terminais, e processadores satélites para a execução de processos de usuários. Por causa da falta de proteção de memória, cada processador satélite executa processos de um único usuário. O sistema é baseado no MINIX, mas o servidor de processos e o servidor de arquivos não foram distribuídos.

A memória virtual é descartada até que a INMOS crie um modelo de transputer que a implemente [MCC90]. O escalonador do transputer efetua a despacho dos processos por *hardware*, o que dificulta a implementação de sinais segundo a semântica UNIX, pois não se sabe qual o processo em execução a cada instante. A solução para este problema no DISTRIX é um processo extra, que não faz nenhuma computação útil mas garante que ele é o processo do qual o processador foi tomado no instante do sinal, permitindo-se sinalizar qualquer outro processo do sistema[1].

2.4.4 Gerência de Processos em Sistemas Distribuídos Tolerantes a Falhas

Foi projetado um sistema que possui um gerente de processos com duas tarefas principais, além das tradicionais em um sistema cetralizado comum [BEL93]. São elas: manutenção do equilíbrio da carga dos processadores do sistema e a recuperação dos processos após uma falha em algum processador. Os algoritmos empregados pelo gerente de processos foram projetados para que nao se tornassem escoadouros dos recursos computacionais do sistema. Isso permite que a distribuição se traduza em ganhos mais significativos no desempenho do sistema.

O trabalho aborda aspectos relacionados a gerência de processos em sistemas operacionais distribuídos tolerantes a falhas, com ênfase em políticas transparentes de recuperacão de processos e equilíbrio de carga. Tambem são considerados mecanismos que implementem eficientemente essas políticas, tais como pontos de recuperação e migração de processos, respectivamente. A inter-relacao entre tais politicas é avaliada, dando origem ao projeto de um gerente de processos, que é simulado através da implementação de um protótipo, com a finalidade de validar as politícas empregadas.

[1] Este problema será visto com mais detalhe em seções posteriores.

3 O MICROPROCESSADOR TRANSPUTER

Transputer é o nome genérico de uma família de processadores projetados pela empresa inglesa Inmos [INM87]. O transputer possui uma série de características especiais, que facilitam a iplementação de aplicações com processamento paralelo, como será detalhado neste texto. Os processadores deste tipo variam no seu conjunto de instruções, no tamanho de sua palavra e na sua velocidade. Mesmo assim, os processadores mais novos mantêm compatibilidade com os anteriores.

Os modelos mais comuns atualmente são o IMS T414 e IMS T800, ambos de 32 bits. O segundo é mais novo e possui um conjunto de instruções mais completo, incluindo aritmética de ponto flutuante. Um T800 com freqüência de relógio de 33 MHz lhe confere um desempenho próximo de 2 MFLOPS. Estes processadores apresentam uma deficiência com relação a falta de proteção de memória e relocação no código executável. O modelo T9000 [POU91b], possui um controle de memória mais avançado, suprindo esta deficiência. Além disto, O T9000 implementa um sistema de comunicação com roteamento automático de mensagens. Não obstante, o seu ponto forte é o desempenho, pois com um relógio de 50 MHz, executa 200 MIPS e 50 MFLOPS.

3.1 O Conjunto de Instruções

Embora existam transputers de 16 e de 32 bits, o seu conjunto de instruções independe do tamanho da sua palavra [SHE87]. Isto permite executar um programa feito para um transputer de 16 bits em outro de 32 bits sem alterar o seu código. Um endereço de memória tem o tamanho de uma palavra do processador e é composto de dois campos: *word address* e *byte selector*. O *byte selector* possui o número de bits necessários para selecionar um byte dentro de

uma palavra. Assim, processadores de 16 bits tem 1 bit para *byte selector* e os de 32 bits usam dois. O restante dos bits do endereço são usados como *word address*, o que define uma palavra do espaço de endereçamento.

No início do espaço de endereçamento de memória do transputer existem algumas palavras reservadas, como será visto nas seções seguintes. A tabela 3.1 mostra um mapa das palavras reservadas. O transputer possui uma memória estática interna ao *chip* que é mapeada nos endereços iniciais do espaço de endereços. O tamanho desta memória é de 2 a 8 kilobytes, dependendo do modelo do transputer [INM88a].

Tabela 3.1: Mapa do início da memória do T414

palavra	descrição
0 a 3	*link inputs*
4 a 7	*link outputs*
8	reservada
9 e 10	ponteiros das filas dos *timers*
11 a 17	`RegSaveArea` (ver seção 3.2)
18	`MemStart` (primeiro endereço livre)

3.1.1 Os Registradores do Transputer

O transputer possui seis registradores todos do tamanho de uma palavra. Dentre eles, três registradores são organizados na forma de uma pilha, chamados `Areg`, `Breg` e `Creg`. Ao se mover um valor para `Areg`, seu valor é anteriormente copiado para `Breg`, que tem seu valor copiado em `Creg`. Ao se retirar um valor de `Areg`, este assume o valor de `Breg` e o `Breg` assume o valor de `Creg`. Além dos registradores da pilha, que são de uso geral, existem ainda mais três, de nomes `Iptr`, `Wptr` e `Oreg`, que também são usados em programas de usuário.

`Iptr` (*Instruction Pointer*) é o registrador que contém o endereço da próxima instrução a ser executada. `Wptr` (*Workspace Pointer*) contém o endereço da

área de trabalho. É usado como ponteiro da pilha, onde são salvos os endereços de retorno de subprogramas. Variáveis locais a um subprograma são implementadas com endereços relativos à ele. `Oreg` (*Operand Register*) serve para armazenar o operando de uma instrução durante sua execução. A tabela 3.2 mostra todos os registradores do transputer. Alguns deles serão descritos nas próximas seções.

Tabela 3.2: Registradores do transputer

nome	descrição
`Areg`	topo da pilha
`Breg`	segundo da pilha
`Creg`	último da pilha
`Oreg`	registrador operando
`Wptr`	endereço do *workspace*
`Iptr`	endereço da próxima instrução
$\texttt{ClockReg}_0$	contador do relógio urgente
$\texttt{ClockReg}_1$	contador do relógio não-urgente
$\texttt{TNextReg}_0$	*timer input* urgente
$\texttt{TNextReg}_1$	*timer input* não-urgente
$\texttt{FptrReg}_0$	endereço início da fila urgente
$\texttt{BptrReg}_0$	endereço fim da fila urgente
$\texttt{FptrReg}_1$	endereço início da fila não-urgente
$\texttt{BptrReg}_1$	endereço fim da fila não-urgente
`STATUSreg`	*error* e *halt-on-error flags*

3.1.2 Formato das Instruções

As instruções do transputer têm todas um byte de comprimento, dividido em dois campos de 4 bits. Os 4 bits mais significantivos contém um *function code* e os 4 menos significativos, um *data value*, que serve como operando da função. Esta representação permite dezesseis funções diferentes, numeradas de 0 a 15. Treze delas (tabela 3.3) são usadas para codificar as instruções mais importantes na tradução de programas em linguagem de alto nível, como desvios e acesso a variáveis. Duas, chamadas de prefixos, são usadas para estender o tamanho do operando da instrução. Por fim, existe uma instrução chamada *operate*, que permite executar todas as outras instruções, que não as treze mais comuns.

Antes de executar qualquer instrução, o *data value* é somado ao registrador de operando. Para que as instruções possam operar com valores de mais de quatro bits, existem as intruções de prefixo. Tais instruções rodam o registrador de operando quatro bits à esquerda, logo após terem somado o *data value*. Após executar qualquer instrução que não seja de prefixo, o registrador de operando recebe o valor zero.

Tabela 3.3: Funções diretas do transputer

código	instrução	descrição	
0x0n	`j`	*jump*	Iptr += Oreg
0x1n	`ldlp`	*load local pointer*	Areg = Wptr + Oreg * wl[1]
0x2n	`pfix`	*prefix*	Oreg = (Oreg + n) << 4
0x3n	`ldnl`	*load non-local*	Areg = *(Areg + Oreg * wl)
0x4n	`ldc`	*load constant*	Areg = Oreg
0x5n	`ldnlp`	*load non-local pointer*	Areg = Areg + Oreg * wl[1]
0x6n	`nfix`	*negative prefix*	Oreg = ~(Oreg + n) << 4
0x7n	`ldl`	*load local*	Areg = *(Wptr + Oreg * wl)
0x8n	`adc`	*add constant*	Areg += Oreg
0x9n	`call`	*call*	*Wptr– = Creg,Breg,Areg,Iptr Iptr += Oreg
0xAn	`cj`	*conditional jump*	Iptr += Areg ? 0 : Oreg
0xBn	`ajw`	*adjust workspace*	Wptr += Oreg * wl
0xCn	`eqc`	*equals constant*	Areg = (Areg == Oreg) ? 1 : 0
0xDn	`stl`	*store local*	*(Wptr + Oreg * wl) = Areg
0xEn	`stnl`	*store non-local*	*(Areg + Oreg * wl) = Breg

3.2 Processos em um Transputer

Os processadores do tipo transputer têm embutido em seu *hardware* um escalonador de processos. Ele permite disparar e terminar processos, implementando o bloqueio destes processos em operações de entrada e saída. Um processo pode ser executado com prioridade 0 ou 1, sendo chamado de **urgente** e **não-urgente** respectivamente.

O registrador `Wptr` é chamado também de descritor do processo. Ele, por ser um endereço, é composto por um *word address* e um *byte selector*. O `Wptr`

não usa o campo *byte selector*, assumindo sempre para ele o valor zero, apontando para o início de uma palavra. O bit menos significativo do *byte selector* é usado então para guardar a prioridade do processo. Por exemplo, um processo de 16 bits com `Wptr` valendo 0x1201, possui seu *Workspace* no endereço 0x1200 e é um processo não-urgente (prioridade 1).

A literatura sobre transputers se refere a processos como sendo os fluxos de execução controlados pelo seu escalonador por *hardware*. Como neste trabalho o termo *processos* tem um significado diferente, a partir deste parágrafo será atribuído o nome de *thread* para tais fluxos de execução. Este nome é bem mais apropriado sob o ponto de vista do sistema operacional. É normalmente adotado como convenção que quaisquer fluxos de execução de processos sejam chamados de *thread* [RAS86, MUL90].

Existem duas listas de *threads* prontas para executar, uma para cada prioridade. Quando um processo está nesta lista, seu `Iptr` está armazenado na posição `Wptr`-1 e em `Wptr`-2 está o valor do `Wptr` do próximo processo na lista. Disparar uma *thread* em um transputer é o mesmo que torná-la pronta para ser executada. Para isto, é preciso simplesmente adicionar um valor para ser assumido por seu registrador `Wptr` ao fim da lista de *threads* prontas de sua respectiva prioridade. Para implementar isto, existem quatro registradores especiais que apontam para o início ($\texttt{FptrReg}_0$ e $\texttt{FptrReg}_1$) e o fim ($\texttt{BptrReg}_0$ e $\texttt{BptrReg}_1$) de cada uma das filas. Eles são usados pelo escalonador e pelas instruções de controle das *threads*, mostradas na tabela 3.4.

Tabela 3.4: Funções do escalonador do transputer

instrução	descrição
`runp`	inicia *thread* com `Wptr` = `Areg` e `Iptr` = `Wptr` -1
`stopp`	termina *thread* e guarda `Iptr` em `Wptr` -1
`startp`	inicia *thread* com `Iptr` = `Iptr` + `Breg` e `Wptr` = `Areg`
`endp`	carrega `Wptr` com `Areg`, decrementa conteúdo de `Wptr` -1 e termina *thread* se resultado > 0

Uma *thread* urgente ganha o controle do processador no instante que estiver no início da lista de *threads* prontas para executar. Se no momento que a *thread* urgente for executar, o processador estiver executando uma *thread* não-urgente, os registradores são salvos em uma área especial de memória, chamada `RegSaveArea`. Se uma *thread* urgente se tornar pronta para executar, aparecendo na fila enquanto uma outra também urgente estiver sendo executado, nada acontece, a que estiver executando continua em execução.

Uma *thread* urgente somente perde o controle do processador ao terminar ou ficar bloqueada por uma operação de entrada ou saída. Quando isto ocorre, se existir alguma *thread* na lista das urgentes, o controle do processador é passado à ela. Se não existir, é verificado se existem registradores de alguma *thread* não-urgente salvos na `RegSaveArea`. Caso positivo, tal *thread* não-urgente retoma a execução. Se ambas alternativas não forem satisfeitas e existir alguma *thread* na fila não-urgente, é executado a primeira *thread* da fila. Se, por fim, nada disso for satisfeito, o processador pára, esperando alguma *thread* aparecer em alguma das filas. Isso pode acontecer, por exemplo, quando chegar uma mensagem por um dos *links*.

Uma *thread* não-urgente pode perder o controle do processador de duas formas: por bloqueio em uma instrução de entrada e saída (seção 3.3) ou por *timeslice*. Após cada 1024 microssegundos termina um *timeslice*. Quando uma *thread* não-urgente executar continuamente por dois *timeslices*, o processador espera a *thread* executar uma instrução *jump* ou *loopend*. Neste instante, o registrador `Wptr` é salvo no fim da fila de *threads* não urgentes. Um novo valor para `Wptr` é carregado a partir do início da mesma fila. Assim, é implementado um *round-robin* entre as *threads* de baixa prioridade.

3.3 Entrada e Saída de Dados

O transputer implementa instruções para entrada e saída de dados através de canais de comunicação, de forma integrada ao seu escalonador de processos (*threads*). Um canal é descrito como uma palavra comum de memória. Para ligação do transputer com o exterior, existem quatro canais especiais, chamados *links*. Seu funcionamento é idêntico ao de canais comuns, porém seus endereços são fixos, no início da memória do processador (ver tabela 3.1).

Basicamente, existem duas instruções de E/S: `in` (*input message*) para receber dados e `out` (*output message*) para enviar dados. Ambas recebem como parâmetros, três valores: endereço do canal, endereço dos dados e tamanho dos dados. Ao executar uma instrução `in` ou `out`, o transputer verifica se não existe nada armazenado na palavra apontada pelo canal. Se existir, este conteúdo representa o registrador `Wptr` da *thread* interlocutora da comunicação. Com isto, a transferência dos dados é feita de uma *thread* à outra. Se a palavra apontada pelo canal estiver vazia, os valores de endereço e tamanho dos dados são armazenados no *workspace* da *thread* e seu `Wptr` é armazenado no endereço do canal. O `Wptr` da *thread* não é inserido no final da fila das *threads* prontas, ficando parada (bloqueada). Assim, a troca de mensagem será feita quando o interlocutor executar uma instrução de E/S neste canal.

Nas mensagens internas, a quantidade de dados transferida é dada pelo tamanho solicitado pela segunda *thread* a se comunicar. Por isto, se dois tamanhos diferentes forem usados, os dados do destinatário podem ser corrompidos. Não é feito o controle do sentido da comunicação. Se uma *thread* executa *out* em um canal, ficando bloqueada, e outra *thread* interlocutora executar novamente *out* neste canal, o sentido da transferência será dado pela última. É necessário portanto um controle adicional tanto no sentido como no tamanho das transferências de mensagens.

Tal problema não ocorre com as mensagens inter-processador, porque para cada *link* existe um contador de bytes transferidos. Assim o transputer desbloqueia o processo no instante em que tiverem sido transferidos exatamente todos os bytes solicitados. Além disso, existem canais específicos para entrada e para saída em cada *link*. Não é possível enviar dados em um *link* de entrada ou vice-versa.

3.4 Temporização

Dentro do transputer existem dois contadores de tempo, um para cada prioridade de *thread*. O contador de alta prioridade $\texttt{ClockReg}_0$ é incrementado a cada 1 μs (10^{-6} segundos) e o contador de baixa prioridade $\texttt{ClockReg}_1$, a cada 64 μs. Estes incrementos são chamados de pulsos de relógio. Existem instruções para ler e gravar valores nos contadores de tempo. O contador de alta prioridade é também usado pelo *escalonador*, que troca a *thread* não-urgente em execução a cada 2048 pulsos de relógio(ver seção 3.2).

A instrução `tin` (*timer input*) bloqueia a *thread* corrente até que o relógio da prioridade desta *thread* atinja o valor especificado no topo da pilha (registrador `Areg`). Na seqüência de instruções da figura 3.1, em uma *thread* de baixa prioridade, é empilhado o valor corrente do relógio (`ldtimer`) e somado com 15625. Logo após aparece a instrução `tin`, que bloqueia a *thread* por um segundo ($15625 \times 64\ \mu s$).

```
0x22F2      ldtimer
0x232D2049  ldc       15625
0x25F2      sum
0x22FB      tin
```

Figura 3.1: Bloqueio temporizado do tranputer

O bloqueio de *threads* por tempo é implementado por meio de duas filas de *threads*, uma para cada prioridade e dois registradores (`TNextReg`$_0$ e `TNextReg`$_1$) indicando o próximo valor esperado pelos dois *timers*. Desta forma, o *escalonador* verifica estes registradores a cada troca de contexto. Enquanto o relógio não atingir o valor especificado no *timer input*, *threads* das filas de prontas são executadas. Isto implica que uma *thread* bloqueada por tempo espere, no mínimo, o tempo pedido, mas não garante que, **exatamente** após este tempo passar volte a receber a CPU.

3.5 Seleção de Alternativas

O fato de uma instrução `in` ou `out` sempre bloquear o processador não permite que se construa *threads* que recebam mensagens de diversas outras *threads*, de forma aleatória. Por exemplo, seja um servidor que receba mensagens contendo pedidos de vários processos. Este servidor, em um dado instante, para receber uma mensagem de um de seus processos clientes executa uma instrução `in`. Se o processo cliente não estiver bloqueado (no caso, não tiver executado um `out`), o servidor ficará bloqueado. Neste instante, todos os outros processos clientes que tentarem se comunicar com o servidor ficarão bloqueados, sem que o servidor tome conhecimento disto, por também estar bloqueado. Somente após receber a mensagem do cliente que o bloqueou é que o servidor poderá verificar se os outros processos desejam algum serviço.

Para solucionar este problema, o transputer implementa um conjunto de instruções, chamado de alternativas. Este conjunto permite uma *thread* indicar vários canais com os quais deseja se comunicar, ficando bloqueada após isto. Com a detecção de comunicação em algum dos canais especificados, a *thread* é desbloqueada e desviada para um trecho de código previamente definido, o qual concretiza a troca de mensagem.

Além de bloquear em diversos canais, uma seqüência de alternativas pode ser temporizada, permitindo que se especifique um período de *timeout*, após o qual a comunicação é cancelada se nenhuma mensagem foi enviada. Se nenhum canal especificado nas alternativas demonstrar interesse em se comunicar no período de tempo especificado, a *thread* é desbloqueada e desviada para o trecho de código que trata o *timeout*.

A seleção de alternativas é implementada da seguinte forma [SHE87]:

- Instrução `alt` ou `talt`, indicando o início de uma seleção de alternativas, sem ou com temporização, respectivamente.
- Uma seqüência de instruções para habilitar canais (`enbc`) ou temporizadores (`enbt`).
- Instrução `altwt` (ou `taltwt`, com temporização), usada para esperar a ocorrência de alguma alternativa. Esta instrução bloqueia o processo até que apareç um interlocutor em algum canal ou termine um *timer*.
- Uma seqüência de instruções de desabilitação de canais (`disc`) e temporizadores (`dist`). Estas instruções fornecem um endereço para desvio, para o caso de a alternativa ser satisfeita.
- Instrução `altend` que desvia a execução para a primeira alternativa selecionada a ser desabilitada.

4 O SISTEMA OPERACIONAL MINIX

O sistema operacional MINIX é compatível com o UNIX versão 7 [BAC87], feito por Andrew Tanenbaum, professor da Universidade Livre de Amsterdam. O sistema foi escrito em linguagem C, sendo publicado em um livro [TAN87], para ser usado em aulas de sistemas operacionais. Existem versões do sistema para microcomputadores do tipo IBM-PC, MacIntosh, Amiga e Atari.

4.1 Estrutura do Sistema

O MINIX é dividido em quatro camadas, como mostra a figura 4.1, cada qual desempenhando uma atividade específica. A camada mais baixa, chamada *Process Management* é responsável por controlar as interrupções e o escalonamento de processos, salvando e a restaurando registradores. É esta camada também que implementa um mecanismo de troca de mensagens entre processos. Sua função é prover às camadas superiores um modelo de processos seqüenciais independentes, que se comunicam usando mensagens.

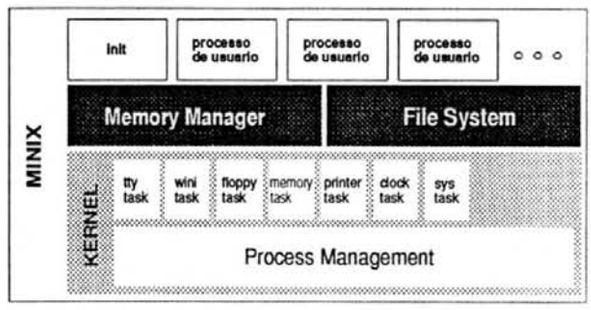

Figura 4.1: Estrutura do MINIX

A segunda camada é formada pelos processos que controlam os dispositivos de entrada e saída (E/S). Cada dispositivo de E/S possui um processo de controle, denominado *task*, ou *driver*. Existem *tasks* para controlar terminais, discos, memória, impressoras e relógio. Existe ainda a *system task*, que serve para

auxiliar as camadas superiores no controle do sistema, não efetuando operações de E/S.

Todas as funções e processos das duas camadas inferiores são ligadas em um único módulo executável, denominado núcleo. Processadores que permitem execução em modo supervisor e modo usuário, executam o núcleo em modo supervisor. Embora as *tasks* estejam todas em um único módulo, elas são escalonadas independentemente e se comunicam usando mensagens. Existem porém algumas situações onde uma *task* chama diretamente uma função de outra.

A camada 3 é formada por dois processos: *Memory Manager* e *File System* (MM e FS). O primeiro interpreta todas as chamadas MINIX que dizem respeito o controle de memória e gerência de processos, como `fork`, `exec` ou `brk`. O segundo, interpreta todas as chamadas referentes ao sistema de arquivos, como `open`, `read` ou `chdir`. O *File System* foi escrito para ser um servidor de arquivos, podendo ser deslocado para um processador remoto, com poucas alterações.

As três primeiras camadas implementam o sistema operacional propriamente dito. A quarta e última camada é formada pelos processos de usuário, como `shell`, editores, compiladores e os programas executáveis criados pelo usuário. O processo `init` é um processo de inicialização do sistema, executado como processo de usuário.

4.2 Processos no MINIX

Todos os processos MINIX seguem o modelo descrito na seção anterior. Um processo pode criar subprocessos, que por sua vez, podem criar outros subprocessos, criando uma árvore de processos. Na verdade, todo o sistema faz parte de uma única árvore de processos, como está descrito a seguir.

Ao ligar o equipamento, o seu *firmware* lê o primeiro setor do disco para a memória e o executa. O programa contido nesta parte do disco é normalmente chamado de *bootstrap*. Este programa, por sua vez, lê do disco todo o sistema operacional, transferindo o controle do processador à ele. O sistema operacional, ao tomar o controle pela primeira vez, inicia a execução de todas as *tasks*, do *Memory Manager* e do *File System*. Feito isto, o controle é passado ao processo `init`, que é carregado junto com o sistema operacional.

O processo `init` dispara um processo *login* para cada terminal existente, cada qual esperando um usuário se conectar ao sistema. Ao aceitar a conexão de um usuário, o processo *login* executa um programa chamado `shell`, que passa a interpretar os comandos digitados pelo usuário, como por exemplo, disparar outros processos. Diz-se que o `shell` é filho do `init`, assim como os processos do usuário são filhos do `shell` e netos do `init`, formando uma árvore. Um exemplo desta árvore em um sistema convencional está na figura 4.2, onde os processos `-csh` e `-sh` são dois tipos diferentes de `shell`.

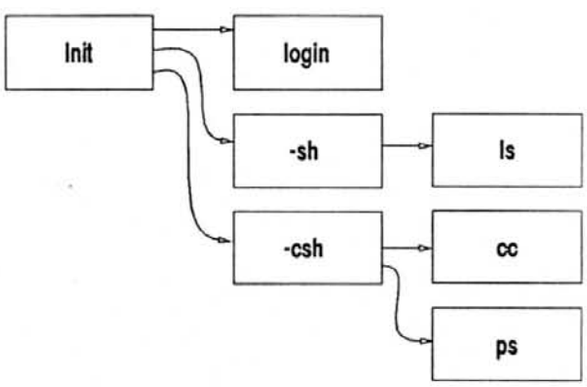

Figura 4.2: Árvore de processos do MINIX

Para se disparar um novo processo no MINIX, é necessário efetuar a chamada `fork`. Esta chamada duplica todo o processo chamador, criando duas cópias idênticas do mesmo. Os dois processos iniciam sua execução logo após o retorno da chamada `fork`. Eles somente se identificam como sendo pai e filho porque cada um recebe um valor diferente no retorno do `fork`. Para fazer um processo executar um novo programa, usa-se a chamada `exec`. Esta chamada

efetua a carga do programa do disco para a memória. No momento da carga, a memória disponível ao processo é ajustada para a quantidade necessária indicada no cabeçalho do programa. Resumindo, um processo somente pode criar um filho diferente de si emitindo uma chamada `fork`, seguida de um `exec`, já no processo filho.

Toda a informação referente aos processos do sistema está em uma tabela, que é dividida entre o núcleo, o *Memory Manager* e o *File System*, cada qual com os campos que necessita. Quando um processo é criado ou terminado, o *Memory Manager* atualiza a sua parte da tabela e envia mensagens ao núcleo e ao *File System*, dizendo para fazerem o mesmo.

4.3 Comunicação entre Processos

A comunicação entre processos MINIX é feita através da troca de mensagens de tamanho fixo. O tamanho da mensagem depende do tamanho da estrutura de dados chamada *message*, que num microprocessador i8088 tem 24 bytes. Em outras CPUs, como transputer por exemplo, onde inteiros são representados em 4 bytes em vez de 2, tal estrutura é maior.

Existem três primitivas para troca de mensagens entre processos, implementadas pelas seguintes funções da biblioteca C:

```
send(dest, &message);
receive(source, &message);
send_rec(src_dst, &message);
```

para enviar uma mensagem a um processo `dest`, para receber uma mensagem de um processo `source` e para enviar uma mensagem e esperar a resposta do mesmo processo, respectivamente. Os valores `source`, `dest` e `src_dst` identificam o

processo interlocutor e representam sua posição na tabela de processos do sistema. Esta posição é também chamada de *slot*. Na função `receive` pode-se usar um identificador especial `ANY`, que permite a recepção de mensagens de qualquer processo.

Cada processo ou *task* pode trocar mensagens com processos na sua mesma camada ou em camadas adjacentes. Um processo de usuário não pode se comunicar diretamente com uma *task*. Servidores podem se comunicar com *tasks* e com processos de usuário.

Quando um processo envia uma mensagem, o núcleo verifica se o destinatário já está bloqueado esperando por uma mensagem, ou seja, já chamou a função `receive`. Caso positivo, a mensagem é copiada pelo núcleo do *buffer* do processo remetente para o *buffer* do processo destinatário. Senão, o remetente é bloqueado, e colocado em uma fila de processos esperando para enviar uma mensagem ao destinatário dado.

Quando um processo destinatário chama a função `receive` e existir algum processo apto a remeter uma mensagem já bloqueado em sua fila, o *buffer* é copiado do remetente ao destinatário, desbloqueando o remetente. Se não houver nenhum processo na sua fila, o destinatário é bloqueado até alguma mensagem ser enviada para ele. Este método, em que tanto o rementente quanto o destinatário ficam bloqueados até a comunicação ser efetuada, é chamado de *rendezvous*. Embora seja menos flexível que um sistema assíncrono, com *buffers*, é bem mais simples de implementar, usar e depurar.

4.4 Escalonamento de Processos

O escalonador de processos do MINIX possui três níveis de prioridade, divididas de acordo com as camadas do sistema. As *tasks* são os processos de mais

alta prioridade, seguidos dos servidores, e por fim, pelos processos de usuário. O sistema mantém uma fila de processos prontos para cada prioridade, como pode ser visto na figura 4.3. Sempre que um processo for desbloqueado ele é colocado no fim da sua fila correspondente. Um vetor `rdy_head` mantém apontadores para o início das filas dos processos prontos de cada prioridade. O vetor `rdy_tail` é implementado para facilitar esta operação. Quando um processo é bloqueado ou terminado, ele é retirado da fila.

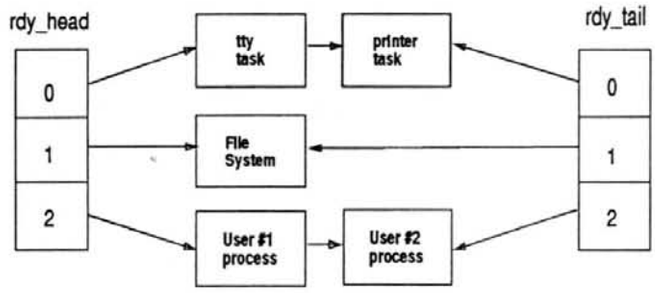

Figura 4.3: As filas de processos MINIX

Para escolher qual processo executar, o escalonador verifica se não há nenhuma *task* apontada por `rdy_tail`. Se existir, esta é executada. Se não houverem *tasks* capazes de executar, o mesmo procedimento é feito para os servidores (MM e FS). Se ambos estiverem bloqueados, repete-se o algoritmo para processos de usuário. Nos casos em que não houver nenhum processo pronto para executar, o sistema executa um trecho de código chamado *idle* que faz a CPU ficar parada enquanto espera algum processo ficar pronto.

Normalmente, um novo processo só perde a CPU quando ficar bloqueado ou terminar. No entanto, existe uma situação em que é trocado o processo de usuário em execução. Ao final de cada *timeslice* (100 milisegundos) o escalonador toma o controle do processador. Se o processo que estiver executando for o mesmo em execução no início do *timeslice* e for um processo de usuário, este perde o controle da CPU. O processo que estava executando é colocado no fim da fila, passando a CPU ao próximo processo de usuário que esteja pronto. Se não houver nenhum outro processo pronto para executar, o processo que acabou

de perder o controle voltará a ganhar a CPU. Isto implementa um *round-robin*, distribuindo a CPU entre os diversos processos de usuário.

Processos *tasks* e servidores não são escalonados como processos de usuário. Garante-se que estes processos do sistema operacional desempenham rápidas tarefas, bloqueando-se sempre ao final. As *tasks* somente perdem o controle da CPU ao se bloquearem, garantindo que funcionem como monitores [HOA74]. Os servidores tomam a CPU dos processos de usuário mas perdem para as *tasks*, formando assim uma hierarquia de prioridades.

5 PROPOSTA DO TRIX

O projeto do sistema TRIX tomou como ponto de partida a necessidade de um sistema operacional para transputers. Decidiu-se que este sistema operacional seria compatível com um sistema popular, o UNIX, para ter maior amplitude em sua aplicação. Para atingir este objetivo sem escrever um sistema operacional a partir do nada, optou-se por dotar um sistema operacional conhecido — o MINIX — de características de multiprocessamento.

Este sistema deve controlar um grupo de processadores de maneira homogênea. Qualquer processador que estiver executando o sistema está em um mesmo nível de hierarquia, não devendo haver processadores centrais ou principais. Isto dá mais flexibilidade, escalabilidade e disponibilidade (e mais complexidade) ao sistema, permitindo que qualquer processador seja alocado para qualquer tarefa, à medida que necessário.

O processamento de tarefas de usuário deve ser feito em mais de um processador, para permitir um crescimento incremental de desempenho para aplicações bastante complexas. Isso implica que todos os serviços do sistema estejam distribuídos de maneira transparente pelos diversos processadores. Para tanto, os servidores do sistema também devem ser distribuídos em todos os processadores ou, pelo menos, devem poder atender processos clientes executando em processadores diferentes ao seu.

Ainda, transformar o MINIX em TRIX não significa simplesmente recompilar ou portar o sistema. Os programas, por executarem tarefas de baixo nível, devem ser alterados de forma a se adaptar ao novo *hardware* a ser usado. Esta tarefa se torna ainda mais trabalhosa e delicada ao inserir o paradigma de multiprocessamento, onde a probabilidade de se inserirem novos erros nos programas, independentes do porte em si, aumenta. Neste capítulo será apresentado o pro-

jeto do sistema como um todo, detalhando o porte do MINIX e os motivos de cada nova característica implementada.

5.1 Divisão dos Processos no Sistema

Para controlar uma máquina multiprocessada, o código do MINIX teve que ser repartido em várias partes e algumas destas, replicadas em mais de um lugar. Nesta seção serão descritas quais partes são necessárias em cada processador e porquê. Mais precisamente, será discutido aqui como se distribuem no sistema os processos de usuário, servidores e *tasks*. Embora o TRIX deva suportar a execução de todo e qualquer processo em todo e qualquer processador, será visto que isso nem sempre será feito. Mais que isso, em alguns casos é mais conveniente que algum processo execute em apenas um processador específico.

5.1.1 Arquitetura Suportada

Para se discutir o local dos processos e os motivos de sua distribuição, se faz necessário conhecer melhor a arquitetura a ser suportada. A figura 5.1 exemplifica um *hardware* típico no qual o TRIX executa. Nela aparece um processador hospedeiro (um Intel 80x86) e alguns transputers interligados. Na verdade, a maneira de interligação entre os transputers, bem como a sua quantidade, pode variar bastante, mas normalmente um só (ou poucos deles) tem contato direto com o processador hospedeiro do sistema.

Nos sistemas comerciais existentes, os transputers não executam nenhum sistema operacional e o hospedeiro executa seu sistema nativo. Para se fazer uso dos transputers, os programas são enviados aos transputers pelo hospedeiro. Nestes sistemas, os transputers são usados como coprocessadores de

alto desempenho para cálculos complexos. Com o TRIX, os transputers passam a controlar o sistema, transformando o hospedeiro em um processador auxiliar, para controle dos dispositivos de entrada e saída.

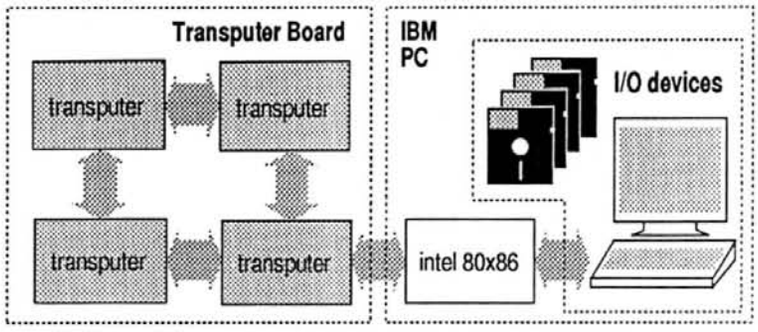

Figura 5.1: Arquitetura típica do equipamento usado no TRIX

Os dispositivos de entrada e saída podem ser controlados por qualquer processador do sistema, embora no equipamento usado somente o hospedeiro é que o faz. Isso é apenas uma restrição da arquitetura mais comum existente. Nada impede que se conecte um dispositivo de entrada e saída a algum *link* desocupado de algum transputer da rede. O TRIX prevê tal situação, ou melhor, não se prende necessariamente ao fato de o hospedeiro ser o único a ter dispositivos de entrada e saída.

Cada processador deve executar um conjunto mínimo de processos para implementar um ambiente de multiprogramação. No MINIX isto é feito pelo núcleo, que incorpora também as *tasks* (ou *device drivers*). No sistema TRIX, é desnecessário executar todas as *tasks* em cada processador, pois poucos (em geral nenhum) possuem dispositivos de entrada e saída. Mais que isso, um dos grandes defeitos do MINIX é o compartilhamento de código e de dados entre o escalonador do núcleo e as *tasks*.

Para a solução destes dois problemas, na criação do novo sistema o núcleo do MINIX foi totalmente repartido e isolado. O núcleo do TRIX é na verdade um micro-núcleo, e as *tasks* são processos totalmente independentes. O código que todos tinham em comum foi agrupado em uma biblioteca de funções, que é ligada a cada um dos módulos. Os dados em comum foram reduzidos ao máximo,

passando a ser somente a tabela local de processos, que é acessada protegida por semáforos.

Com relação à localização dos servidores, há que avaliar cada um deles isoladamente. O *Memory Manager*, por tratar do controle dos processos locais em execução e de sua memória, é necessário que exista em todos os processadores do sistema. Já o *File System*, pode ser localizado em um único processador, pois seu código foi feito para que funcionasse como um servidor (remoto) de arquivos. Ainda assim foi estudada a possibilidade de se distribuir o *File System* no sistema, por exemplo, com *caches* em cada processador. Pela maneira como ele foi escrito, esta alteração implicaria em reescrevê-lo quase totalmente. Isto não foi feito porque dispersaria a pequena força de trabalho existente.

Como em geral somente um processador possui dispositivos de entrada e saída, ele se apresenta naturalmente como um ponto centralizador das operações de entrada e saída, objeto principal do *File System*. Optou-se então por manter o *File System* original do MINIX, executando no processador mais próximo ao hospedeiro. Fica a preocupação de distribuí-lo para uma segunda fase, quando o sistema já estiver funcionando há mais tempo, possivelmente já tendo mais de um processador com dispositivos de entrada e saída, dando prosseguimento aos estudos neste sistema.

A partir do que foi exposto, pode-se classificar os processadores do sistema TRIX como sendo:

- controlador: processador que possui dispositivos de entrada e saída,
- servidor: processador que executa o *File System*, e
- cliente: processador que executa processos de usuário.

O TRIX embora possa executar *drivers* de entrada e saída, *File System* e processos de usuário no mesmo processador, não o faz. Controladores não são clientes

ou servidores porque na arquitetura atual o único controlador é o hospedeiro. O servidor não deve ser também cliente para se ter melhor desempenho e por motivos de segurança: os transputers usados não têm proteção de memória. No protótipo implementado, por haverem poucos transputers disponíveis, o servidor está funcionando também como cliente.

Os processos de usuário são executados em todos os transputers clientes da rede. Entretanto, sua existência não é perene como a do núcleo e dos servidores. Enquanto estes podem ter sua localização definida na carga do sistema, aqueles não, pois dependem das necessidades dos usuários. Para tanto, é necessário projetar um servidor de processos distribuído que equilibre dinamicamente a carga de processamento. Ao portar-se o *Memory Manager* para o TRIX, este foi dotado de recursos de criação remota de processos. Foi escrito também um mecanismo simplificado de equilíbrio de carga, para testar os novos recursos. Um trabalho mais elaborado nesse sentido pode ser agora desenvolvido, usando tais mecanismos.

Todos os processadores necessitam um número mínimo de processos executando em seu interior. São eles:

- micro-núcleo, não é um processo, mas é um trecho de código que tem que estar presente em todos os processadores, implementando a troca de mensagem em baixo nível e escalonando os processos;
- *link task*, *driver* de controle do(s) *link*(s) de comunicação;
- *system task*, *driver* de controle do processador, com funções de controle do escalonador e cópias de memória;
- *clock task*, para fazer as temporizações necessárias.

Um processador controlador somente precisa conter em sua memória o código do micro-núcleo e das *tasks*, quase sem alteração em relação ao MINIX

original, já que este foi feito para processadores do tipo Intel 80x86. Ver a figura 4.1 em comparação com a figura 5.2. Notar a existência de uma nova *task*, chamada *link task*, que será detalhada mais adiante, na seção 5.3. Por fim, a memória que restar desocupada no hospedeiro deverá ser usada para manter grandes *buffers* de E/S. Esta distribuição é semelhante à apresentada no hospedeiro do sistema DIX [STE92].

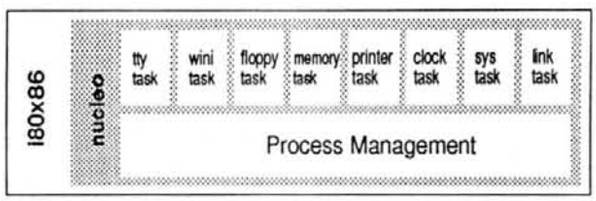

Figura 5.2: Processos do processador hospedeiro

Há um *driver* distinto para cada tipo de dispositivo conectado ao controlador. Nenhuma tarefa de usuário é executada nele, pelo menos enquanto o controlador for o hospedeiro. Nenhum servidor é executado no controlador. O *Memory Manager* não é necessário porque não existirão processos de usuário sendo criados e destruídos no hospedeiro e toda a memória disponível é inicialmente alocada pelas *tasks*. O *File System* poderia ser executado pelo hospedeiro, pois este possui todos os dispositivos de entrada e saída. Pelo mesmo motivo do projeto DIX [BAR90] — usar um processador com melhor desempenho — o *File System* será localizado no transputer mais próximo ao hospedeiro.

Nos transputers clientes, além do micro-núcleo e dos três tipos de *tasks* necessárias a todos os processadores, são executados o *Memory Manager* e os processos de usuário, conforme a figura 5.3. O *File System* executa apenas no transputer chamado servidor, que está diretamente conectado ao hospedeiro. O processo `init` poderia executar em qualquer um dos transputers existentes. Por simplicidade na carga do sistema, optou-se por colocá-lo no mesmo processador do *File System*.

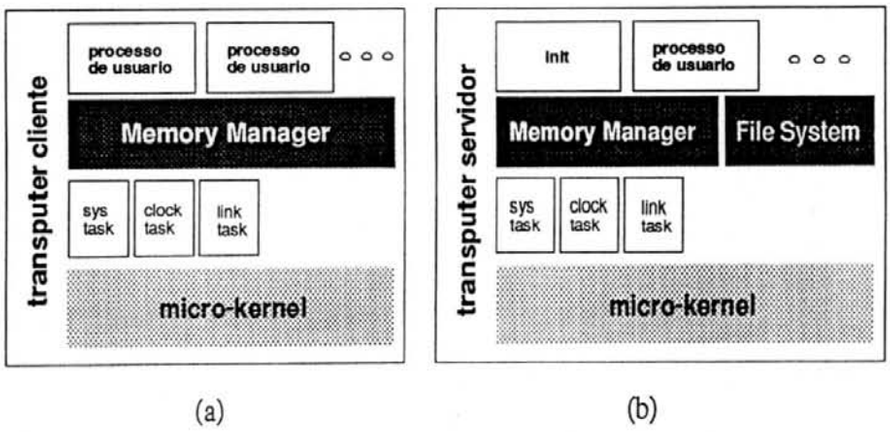

Figura 5.3: Processos dos transputers clientes (a) e servidores (b)

5.2 Mecanismo de Comunicação entre Processos

Aqui será tratado o mecanismo de envio de mensagens entre os processos do sistema. Com um único processador, como no MINIX, não existe preocupação com a localização do processo destino de uma mensagem, pois ele está sempre no mesmo processador do processo origem. No TRIX existem recursos adicionais para que as mensagens sejam identificadas como sendo para processos remotos e enviadas a outro processador.

5.2.1 Mensagens para Processos Remotos

Para se poder endereçar processos em todo o sistema, o identificador do MINIX foi aumentado. Se no MINIX o que identifica um processo é sua posição na tabela de processos do núcleo (*slot*), no TRIX é sua posição na tabela e o número de seu processador. Mais que isso, no TRIX há dois tipos de identificadores: locais e globais (vide figura 5.4). O primeiro identifica somente a posição do processo na tabela (*a la* MINIX) e o segundo identifica univocamente um processo no sistema inteiro. O novo identificador é formado por dois campos: *slot* e CPU. Quando o campo CPU for zero, o identificador é local, senão, global.

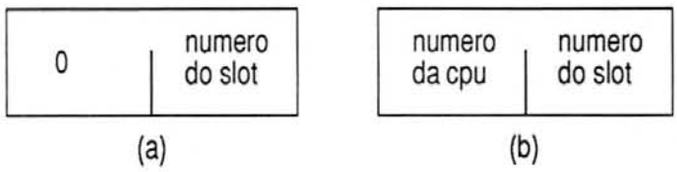

Figura 5.4: Identificadores local (a) e global (b) de processos

Quando um processo desejar se comunicar com outro, vai pedir ao núcleo que envie uma mensagem. O processo remetente aponta o destinatário através de um campo na mensagem, com um identificador local. Isto é feito para dar transparência de localidade, além de minimizar as alterações na biblioteca do MINIX. O núcleo usa uma tabela de conversão para trocar o identificador local por um global. O núcleo também insere no campo origem da mensagem o identificador global do processo que a está enviando. Se algum processo enviar alguma mensagem já contendo no destinatário um identificador global, este não precisa ser convertido.

O núcleo possui uma tabela de conversão de endereços de processos, contendo o endereço de todos os processos remotos para os quais algum processo local possa enviar uma mensagem. Dimensionar esta tabela aparentemente não é uma tarefa trivial. Qualquer núcleo precisaria ter uma tabela contendo os identificadores globais de **todos** os processos do sistema, pois todos os processos podem enviar e receber mensagens. Este tamanho pode ser difícil de se prever no momento da criação da tabela. Avaliando melhor o mecanismo de envio da mensagem, nota-se que esta tabela pode ser largamente diminuída, reduzindo-se a menos de uma dezena de entradas.

O sistema é todo montado sob o mecanismo de cliente–servidor, onde os servidores somente enviam mensagens em resposta a algum pedido. Os pedidos têm que ser feitos através de mensagens, que obrigatoriamente contêm os identificadores globais dos processos que as enviaram, inserido pelo núcleo do processo cliente. Assim, os servidores, como somente atentem a pedidos de clien-

tes, nunca usam identificadores locais nos endereços de resposta, desobrigando o núcleo de possuí-los em sua tabela de conversão.

Os processos de usuário somente enviam mensagens para o *Memory Manager* e o *File System*. O *Memory Manager* é sempre local, logo não precisa estar na tabela. O endereço do *File System* precisa estar na tabela, pois é um processo remoto, pelo menos enquanto este não for distribuído. Os processos que servem ao *Memory Manager* são as *tasks* locais e a *tty task*. Logo, somente a última precisa estar na tabela. O processador que executar o *File System* é o único que vai precisar de mais de duas entradas nesta tabela, pois terá que ter o endereço de todas as *tasks* remotas de dispositivos, que ainda assim não são muitas.

5.2.2 Tabela de Processos

Como os identificadores dos processos remotos a serem convertidos pelo núcleo não são muitos, optou-se por usar a própria tabela de processos do sistema para tal conversão. Quando um processo é local, todos seus dados, como segmentos de memória alocados, tempo de cpu gasto, etc, existem e estão guardados na tabela. Do contrário, com processos remotos, a entrada na tabela está em branco e um campo adicional indica qual o seu identificador global.

Um sistema com dois transputers e um hospedeiro i80x86 é montado como exemplo nas tabelas 5.1, 5.2 e 5.3. Nelas aparecem as tabelas de processos de cada processador, indicando os processos contidos em cada um.

O processador hospedeiro possui as *tasks link* (para controle do *link*), *printer* e *tty* (para interfaces paralela e serial), *wini* e *floppy* (para discos), *clock* (relógio) e *system* (controle do sistema). Nenhum outro processo consta da tabela de processos 5.1. Nenhuma *task* do hospedeiro enviará uma mensagem cujo destino seja um processo com um endereço local. Todas as mensagens de *tasks* são enviadas usando uma função `reply`, que envia uma mensagem de resposta a

Tabela 5.1: Tabela de processos do TRIX no hospedeiro

endereço local	nome do processo	global cpu	global slot
-9	link task	1	-9
-8	printer task	1	-8
-7	tty task	1	-7
-6	winchester task	1	-6
-5	floppy task	1	-5
-4	ramdisk task	1	-4
-3	clock task	1	-3
-2	system task	1	-2
-1	hardware	1	-1

um pedido anterior. Obrigatoriamente, um pedido contém o identificador global de seu processo origem.

O transputer servidor (CPU 2) possui em seu interior a *link task*, a *clock task*, a *system task*, o *Memory Manager*, o *File System*, o processo `init` e, hipoteticamente, dois processos de usuário. Constam em sua tabela de processos (vista na tabela 5.2), além de todos os seus processos internos, todas as *tasks* do hospedeiro, para que o seu *Memory Manager* e o seu *File System* possam enviar-lhes mensagens. Nenhum processo de usuário de fora da CPU 2 consta desta tabela porque nenhum processo desta CPU envia involuntariamente mensagens para eles. O único processo que se comunica com processos de usuário de fora desta CPU é o *File System*, mas somente sob a forma de respostas a pedidos e neste caso o identificador usado é global. Os pedidos gerados localmente por processos de usuário são atendidos localmente pelos servidores. Os pedidos gerados pelos servidores são para serem atendidos pelas *tasks*, que se não forem locais, constam na tabela como remotas.

O transputer cliente executa as *tasks link*, *clock* e *system*, o *Memory Manager* e processos de usuário (três, no exemplo da tabela). A sua tabela de processos (tabela 5.3) conta com oito entradas, sendo sete para os processos locais e mais

Tabela 5.2: Tabela de processos do TRIX no transputer servidor

endereço local	nome do processo	global cpu	global slot
-9	link task	2	-9
-8	printer task	1	-8
-7	tty task	1	-7
-6	winchester task	1	-6
-5	floppy task	1	-5
-4	ramdisk task	1	-4
-3	clock task	2	-3
-2	system task	2	-2
-1	hardware	2	-1
0	Memory Manager	2	0
1	File System	2	1
2	init	2	2
3	proc. usuário	2	3
4	proc. usuário	2	4
5	. . .	2	5

Tabela 5.3: Tabela de processos do TRIX no transputer cliente

endereço local	nome do processo	global cpu	global slot
-9	link task	3	-9
-7	tty task	1	-7
-3	clock task	3	-3
-2	system task	3	-2
-1	hardware	3	-1
0	Memory Manager	3	0
1	File System	2	1
2	proc. usuário	3	2
3	proc. usuário	3	3
4	. . .	3	4

duas: uma para o *File System* remoto e outra para a *tty task* remota, para o caso de *panic* do *Memory Manager* e para mensagens de depuração.

Isso mostra que, mesmo sem existir uma tabela global de processos, é possível fazer uma mensagem trafegar de uma CPU à outra, com transparência de localidade aos processos interlocutores.

5.3 A *Link Task*

O TRIX faz uso de instruções específicas para troca de mensagens, implementadas no *hardware* do transputer. Ocorre que estas mensagens podem precisar trafegar por mais de um processador, o que implicou em uma reforma no mecanismo de troca de mensagens do MINIX. Isto se deve ao fato de que nem o MINIX nem o transputer implementam explicitamente a comunicação entre processos de processadores diferentes. O núcleo, que é responsável por isto, teve de ser alterado, acrescentando código e tabelas adicionais, implementando um roteamento de mensagens.

Conforme descrito anteriormente, ao receber o pedido de envio de uma mensagem, o núcleo converte o identificador do processo de seu destino e insere nela o identificador do processo origem. Se o processo destino residir na mesma CPU do núcleo em questão, a mensagem é copiada ao espaço de endereçamento do devido processo, como no MINIX original. Quando o processo destino for remoto, a mensagem é entregue para um processo específico que se encarrega de enviá-la para outro processador. Reciprocamente, se uma mensagem remota chegar, com destino a algum processo local, o núcleo a entrega a este processo como se fosse enviada localmente.

Como um transputer não possui compartilhamento de memória com nenhum outro processador, toda e qualquer movimentação de dados entre dois

processadores deve ser enviada pelos seus *links*. Foi criada uma *task* para controlar o fluxo de dados entrando e saindo em cada *link*, chamada *link task*. É esta *task* que recebe do núcleo as mensagens para serem enviadas a outros processadores. Ela também recebe dos outros processadores as mensagens a serem enviadas a processos locais. Sendo a troca de mensagens mediada sempre pelo núcleo, os processos interlocutores não notam que estão em processadores diferentes.

É a *link task* também que faz as cópias de memória entre processadores. Quando um servidor precisa efetuar uma cópia de memória de um processo para outro, o servidor faz um pedido de cópia à *system task* local ao seu processador. Se ambos processos (origem e destino da cópia) estiverem no mesmo processador, os dados são copiados do espaço de endereçamento de um processo para o outro, como no MINIX. Caso contrário, uma mensagem é enviada pela *system task* para a *link task* do mesmo processador onde está o processo de origem da cópia. Esta mensagem contém um pedido de cópia remota de memória, a ser enviado pelo *link*.

Para fazer a cópia de memória, a *link task* envia para outro processador o bloco de memória, que é recebido por uma outra *link task*. Esta, por sua vez, verifica se o bloco recebido tem como destino algum processo local ao seu processador. Caso positivo, efetua a cópia para o interior da área de dados do processo, enviando uma mensagem à *system task* do processador que pediu a cópia, sinalizando o fim da mesma. Caso negativo, a *link task* envia através de outro *link*, na direção do processador de destino. Foi implementada uma função que mapeia processadores remotos em *links*, que é usada na escolha desta direção (ver seção 6.2.2).

A *link task* é implementada com cinco *threads*, de forma a ficar bloqueada concorrentemente esperando mensagens do núcleo local e esperando pedidos dos quatro *links*. As mensagens enviadas pelo núcleo local podem ser:

- `LINK_SEND`, quando o núcleo desejar enviar uma mensagem para fora do processador.
- `LINK_COPY`, enviada por uma *system task*, pedindo uma cópia de dados através do link.

Ao receber uma destas mensagens, a *link task* envia um cabeçalho através do *link*, seguido da mensagem ou do bloco de memória a ser transferido.

Ao se receber dados por um *link*, estes podem ser uma mensagem ou uma cópia de memória. Isto é identificado através de seu cabeçalho. Se esta operação tiver um destinatário fora do processador local, simplesmente é repassada para fora do processador através de um outro *link*. Se for uma mensagem com destino local, é feito um pedido ao núcleo para que a envie ao processo destino. Se o destino for local e for um `LINK_COPY`, os dados são copiados para a área do processo destino, enviando uma mensagem do tipo `TASK_REPLY` à *system task* que pediu a cópia. Esta mensagem leva no campo remetente a *link task* que inicialmente enviou a cópia. Este ajuste é necessário já que a *system task* enviou um pedido a uma *link task* específica e está esperando dela a resposta.

A *link task* possui uma tabela que determina qual o *link* usar para atingir cada processador do sistema. Atualmente esta tabela de roteamento é montada durante a carga do sistema, se mantendo estática por todo o tempo. Dependendo de como estivessem ligados os processadores, esta tabela de roteamento poderia possuir mais de um *link* que atingisse o processador destino desejado. Isto seria útil para determinar dinamicamente a rota de mensagens, como no caso de falhas de algum processador ou de alguma conexão. Atualmente, cada processador possui uma tabela, com uma única entrada para cada destino, de maneira a simplificar a sua implementação. Como o cálculo do *link* adequado para se enviar uma dada mensagem é feito por uma função da *link task*, no futuro poderá ser implementado um roteamento dinâmico, ao estilo dos roteadores da rede Internet [HED88, REK89].

5.4 Primitivas de Comunicação do Micro-núcleo

As funções para comunicação entre processos no MINIX foram descritas na seção 4.3 e sua semântica é mantida inalterada no TRIX. Entretanto, estas funções precisam de uma alteração em seu método de funcionamento de forma a suportar a distribuição dos processos em diversos processadores.

Em uma chamada do tipo `send_rec`, onde é feito um `send` em conjunto com um `receive`, o processo chamador pode ter seu interlocutor em outro processador sem que o núcleo se preocupe com isto. O desbloqueio do processo chamador somente ocorre quando chegar uma mensagem de volta do seu interlocutor. Isto pressupõe que a mensagem tenha chegado até o interlocutor e este tenha enviado uma resposta. Situação semelhante ocorre em um `receive`, porque o processo fica localmente bloqueado até receber uma mensagem.

O núcleo do TRIX, caso funcionasse exatamente como no MINIX, poderia desbloquear um processo que efetua um `send`, se e somente se, a mensagem já tiver sido recebida pelo seu processo destino. Isso se deve ao fato de o MINIX usar o método de *rendezvous* para sincronizar a comunicação entre processos (ver seção 4.3. Ocorre que quando o destino estiver a vários processadores de distância, o núcleo local não tem como saber quando a mensagem foi efetivamente entregue ou não. Mais que isso, o núcleo local não tem nem condições de saber se o processo destino já esta bloqueado em `receive`, para poder fazer o envio da mensagem.

Para resolver este problema foram estudadas duas soluções. A primeira seria criar mensagens auxiliares, de núcleo para núcleo, para saber se um processo remoto está bloqueado e para avisar que uma mensagem remota foi entregue. Outra solução seria criar *buffers* para mensagens não entregues e supor que uma mensagem foi enviada no momento em que for entregue ao núcleo local.

A primeira opção, por necessitar troca de mensagens entre os *kernels*, implica em dotá-los de mecanismos de mais alto nível, não mais só trocando men-

sagens entre processos. Quando um núcleo enviasse uma pergunta a um núcleo remoto, ele teria que ficar bloqueado esperando uma resposta. Para que o sistema não congelasse nesse ínterim, seria necessário que o micro-núcleo fosse também multi-*threaded*. Estes mecanismos iriam complicar bastante sua implementação, além de criar uma série adicional de mensagens pela rede.

A segunda opção, embora explicitamente torne necessário deixar-se uma certa quantidade de memória à disposição do núcleo sob a forma de *buffers*, simplifica bastante a sua implementação. Quando um processo faz uso da primitiva `send`, ele não bloqueia, sendo sua mensagem copiada para um *buffer* ou passada para o exterior via uma *link task*. Um processo que emitir um `receive` fica bloqueado até que haja alguma mensagem para ele nos *buffers*. Isto termina com o *rendezvous* do MINIX mas não interfere em nada em seu funcionamento porque todos os processos de usuário são obrigados a fazer `send_rec`, o que os mantêm bloqueados até o retorno da resposta do servidor.

As ocasiões em que o `send` é usado são as seguintes: em um *reply* de um servidor e em interrupções. No primeiro caso, o destinatário da mensagem certamente está bloqueado esperando a resposta, pois fez um pedido usando `send_rec`, logo, o resultado é idêntico ao obtido com *rendezvous*. No segundo caso, o resultado é diferente, porém até mais vantajoso. No MINIX existem funções para evitar *deadlocks* e para armazenar interrupções pendentes (ainda não recebidas), que se tornam desnecessárias com o não-bloqueio do `send`.

No MINIX as mensagens têm um tamanho fixo, igual ao tamanho da maior mensagem existente. Com o porte do sistema para um processador de 32 bits, aumentou-se também o tamanho das mensagens. Não obstante, no TRIX as mensagens comumente trafegam de um processador a outro, sendo importante não transmitir dados desnecessários para não haver desperdício da taxa de transferência da rede de interconexão. Por isso, as primitivas de troca de mensagens foram alteradas para suportar mensagens de tamanho variável.

5.5 Gerência de Processos

Como já se pôde notar, as mudanças mais marcantes do MINIX para a criação do TRIX são feitas no núcleo que executa nos transputers. Como o transputer possui um escalonador de processos embutido, as funções do *Process Management* do MINIX que controlam o tratamento de interrupções, salvamento e restauração de registradores desaparecem. O fluxo pricipal de execução do núcleo se mantém, onde são atendidas as chamadas ao sistema operacional, com pedidos do tipo `send`, `receive` e `send_rec`. Nesta seção serão descritos os mecanismos do TRIX para gerência de processos.

5.5.1 Despacho e Bloqueio de Processos

No transputer, a instrução `runp` ordena ao escalonador em *hardware* o disparo de uma nova *thread*, tendo como efeito a sua colocação ao fim de uma fila controlada pelo próprio escalonador. Ela é usada pelo núcleo para criação de processos de usuário. As instruções `in` e `out` bloqueiam uma *thread* à espera da tranferência de uma mensagem e são usadas para processos de usuário enviarem chamadas ao núcleo.

Cada processo ao ser disparado recebe um canal de comunicação do transputer, que será usado para enviar ao núcleo seus pedidos de envio e recepção de mensagens, que são as chamadas ao sistema. Esta operação, chamada `sys_call`, que era feita no MINIX convencional através de uma interrupção de *software* (ou *trap*), no TRIX é feita quando o processo enviar por seu canal um cabeçalho contendo as seguintes informações: operação (`send`, `receive` ou `send_rec`), processo interlocutor (destino do `send` ou origem do `receive`) e posição do *buffer* da mensagem.

A função de nome `sys_call` do núcleo do MINIX que é chamada no momento da interrupção do processo chamador, passa a ter um acionamento diferente. Em vez de ser chamada através de uma interrupção e retornar o controle ao processo chamador após processar o pedido, executa-se um laço de repetição, sempre esperando cabeçalhos enviados por processos. A função recebe um pedido, processa-o, e volta a esperar pedidos.

Para executar esta rotina, que na verdade é o micro-núcleo do TRIX, é criada uma *thread* do transputer, sem criar nenhum processo sob o ponto de vista do sistema. Esta *thread* se bloqueia através de uma instrução `alt` em um conjunto de alternativas — todos os canais dos processos de usuário — à espera de uma chamada de sistema. Ao receber esta chamada, que é enviada por uma instrução `out`, o micro-núcleo desbloqueia-se e ganha o controle da CPU. Deste momento em diante, o seu procedimento é praticamente o mesmo que quando ocorre uma interrupção no MINIX. Como já foi visto na seção 5.4, existe aí uma diferença entre os dois sistemas, pois no TRIX um `send` não bloqueia o processo chamador, sendo a mensagem copiada para um *buffer*.

Como no MINIX as funções `mini_send` e `mini_rec` são chamadas quando a `sys_call` do núcleo recebe pedidos de `send` e `receive` respectivamente (com `send_rec` chamam-se as duas). Elas executam no núcleo as instruções do transputer que automaticamente desbloqueiam os processos destinatários bloqueados em `RECEIVE`, inserindo-os na fila do escalonador do transputer. As funções `ready` e `unready`, que no MINIX manipulam as filas do escalonador, perdem a sua função porque no TRIX o próprio *hardware* bloqueia e desbloqueia os processos. Elas foram preservadas apenas para manter a tabela de processos atualizada, à exemplo da tabela do MINIX, onde processos bloqueados aparecem com estado `SEND` ou `RECEIVE`.

Não existe mais diferença, sob o ponto de vista do micro-núcleo, se um processo está executando ou na fila dos prontos para executar (`RUNNING` ou `READY`, no MINIX). Quem faz o escalonamento, inclusive com *round-robin* é

o *hardware* do transputer. Assim as funções `pick_proc` — para selecionar o próximo processo a pronto para executar — e `sched` — para trocar processos de usuário no fim de um *timeslice* — desapareceram.

Os mecanismos para disparo e término de processos estão implementados na *system task*, nas funções `do_fork`, `do_newmap` e `do_xit`. A primeira função chama a função `ready` para inserir o processo na fila dos prontos. Como essa tarefa é implementada pelo *hardware* do transputer, e a *system task* não tem mais acesso ao código do núcleo, isto é feito com uma instrução `runp`. A função `do_xit` ao seu final, além de atualizar a tabela de processos, libera o canal de comunicação do transputer que estava alocado ao processo. Esta liberação é importante para que o núcleo não mais atenda chamadas a partir de tal canal.

5.5.2 Envio de Sinais

Em um sistema padrão UNIX, um processo ao receber um sinal, desvia sua execução para uma de suas funções. Isto é implementado inserindo-se na sua pilha as informações necessárias para simular uma chamada de uma função e alterando o endereço em execução para o início da função de tratamento de sinal. No TRIX, quando a *system task* toma o controle do processador para enviar um sinal a um processo, este processo tem que estar bloqueado ou na fila dos prontos. Ocorre que no transputer um processo de alta prioridade pode tomar o processador de um processo de baixa prioridade, deixando seus registradores na `RegSaveArea` (ver seção 3.2). Como esta área não pode ser alterada por *software*, o processo ali guardado não pode ser o mesmo a ser sinalizado.

A *system task* possui uma função auxiliar chamada `do_synchr`, para poder retirar processos em execução no escalonador do *hardware*. Esta função dispara uma *thread* de baixa prioridade e pára a *thread* que atende a *system task*, guardando o seu estado em uma área pré-definida. Esta nova *thread*, chamada

`synchr_thread` somente faz duas coisas: disparar de novo a *thread* principal, com prioridade urgente e parar a si mesma. Quando a *system task* quiser se certificar que nenhum processo não-urgente útil está na `RegSaveArea`, ela executa a função `do_synchr`. Isto pára a *system task* e dispara a `synchr_thread`. São executados todos os processos não-urgentes da fila até a `synchr_thread` ganhar o processador, que dispara de novo a *thread* urgente. Com certeza, neste instante o processo em execução será a *system task* e o processo preso na `RegSaveArea` será a *thread* `synchr_thread`. Este procedimento também é executado quando a *system task* receber uma mensagem do tipo `SYS_SYNCHR`, deixando este mecanismo disponível a qualquer *task* do sistema.

5.5.3 Mecanismo de *Shadowing*

No sistema o UNIX, a forma de disparar processos é a chamada `fork` do sistema. Esta chamada cria um processo que é uma réplica exata do processo que a chamou. O processo recém-criado em geral faz a chamada `exec` do sistema, que executa um novo programa, com novos segmentos de memória alocados para tal.

É extremamente comum existirem ponteiros na área de dados do processo a ser duplicado, os quais no processo réplica devem apontar para a sua própria área de dados. Os transputers atualmente em uso não possuem mecanismos de gerência de memória, não permitindo a relocação da área de dados de um programa durante sua execução. Um ponteiro é um endereço absoluto de memória. Ao mover a área de dados de um processo seria necessário alterar **todos** os seus ponteiros, o que é impossível, pois em linguagens como C nem sempre se sabe o tipo dos dados que compõem um programa.

Para resolver este problema, existem duas soluções. Pode-se alterar o compilador C para somar um deslocamento à uma base relocável em todos os acessos a dados na memória. Isto foi usado no DISTRIX [MCC90] e foi notada uma

perda de 30% no desempenho do sistema. A segunda solução é implementar um mecanismo de *shadowing*, que já é normalmente usado nas versões do MINIX para processadores Motorola, sem unidade de gerência de memória.

Shadowing consiste em manter ambos processos, original e réplica executando no mesmo espaço de memória. Isto é implementado copiando a área de dados do processo a ser duplicado para uma área de memória chamada *shadow*, deixando-se o novo processo bloqueado. Após um certo intervalo de tempo, é efetuada uma troca, passando os dados da nova réplica para o local correto e o bloqueando o original na área de *shadow*. Esta solução, em termos gerais é bem mais lenta que a anterior. Ocorre que em quase todas as vezes que se executa um `fork`, o processo réplica executa um `exec`, rodando um novo programa. O *shadowing* não precisa mais ser mantido após o `exec`, pois o novo programa possui uma nova área de dados.

Como um processo já replicado pode ainda se replicar várias vezes, é necessária uma lista unindo os processos que estão em um único *shadow* disputando pelo mesmo espaço de memória. A tabela de processos tem um *flag* adicional em relação à do MINIX para indicar se o processo está ou não no *shadow* e um *buffer* adicional para armazenar alguma mensagem que porventura seja enviada ao processo, também caso este esteja no *shadow*. A *system task* além de incluir o manuseio do *shadow* nas funções de disparo e término de processos, possui uma nova função, que é a de trocar todos os *shadows* atualmente em execução. Esta função, chamada `do_shadow` é acionada por uma mensagem `SYS_SHADOW` enviada periodicamente pela *clock task*, trocando todos os processos pelo próximo da lista, de forma circular.

A função `do_shadow`, como o envio de um sinal, não pode ser executada no instante em que o processo em execução estiver entre os que devem ser copiados. Como este problema é similar ao da seção 5.5.2, porque os registradores do processo estarão presos na `RegSaveArea`, a solução adotada será a mesma:

usar a função `do_syncr`, que garante que o processo em execução no momento não tem relação com o *shadowing*.

5.5.4 Contabilização de Tempo

A *clock task*, como no MINIX, controla os eventos dependentes do tempo. Esta *task* tem as mesmas chamadas da *clock task* original do MINIX. A função `do_clocktick` foi alterada para, em vez de esperar por uma interrupção do temporizador, usar a instrução *timer input*. Esta função também controla a troca de *shadows*, enviando mensagens à *system task* após esperar um intervalo de tempo (de vários pulsos de relógio).

No UNIX, um processo acumula o tempo real de execução de um processo em dois contadores. Um para o tempo gasto executando código do usuário e outro para o sistema. No MINIX, 60 vezes por segundo a *clock task* recebe um pulso do relógio, na forma de uma mensagem. A cada pulso do relógio, o tempo de usuário do processo em execução é incrementado. Caso o processo interrompido seja um processo de sistema, o tempo de sistema do último processo de usuário executado é incrementado.

No transputer, os pulsos da *clock task* serão muito maiores que os pulsos do escalonador de processos. Cada vez que um pulso do relógio chegar à *clock task*, o *escalonador* já terá escalonado diversos processos. Isto inviabiliza o uso do mesmo algoritmo do MINIX pela perda de precisão.

A contabilização do tempo de processos não é mais feita pela *clock task*. O próprio micro-núcleo executa esta função, calculando o tempo decorrido entre duas chamadas ao sistema. Se o processo chamador for urgente, certamente ele é o único processo que executou desde a última `sys_call`, sendo nele acumulado o tempo total. Senão, o tempo decorrido é repartido entre todos os processos não

urgentes que estiverem no estado READY, que provavelmente receberam quinhões parecidos de tempo de CPU no intervalo.

5.6 Controle Distribuído da Árvore de Processos

O *Memory Manager* implementa as primitivas UNIX relacionadas com processos e memória. Fazendo uso de chamadas de baixo nível à *system task*, ele controla toda a alocação de memória de cada processo, cria e destrói processos, envia sinais, etc. Todas as funções originais do *Memory Manager* do MINIX são mantidas, quando se referem à memória e aos processos locais. As funções de criação e término de processos foram reescritas, incluindo mensagens adicionais para controlar de forma **distribuída** a árvore de processos.

Para implementar uma árvore de processos como a do UNIX de forma distribuída, cada entrada na tabela de processos do *Memory Manager* do TRIX possui os seguintes campos:

- `mp_child`, endereço global apontando para o último processo filho criado por este;
- `mp_brother`, endereço global apontando para o processo filho do mesmo pai, que foi disparado antes deste;
- `mp_parent`, endereço global do processo pai.

A figura 5.5 mostra um exemplo de árvore de processos do TRIX, que explica graficamente o uso dos campos descritos acima. Nela, há dois processadores, um com três e outro com quatro processos. O processo `init` possui três filhos: `-sh`, `-csh` e `login`. O `-sh` por sua vez criou o filho `ls`, o `-csh` criou `cc` e `ps` e o `login` não possui filhos.

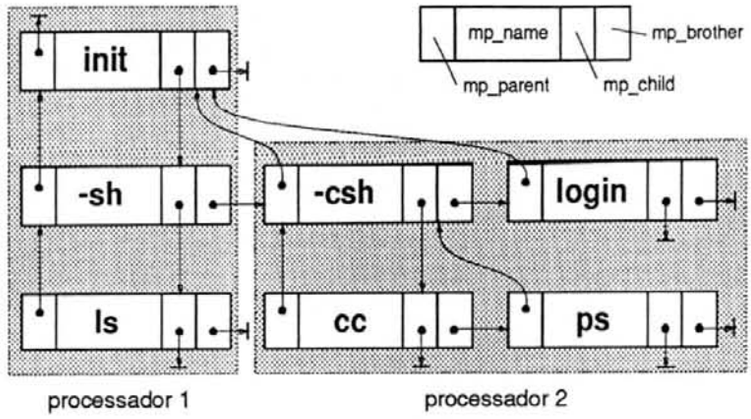

Figura 5.5: Um exemplo de árvore de processos

5.6.1 Mensagens de Controle

O controle distribuído desta árvore é feito através de mensagens, entre os *Memory Managers* envolvidos. Tais mensagens são enviadas à medida que forem criados ou terminados processos que possuam algum pai, filho ou irmão remoto. Para evitar *deadlocks*, não são esperadas respostas para nenhuma destas mensagens. São elas:

- `RFORK`: pede ao *Memory Manager* remoto para executar um processo;
- `RYFORK`: processo executado remotamente (`RFORK` obteve sucesso);
- `RNFORK`: processo não executado remotamente (`RFORK` falhou);
- `REXIT`: filho terminou, envia codigo de retorno ao pai;
- `RWAIT`: pai executou `wait`, para esperar filhos terminarem;
- `FRESLT`: pai recebeu código de `exit`, pode liberar entrada na tabela;
- `ORPHAN`: processo com filhos morreu, seus filhos passam a apontar para o `init` como pai;
- `INHERIT`: processo com filhos morreu, filhos do `init` recebem seus filhos como irmãos.

Quando um processo fizer a chamada `fork`, o seu *Memory Manager* vai escolher um processador para disparar o novo processo (ver seção 5.6.2). O *Memory Manager* local envia então uma mensagem `RFORK` para o *Memory Manager* do processador escolhido. Tal *Memory Manager* remoto avalia sua possibilidade de executar o novo processo, respondendo com `RYFORK` ou `RNFORK`. Se a resposta for positiva, o processo já terá sido criado remotamente. Caso contrário, o processo é criado localmente.

Ao terminar, um processo executa a chamada `exit` do sistema. Tal chamada vai fazer o *Memory Manager* local enviar uma mensagem do tipo `REXIT` ao *Memory Manager* do processo pai do chamador. Se o processo pai estiver bloqueado em uma chamada `wait` para esperar a morte de seus filhos, o *Memory Manager* do processo pai envia uma mensagem `FRESLT` para o *Memory Manager* do processo filho que morreu. Se o pai não estiver bloqueado, a mensagem `REXIT` é ignorada. Quando um processo que tem filhos bloquear na chamada `wait`, seu *Memory Manager* envia uma mensagem `RWAIT` ao *Memory Manager* do seu primeiro filho. O *Memory Manager* de cada filho repassa sucessivamente a mensagem ao *Memory Manager* do irmão. Caso um dos processos filhos já tenha chamado `exit` uma nova mensagem `REXIT` é enviada ao *Memory Manager* do pai.

Quando um processo que possui filhos morre, o seu *Memory Manager* envia uma mensagem `INHERIT` para o *Memory Manager* do processo `init` e uma mensagem `ORPHAN` para o *Memory Manager* de seu primeiro filho. A mensagem `INHERIT` percorre a lista de filhos do `init`, acrescentando ao seu final a lista de filhos do processo morto. Concorrentemente, a mensagem `ORPHAN` percorre a lista dos filhos órfãos, alterando seu pai como sendo o `init`.

5.6.2 Distribuição da Carga de Processamento

O *Memory Manager* do TRIX implementa a distribuição dos processos de usuários pelo sistema, tentando manter um equilíbrio da carga de processamento. Este equilíbrio é implementado em um módulo fonte independente do controle da árvore do *Memory Manager*, podendo ser alterado para experimentar diferentes modelos. O modelo implementado neste trabalho é extremamente simples, já que sua finalidade é de testar o controle distribuído da árvore, mesmo que não efetue sempre um bom equilíbrio.

A idéia base é um modelo publicado em [LEI91], já estudado e implementado no CPGCC da UFRGS na dissertação [BEL93], combinado com o mecanismo de Gradiente [LIN87]. O mecanismo implementado é totalmente distribuído, onde cada processador somente mantém o endereço do processador menos ocupado mais próximo. Quando o processador local precisar criar um novo processo, faz um pedido a tal processador. A idéia se aplica bem a uma rede de transputers e também à árvore distribuída do *Memory Manager*.

Cada *Memory Manager* possui uma variável que diz **qual** o processador que está com menos carga, e outra com a distância até tal processador. Quando for necessário disparar um novo processo, o *Memory Manager* local envia um RFORK para tal processador. Se aquele processador estiver apto a receber um novo processo, cria-o e responde com `RYFORK` senão, responde com `RNFORK`.

A cada intervalo de tempo a *clock task* envia para o *Memory Manager* uma taxa de ocupação média da CPU. Se esta taxa for menor que a taxa conhecida como menor, ele substitui a antiga e envia mensagens para os *Memory Managers* de todos os processadores vizinhos, dizendo estar à distância zero de um processador com a sua carga.

Cada *Memory Manager* ao receber uma mensagem do tipo descrito, compara a medida de carga recebida com a sua variável. Se a medida for menor

que a conhecida, a conhecida é alterada, a distância incrementada de 1 e a mensagem repassada para todos os vizinhos. Todos *Memory Managers* se comportam da mesma forma, acontecendo uma espécie de *broadcast* por difusão.

Qualquer mensagem com uma taxa de ocupação igual a que é atualmente conhecida como menor, tem a distância comparada. Se a distância for menor que a conhecida, o procedimento ainda se repete, senão a mensagem é descartada, parando de propagar. Caso seja recebida uma taxa de ocupação maior que a atual, o *Memory Manager* local responde, somente ao último *Memory Manager* que a enviou, com a **sua** taxa de ocupação (a menor), e distância zero, reiniciando um ciclo.

6 IMPLEMENTAÇÃO DO SISTEMA

6.1 O Núcleo do Sistema

Mesmo implementando processos concorrentes e a comunicação entre eles, o *hardware* do transputer não provê todos os mecanismos necessários para dar suporte completo a um sistema como o UNIX. Há que suprir com *software* esta deficiência, implementando funções como, por exemplo, controle de sinais, alocação de memória e contabilização do tempo de CPU por processo.

Em segundo lugar, o mecanismo de comunicação entre os processos do MINIX é bem mais sofisticado que o do *hardware* do transputer. Nele, o sentido do envio das mensagens é rigorosamente controlado: processos comuns somente usam `send_rec`, servidores e *tasks* usam `send`, `receive` ou `send_rec`. No TRIX as mensagens têm tamanho variável (ver seção 5.4), logo, há que controlar também o tamanho de uma mensagem enviada, que não pode ultrapassar o tamanho do *buffer* alocado no processo de recepção. Nenhum destes cuidados é tomado pelo *hardware* de um transputer.

Por último, há que embutir no núcleo a capacidade de transportar mensagens entre processos de processadores diferentes. Por se caracterizar como um sistema distribuído, o TRIX provê um mecanismo de troca de mensagens transprocessador. Este mecanismo é apenas parcialmente suportado pelo *hardware*, ao permitir que se use as mesmas intruções `in` e `out` para se enviar mensagens por *links* ou por canais. Porém o *hardware* não faz roteamento de mensagens entre processos localizados em processadores que não estejam diretamente conectados.

6.1.1 Mecanismo de Comunicação

Todos os processos do sistema (*tasks*, servidores, processos de usuário) trocam mensagens usando as segintes funções:

- `send`, envia mensagem,
- `receive`, recebe mensagem,
- `send_rec`, envia e depois recebe mensagem,
- `vsend`, envia mensagem com tamanho variável,
- `vreceive`, recebe mensagem com tamanho variável e
- `vsend_rec`, envia e depois recebe mensagem com tamanho variável.

Estas funções foram inseridas (como no MINIX) na biblioteca padrão do sistema e todas as funções que utilizam um servidor enviam mensagens a estes através daquelas. O código destas seis funções de envio e recepção de mensagens é praticamente o mesmo, simplesmente enviando um cabeçalho para o *kernel*. Após chamar alguma dessas funções o processo perde a CPU e o núcleo entra em execução. A figura 6.1 mostra como exemplo o código da função `send`, onde a função `ChanOut` executa a instrução `out` do transputer, usando o canal `my_channel`, que é uma variável global que representa o canal do processo, inicializada ao dispará-lo.

Todas seis funções retornam um valor que indica o sucesso ou o motivo da falha no envio da mensagem (representado pela variável `rc` na figura 6.1). Esta variável é atualidada pelo núcleo pouco antes de desbloquear o processo. Sua posição na memória não é fixa, pois é uma variável do tipo automática, mas como ela é sempre criada na mesma posição em todas as seis funções, o núcleo encontra seu endereço subtraindo uma constante do valor da pilha do processo no momento da chamada.

```
int send(int dest, message *ptr) {
  int rc;    /* must be the first var declared */
  sendrec_hdr hdr;

  hdr.src_dst = dest;
  hdr.function = SEND;
  hdr.size = MESS_SIZE;
  hdr.m_ptr  = ptr;
  ChanOut(my_channel, (char *)&hdr, sizeof(hdr));
  return rc;
}
```

Figura 6.1: Função send da biblioteca do TRIX

Uma mensagem contém os campos descritos na figura 6.2. O campo m_source contém o endereço do processo que a enviou. O campo m_dest contém o endereço do processo que deve recebê-la. m_size indica o tamanho total da mensagem em bytes, incluindo estes três primeiros campos. O resto da mensagem, contido em m_body[] é definido da maneira que melhor convir pelos processos interlocutores. Em todos os casos, a primeira palavra (m_body[0]) identifica o tipo da mensagem (há uma macro chamada m_type definida para isso).

```
typedef struct {
  long m_source;
  long m_dest;
  long m_size;
  long m_body[MESSAGE_BODY_WORDS];
} message;

#define m_type    m_body[0]
```

Figura 6.2: A mensagem do TRIX

O mecanismo de comunicação dentro do núcleo do TRIX, no que se refere a mensagens **internas** ao processador, está todo contido no módulo `proc.c`[1]. Este módulo é executado com uma *thread* do transputer logo após o *boot*, permanecendo em execução até o fim dos tempos, seguindo o algoritmo da figura 6.3.

repete sempre:
    espera comunicação por um canal
    contabiliza o tempo
    põe processo em estado `BLOCKED`
    recebe cabeçalho da mensagem
    transforma endereço dado na mensagem (destino do `SEND` ou origem do `RECEIVE`) em endereço global
    se houver alguma inconsistência do cabeçalho
      não envia a mensagem
      devolve processo ao estado `READY`)
    se chamada possui `SEND`
      executa `mini_send()`
    se chamada possui `RECEIVE`
      executa `mini_rec()`
    contabiliza o tempo

Figura 6.3: Algoritmo principal do micro-núcleo

As funções chamadas pelo laço principal do micro-núcleo (`mini_send` e `mini_rec`) fazem a tranferência da mensagem do espaço de endereçamento de um processo para o outro. Quando o processo destinatário não está bloqueado recebendo, a mensagem é copiada para um *buffer*. Isso não existia no MINIX original e teve que ser acrescentado. As figuras 6.4 e 6.5 mostram o algoritmo destas funções.

[1]Como o trecho de programa sendo apresentado é o resultado da alteração do MINIX, seu código não está publicado, para preservar o *Copyright*. Nestes casos, o código será apresentado em notação algorítmica.

```
mini_send:
    se a origem da mensagem é a Link Task
      assume a origem como sendo aquela contida na mensagem pois
esta é uma mensagem chegando do exterior
    senão
      assume a origem como sendo o endereço global do remetente
    se o processo destino da mensagem não está na CPU local
      assume Link Task como sendo o processo destino
    se não há mais buffers, o processo bloqueará, logo
      se haverá deadlock
            retorna ao chamador, com erro
    se o destinatário está bloqueado em RECEIVE
      copia mensagem para o destino, despachando remetente
      despacha destinatário
    senão
      se há buffer livre
            copia mensagem para o buffer, despachando remetente
            põe buffer no fim da lista do destinatátio
      senão
            bloqueia o remetente esperando o destinatário receber
```

Figura 6.4: Envio de mensagens no micro-núcleo

```
mini_rec:
    se a origem da mensagem é o exterior
      assume a origem como sendo a link task
    senão
      assume a origem como sendo o endereço global do remetente
    se há alguma mensagem em sua lista
      copia a mensagem para o destino
      libera o buffer da lista
    senão
      se a origem é local e tal processo está bloqueado ao enviar uma
mensagem ao recebedor
          copia a mensagem de um processo a outro
          despacha o remetente
      senão
          bloqueia o destinatário esperando remetente enviar
```

Figura 6.5: Recepção de mensagens no micro-núcleo

6.1.2 Tempo de CPU por Processo

A mesmo trecho de código que faz a troca de mensagens entre os processos é aquele que conta tempo de CPU gasto por cada processo. Os detalhes referentes a isso na descrição da função `proc` na seção anterior foram omitidos por questão de clareza. Na figura 6.6 está representado o algoritmo que contabiliza o tempo, agora sem a parte referente às mensagens, pelo mesmo motivo. A operação *calcula diferença de horário* a que faz referência a figura 6.6 significa calcular o tempo decorrido desde a última vez em que a mesma operação foi feita.

6.1.3 *Drivers* Independentes do Núcleo

O conceito de núcleo em *software* é mantido no TRIX para implementar o que foi exposto nas seções 6.1.1 e 6.1.2. Porém o núcleo original do MINIX é

```
repete sempre:
    espera comunicação por um canal
    TU = calcula diferença de horário
    se o processo que está se comunicando é urgente
      acresce TU ao seu tempo de usuário
    senão
      acresce o tempo de usuário de todos processos não-urgentes no
estado READY)
    guarda hora atual
    envia e/ou recebe mensagens (ver figura 6.3)
    TS = calcula diferença de horário
    acresce TS ao tempo de sistema do processo sendo atendido
    guarda hora atual
```

Figura 6.6: Contabilização do tempo de CPU no micro-núcleo

formado por uma miscelânea de funções, que são disparadas concorrentemente sob a forma de vários processos compartilhando dados e código. Estas funções, como descrito na seção 4.1, receberam a denominação de *tasks*. Embora estas *tasks* estejam em módulos separados, elas usam funções de outras *tasks*, violando a individualidade dos processos.

No TRIX foi decidido que não deveria haver nenhum processo desta forma, sendo obrigatoriamente todos independentes em relação a sua área de memória. Com isso, as *tasks* tiveram que ser reescritas para funcionarem como processos independentes, passando o núcleo a ser chamado a partir de então de micro-núcleo. Todas funções comuns a mais de uma *task* foram inseridas em uma nova biblioteca, chamada `klib` (*kernel library*).

6.1.4 Semáforos

Embora as *tasks* sejam processos independentes, elas precisam compartilhar alguns dados entre si e com o núcleo. Para controlar os acessos a eses dados compartilhados, que são mínimos, foram criados semáforos. Assim, cada informação que for acessada por mais de uma *task* e for alterada por pelo menos uma delas possui um semáforo associado.

Para definir semáforos no TRIX foram criadas as estruturas de dados da figura 6.7. Para cada objeto a se controlar a concorrência, é criada uma estrutura do tipo `_semaphore`. Esta estrutura tem um contador de ocupação `count` e uma fila de processos esperando para acessar a região crítica. Para cada processo na fila de espera do semáforo é adicionada uma estrutura do tipo `_semaphore_queue`, que contém o descritor do processo parado (`function`, que nada mais é que seu `Wptr`) e um apontador para o próximo processo da fila (`next`).

```
typedef struct _semaphore_queue {
  PDes function;
  struct _semaphore_queue *next;
} semaphore_queue;

typedef struct _semaphore {
  int count;
  semaphore_queue *queue;
} semaphore;
```

Figura 6.7: Estruturas de dados dos semáforos

Na biblioteca `klib` foram embutidas algumas funções para o tratamento de semáforos, apresentadas na figura 6.8. Quando um processo deseja acessar uma região crítica de um objeto, chama a função `semap_p`. Esta função incrementa o número de processos acessando o semáforo, representado pelo campo `count`. Se já havia algum processo acessando sua região crítica o processo chamador executará a função `halt_thread`, que é implementada tão somente pela instrução `stopp` do transputer (ver seção 3.2). Esta instrução guarda o `Iptr` da

thread em seu `Wptr[-1]` e guarda o seu `Wptr` em um endereço passado como parâmetro. No caso, o endereço dado é o campo `function` de uma estrutura `_semaphore_queue` alocada na própria função `semap_p`.

```
void semap_p(semaphore *s) {
  semaphore_queue *q, t;

  if (s->count++) {
      if (q = s->queue) {
          while (q->next) q = q->next;
          q->next = &t;
      }
      else s->queue = &t;
      t.next = 0;
      halt_thread(&t.function);
      s->queue = s->queue->next;
  }
}

void semap_v(semaphore *s) {
  if (--s->count) restart_thread(s->queue->function);
}
```

Figura 6.8: Funções que implementam semáforos

Ao sair de uma região crítica, os processos chamam a função `semap_v`, também mostrada na figura 6.8. Esta função decrementa o valor do campo `count`, que se não resultar em zero, significa que há processos esperando na fila do semáforo. Neste caso, é executada a função `restart_thread` que é a própria instrução `runp` do transputer. Como já foi visto na seção 3.2, a instrução `runp` dispara uma *thread* usando o parâmetro como `Wptr` da nova *thread*, cujo `Iptr` deve estar armazenado em `Wptr[-1]`.

6.1.5 Interrupções

No MINIX original, todo o tipo de interrupção gerado por *hardware* é atendido por um pequeno tratador que envia uma mensagem a um processo. Por

exemplo, quando o controlador de disco terminou de fazer uma leitura ele gera uma interrupção no processador. Nesse momento é enviada uma mensagem à *winchester task* para que esta possa enviar uma resposta ao processo que pediu a leitura. Para enviar este tipo de mensagens iniciadas por interrupções, o MINIX possui um processo especial, de nome *hardware*. Este mesmo processo também é usado quando uma *task* desejar enviar um sinal a um processo. A *task* envia uma mensagem com origem no *hardware* e destino no *Memory Manager*.

Em transputers, não existem interrupções. Todo o tipo de atividade que necessitar de controle por interrupções terá que ser feito em mais alto nível, com mensagens. Logo, no TRIX não há nenhum código para trocar interrupções por mensagens. Porém, não se pode retirar o processo *hardware* do sistema porque as *tasks* podem precisar enviar sinais (tipo *alarm* ou *break*) para algum processo.

Como no TRIX o núcleo identifica o processo que está enviando uma mensagem pelo canal que ele se manifesta, enviar uma mensagem em nome do processo *hardware* significa usar seu canal para enviar o *header* da mensagem. Como mais de uma *task* pode estar desejando enviar mensagens em nome do *hardware*, um semáforo foi criado para regular seu acesso. O trecho de código que finge que o remetente de uma mensagem é o *hardware* está na função `hsend` (figura 6.9), que também pertence à biblioteca `klib`.

6.2 Processo de Controle do *Link*

Cada dispositivo conectado a um dado processador possui uma *task* associada. No caso dos transputers, há duas maneiras de se conectar um dispositivo: em um *link*, através de um *link adaptor* [INM88a] ou mapeando dispositivos em endereços de memória. Embora o transputer usado no porte possua conexões para mapear dispositivos em memória, somente seus *links* estão conectados. Mais

```
int hsend(int dest, message *ptr, int size) {
  sendrec_hdr hdr;
  extern Channel *hardware_chan;
  extern semaphore* hardware_sem;

  hdr.src_dst = dest;
  hdr.size = size;
  hdr.function  = SEND;
  hdr.m_ptr  = ptr;

  semap_p(hardware_sem);
  ChanOut(hardware_chan, (char *)&hdr, sizeof(hdr));
  semap_v(hardware_sem);
  return OK;
}
```

Figura 6.9: Função para enviar mensagem em nome do *hardware*

que isso, todos os seus *links* são usados para estabelecer a comunicação entre os processadores do sistema e não para conectar dispositivos.

A *link task* foi projetada para controlar os *links* do transputer como mecanismo de comunicação interprocessador. Ela envia e recebe cópias de memória e mensagens de e para outros processadores. Pode controlar um número arbitrário de *links*, tornando possível a instalação de mais que quatro *links* ou até mesmo o uso de algum *link* para conexão de um dispositivo de entrada e saída. Uma instalação que possua dispositivos de entrada e saída ligados a *links* de transputers terá que escrever um *driver* para controlar tal dispositivo, pois a *link task* somente serve para comunicação interprocessador.

6.2.1 Envio de Dados para Outro Processador

A *link task* recebe dois tipos de mensagens: `LINK_SEND` e `SYS_COPY`. Ao receber uma mensagem que não é destinada à própria *task*, o seu tipo é entendido como `LINK_SEND`, ou seja, é uma mensagem para ser enviada para fora do pro-

cessador. Se a mensagem for para a *task*, só pode ser do tipo SYS_COPY, ou seja, um pedido de transferência de um bloco de memória para fora do processador. O código da rotina principal da *link task* está na figura 6.10.

```
void main() {
/* Main program of link task. It receives a message and
 * dispatches the procedure to treat it.
 */
  int opcode, res;
  message mc;

  init_link();        /* initialize link tables & globals */
  while (TRUE) {
    receive(ANY, &mc);  /* waits for a message */
    /* Test if message is in fact to the task, or to the link.
     * thistask contains the global address of this link task.
     */
    if (mc.m_dest != thistask)
      opcode = LINK_SEND;
    else
      opcode = mc.m_type;

    switch (opcode) {
      case LINK_SEND:  res = do_linksend(&mc);  break;
      case SYS_COPY:   res = do_linkcopy(&mc);  break;
      default:         res = EINVAL;      break;
    }
  }
}
```

Figura 6.10: Programa principal da *link task*

Como foi descrito na seção 6.1, o núcleo ao perceber que uma mensagem tem como destino um processo remoto, entrega esta à *link task*. Esta *task* envia a mensagem através de um *link* para outro processador, onde é recebida por outra *link task* idêntica. Efeito parecido é o que ocorre com relação às cópias de memória, onde a *system task* repassa à *link task* os pedidos remotos.

Para diferenciar a informação enviada pelo *link* como mensagem ou cópia de memória, a *link task* envia uma palavra adicional antes dos dados. Esta palavra pode assumir os seguintes valores, dependendo do que for ser enviado:

- `SYS_COPY`: os dados a seguir são uma cópia de memória. Esta palavra é seguida de uma estrutura (figura 6.11) que define de onde e para onde está sendo feita a cópia.

- `LINK_SEND`: os dados a seguir são uma mensagem, conforme a figura 6.2.

- `LINK_BOOT`: os dados a seguir são para serem enviados diretamente, sem tratamento, através de um outro *link*. Isso serve para dar a carga (*boot*) em um processador que não estiver diretamente conectado ao hospedeiro.

```
typedef struct {
  long opcode;     /* == SYS_COPY */
  long source;     /* origin process */
  long dest;       /* destination process */
  long size;       /* size of transfer */
  long dst_seg;    /* destination segment */
  long dst_buf;    /* destination buffer */
  long asker;      /* who started the copy */
} copy_header;
```

Figura 6.11: Cabeçalho de uma cópia via *link*

6.2.2 Recepção dos dados de outro processador

A *link task* tem cinco fluxos de execução independentes (paralelos), sendo um para atender às mensagens do núcleo e da *system task* e outros quatro para atender aos *links*. O procedimento executado para escutar os *links*, chamado `link_handler`, está em geral bloqueado através da instrução `in` em um *link*.

Quando alguma informação chega pelo *link*, originada como foi descrito na seção anterior, verifica-se se é uma mensagem local, no caso em que é entregue ao núcleo. Se não for uma mensagem local, pode ser uma cópia para

um processo local, caso em que os dados são recebidos diretamente no espaço de endereçamento do processo destinatário. Se for uma cópia ou uma mensagem para um processo não local, os bytes que seguem são recebidos e repassados através de outro *link*.

Após receber uma cópia de memória para um processo local, a *link task* precisa ainda responder ao processo que a ordenou, para indicar que a mesma foi levada a cabo. Como as cópias de memória são pedidas para a *system task* e não para a *link task*, o processo ordenador está bloquado em RECEIVE da *system task* do seu processador. A *link task* então finge ser a *system task* da mesma CPU do processo que ordenou a cópia e envia uma mensagem a este indicando o fim da mesma. O mecanismo da *system task* nesse procedimento de cópia está detalhado na seção 6.3.2.

Para decidir qual *link* usar para enviar uma mensagem ou cópia a um dado processo remoto, é chamada a função `map_host_to_link`, com o número do processador que se deseja atingir como parâmetro. Esta função mapeia todos os processadores do sistema em um dos *links* existentes no processador local. Atualmente, como a configuração da ligação dos transputers é estática, esta função é avaliada através de uma tabela, gerada na carga do sistema. Em um sistema em que as ligações possam ser reconfiguráveis, como em um T-NODE [TEL91], seria necessário um mecanismo mais dinâmico. Esta tabela pode ser ampliada ou ter seus valores alterados, de maneira que se possa implementar um roteamento mais inteligente, que por razões de falha ou melhor desempenho, seja capaz de alterar dinamicamente as rotas.

6.3 Alterações da *System Task*

Os processos no sistema MINIX, e por conseguinte no TRIX, são controlados pelo servidor *Memory Manager*. Ocorre que este servidor não tem acesso às

tabelas do núcleo e nem controle sobre o escalonador de processos. Existe então uma *task* especializada em controlar o sistema, chamada *system task*, que aloca e libera segmentos de memória, coloca e retira processos da fila do escalonador e faz cópias de memória entre os processos. Como a semântica da maioria destas operações no TRIX é diferente do MINIX, algumas mudanças tiveram que ser feitas nesta *task*.

Esta seção não apresenta detalhadamente todas as alterações da *system task* porque para isso haveria que se mostrar muitas informações pouco significantes. Serão somente descritas nas seções seguintes as novas funções da *system task*, já que representam as maiores mudanças em seu código. São elas:

- `do_rfork`, acionada pela mensagem `SYS_RFORK`, enviada pelo *Memory Manager* ao criar um processo de origem remota.
- `do_gettab`, acionada pela mensagem `SYS_GETTAB`, enviada por uma *system task* que recebeu um `fork` remoto.
- `do_rexit`, acionada pela mensagem `SYS_REXIT`, enviada pelo *Memory Manager* por ocasião da morte de um filho remoto.
- `do_copy`, não é uma função nova, mas passou a suportar pedidos de cópia interprocessador.
- `do_synchr`, acionada pela mensagem `SYS_SYNCHR` em momentos em que é necessário parar a execução do escalonador do *hardware*.
- `do_shadow`, acionada por uma mensagem da *clock task*, para que sejam trocados os processos em *shadowing*.

6.3.1 Criação de um Processo de Origem Remota

O *Memory Manager* ao aceitar um pedido remoto de criação de um processo, antes de fazer a cópia de seus segmentos de memória, comunica à *system task* a sua criação, enviando-lhe uma mensagem SYS_RFORK. Ao receber este tipo de mensagem, a *system task* local envia uma mensagem SYS_GETTAB à *system task* do processador que pediu a execução remota. A operação feita por ela ao receber a mensagem SYS_GETTAB é o envio da entrada na tabela de processos do núcleo que contém o processo a duplicar.

Recebida a resposta da mensagem SYS_GETTAB, a *system task* insere uma cópia da entrada remota na tabela local. Feito isto, ajusta alguns valores desta entrada, como PID e endereço global e responde ao *Memory Manager*, indicando que o processo foi criado localmente.

Da mesma forma, quando um processo filho remoto morre, o *Memory Manager* remoto avisa o *Memory Manager* local da ocorrência para que este atualize suas tabelas (ver seção 6.5.2). O *Memory Manager* local envia então uma mensagem do tipo SYS_REXIT à *system task* contendo os tempos de sistema e de usuário do processo morto, para serem somados aos tempos de seu pai.

6.3.2 Cópia de Memória Interprocessador

Além do mecanismo de mensagens entre processos, os servidores e as *tasks* fazem uso de cópias de memória para trocar dados entre os processos do sistema. A *system task* originalmente tem uma função SYS_COPY que faz a cópia de trechos de memória de um processo para outro. No TRIX, caso a cópia de memória for entre dois processos do mesmo processador, a própria *system task* faz a cópia.

Se para fazer a cópia for necessário transferir dados entre processadores diferentes, a *system task* simplesmente repassa o pedido de cópia para a *link task* do processador que contém o processo **a partir de onde** se vai copiar a memória. Esta *system task*, após repassar o pedido, não responde ao seu cliente. A *link task* se encarrega totalmente da cópia, inclusive respondendo ao final da cópia para o processo que a ordenou, como descrito na seção 6.2.2. Está representada na figura 6.12 uma cópia remota de memória, com as mensagens trocadas entre os processos envolvidos e as transferências de memória feitas, todas numeradas em seqüência.

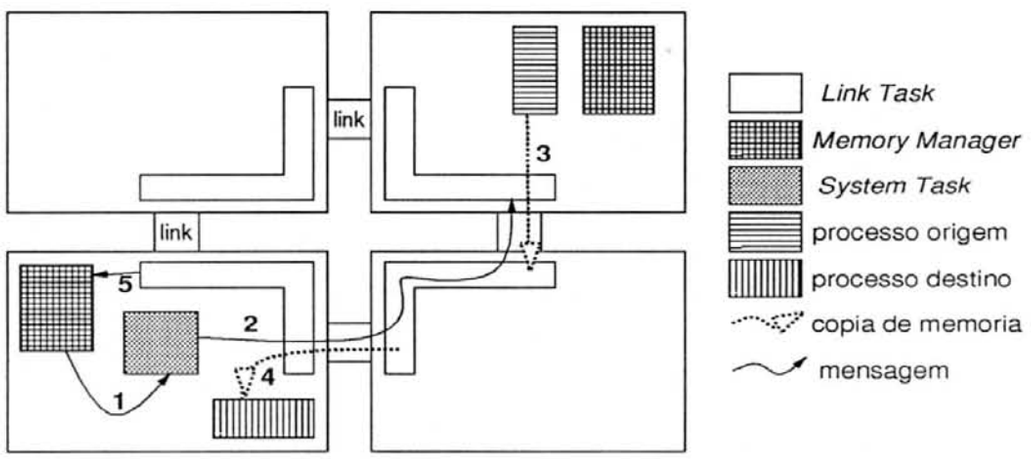

Figura 6.12: Cópia de memória interprocessador

6.3.3 Alteração da fila de processos do *hardware*

O escalonador do transputer automaticamente distribui fatias de tempo da CPU entre *threads* não-urgentes — normalmente processos de usuários. Para inserir uma *thread* no conjunto de *threads* sendo escalonadas pelo *hardware*, usa-se a instrução `runp` conforme descrito na seção 3.2. Para retirar uma *thread* deste conjunto, ela espontaneamente precisa executar uma instrução `in`, `out` ou `stopp`. Não existem instruções para controle externo, onde o núcleo de um sistema operacional possa forçar a parada de uma *thread*.

Assim, foi necessário dotar uma das *tasks* do TRIX de funções que pudessem parar um processo de usuário, como em casos de sinais ou troca de *shadowing* (ver seção 5.5.2). A *system task* foi escolhida por já possuir mecanismos correlatos. A função `do_synchr` foi implementada conforme a figura 6.13. A função `restart_thread` executa diretamente a instrução `runp` e `halt_thread` executa a instrução `stopp` (ver seções 3.2 e 6.1.4). A função `start_thread` dispara uma nova *thread* usando a instrução `runp`, após alocar um espaço de memória para ser usado como *workspace*.

```
PDes synchr_desc;

PRIVATE void synchr_thread() {
  PDes useless;
  restart_thread(synchr_desc);
here:
  halt_thread(&useless);
  goto here;
}

PRIVATE int do_synchr() {
/* Make sure there is no process in RegSaveArea */
  int scratch;

  start_thread(synchr_thread, synchr_stack,
               NOTURGENT, SYNCHR_STACK, 0);
  halt_thread(&synchr_desc);
  return(OK);
}
```

Figura 6.13: Funções usadas antes de alterar a fila do escalonador

6.4 *Clock Task*

Completando o conjuto de *tasks* portadas para o TRIX, aqui está apresentada a *clock task*. Como no MINIX, os processos programam alarmes através de

mensagens à *clock task*. Foram adiconadas à ela as seguintes funções, cada uma executada por uma *thread* separada:

- `shadow_handler`, envia uma mensagem à *system task* em intervalos predeterminados, para que sejam trocados os processos em *shadowing*.
- `balance_handler`, periodicamente calcula a carga do processador local e envia um mensagem ao *Memory Manager*.

A implementação destas funções está mostrada na figuras 6.14 e 6.15.

```
PRIVATE void shadow_handler() {
  for (;;) {
    wait_timer(SHADOW_STEP);
    interrupt(SYSTASK, 0);
  }
}
```

Figura 6.14: Função que ordena a troca de *shadowing*

6.5 Novas Funções do *Memory Manager*

O controle dos processos do sistema é feito pelo servidor *Memory Manager*. Como o TRIX é um sistema distribuído, este servidor está replicado em todos os processadores, cada um deles controlando seus processos de maneira autônoma. Quando for necessário criar um novo processo, o sistema como um todo deve decidir **onde** executar este processo. O *Memory Manager* do processador que estiver criando um novo processo deve então iniciar no sistema um procedimento de escolha. Esta escolha poderia ser de diversas maneiras, como foi discutido na seção 2.3.

Para projetar os artifícios de controle distribuído do sistema foram levados em conta os seguintes critérios:

```
PRIVATE void balance_handler() {
  int last_idle, this_idle, load, nload;
  message m;

  last_idle = proc_addr(IDLE)->user_time;
  m.m_source = message_dest(kernel_parm.this_processor, CLOCK);
  m.m_dest = MM_PROC_NR;
  m.m_size = 20;
  m.m_type = LLOAD;
  load = 0;

  for (;;) {
    wait_timer(BALANCE_STEP);
    this_idle = proc_addr(IDLE)->user_time;
    load += 100 - ((this_idle - last_idle) * 100) /
                (BALANCE_STEP/CLOCK_DIV);
    if (load<0) m.m1_i1 = 0;
    else {
      m.m1_i1 = load;
      load = 0;
    }
    last_idle = this_idle;
    hsend(MM_PROC_NR, &m, 20);
  }
}
```

Figura 6.15: Função que informa a carga do processador

- não possuir controle centralizado,
- poder ser acionado por qualquer algoritmo para equilibrar carga,
- usar o mínimo de mensagens possível.

Como os transputers utilizados não possuiam relocação de processos em execução (ver seção 5.5.3), nesta versão do TRIX não foi levada em consideração a possibilidade da migração de processos. Somente é avaliada a condição do sistema ao se disparar um novo processo, deixando-se o estudo da migração para trabalhos futuros. Com isso, no momento do disparo de cada novo processo são feitas duas operações: levantamento do processador mais adequado para criação do mesmo, seguido de seu efetivo disparo.

6.5.1 Levantamento do processador com menos carga

O presente trabalho não prevê extensos estudos neste assunto, mas para se poder validar o funcionamento do controle distribuído dos processos pelo sistema foi necessário implementar um pequeno algoritmo para equilibrar a carga.

Conforme descrito na seção 5.6.2, periodicamente a *clock task* de cada processador avalia a sua carga de processamento. A carga avaliada chega ao *Memory Manager* através de uma mensagem `LLOAD`. Os *Memory Managers* usam mensagens do tipo `RLOAD` para trocar informações sobre o processador com a menor carga.

A informação de processador com menor carga é armazenada em uma estrutura de dados `load_info` com os seguintes campos:

- `local`: carga do processador local
- `min`: menor carga conhecida
- `place`: local da menor carga conhecida
- `dist`: distância até a menor carga conhecida

Cada vez que o *Memory Manager* recebe mensagens `LLOAD` e `RLOAD` são chamadas as funções `do_lload` e `do_rload`, respectivamente, mostradas da figura 6.16 e 6.17. Estas duas funções alteram a informação de processador com menor carga, comunicando tal alteração aos processadores vizinhos através da função `propagate`, que aparece na figura 6.18.

```
PUBLIC int
do_lload()
{
/* clock task noticed the cpu load. Send to the neighbours.
 */
  int load;

  load = mm_in.m1_i1;
  load_info.local = load;

  if (load<load_info.min) {
    load_info.place = 0;
    load_info.min = load;
    load_info.dist = 0;
    propagate();
  }
  dont_reply = TRUE;
}


PUBLIC int
get_balance()
{
  return load_info.place;
}
```

Figura 6.16: Funções que atualizam e informam a carga local

6.5.2 Distribuição dos Processos

Para controlar a árvore distribuída de processos foi implementada uma série de funções, que são executadas sob a ordem de mensagens. Todo o controle da árvore de processos é feito como se cada processo estivesse em um processador diferente. Para referenciar processos em processadores remotos são enviadas mensagens aos seus *Memory Managers*. Se acontecer de estes processos serem locais, em vez de serem enviadas mensagens, uma função é chamada diretamente.

A mensagem `RFORK` é originada em uma chamada `fork`. É enviada se o mecanismo de equilíbrio de carga decidir que o processador mais apto a executar este processo é um outro processador que não o local. O processador

escolhido vai avaliar o pedido e responder com uma de duas respostas: RYFORK ou RNFORK. A figura 6.19 mostra o código que trata estas mensagens, sem o código original do MINIX.

Quando um *Memory Manager* conseguir criar um novo processo que foi pedido remotamente, a mensagem RYFORK é enviada ao *Memory Manager* que fez o pedido. A função que trata este tipo de mensagem está mostrada na figura 6.20. O resultado (positivo ou negativo) é enviado a um outro servidor em vez de um processo de usuário. O PID do processo filho é gerado de maneira unívoca no sistema. Se fosse mantido o mesmo esquema do MINIX, seria comum existir mais de um processo com o mesmo PID em processadores distintos.

Quando a mensagem RNFORK é recebida significa que seu pedido de execução não pôde ser atendido. O *Memory Manager* que falhou em atender o pedido é quem envia esta mensagem. O processo é então executado localmente, conforme mostra a figura 6.21.

O *Memory Manager* de um processo que terminou e tem pai remoto envia a mensagem REXIT ao *Memory Manager* do processo pai com o código de retorno, passado na função `exit()`. Se o processo pai não estiver bloqueado em `wait()`, esta mensagem é ignorada pelo *Memory Manager* do pai. A função que trata deste tipo de mensagem está na figura 6.22.

Quando um processo que tem filhos executar a chamada `wait` (função `do_wait()`, figura 6.23), para esperar códigos de retorno de filhos que terminarem, o *Memory Manager* local envia a mensagem RWAIT ao *Memory Manager* do primeiro filho. Esta mensagem é repassada por todos os *Memory Managers* para o próximo irmão, até chegar a um processo que não tenha mais irmãos. O *Memory Manager* que receber um RWAIT e estiver com algum processo já morto (mesmo já tendo enviado REXIT), envia um REXIT ao *Memory Manager* do pai. Estes procedimentos estão implementados na função `do_rwait()` da figura 6.24.

Se um processo que possui filhos terminar, o `init` passa a ser o novo pai de seus filhos. Uma mensagem `ORPHAN` é inicialmente enviada pelo *Memory Manager* que coordena um processo que tenha filhos, para o *Memory Manager* do primeiro filho do processo. Os *Memory Managers* dos filhos repassam sucessivamente a mensagem aos *Memory Managers* dos irmãos dos processos, até que não haja mais filhos do processo morto (figura 6.25).

Quando um processo com filhos morre, os seus filhos são assumidos pelo processo `init`. O *Memory Manager* do processo pai morto envia uma mensagem `INHERIT` ao *Memory Manager* do processo `init` contendo o endereço global do primeiro filho do processo morto. O *Memory Manager* do `init`, se possuir filhos, repassa a mensagem adiante, até esta chegue ao último filho, que recebe como irmão tal processo (e conseqüentemente, todos os seus irmãos).

Como o *Memory Manager* do pai pode descartar algumas mensagens do tipo `REXIT`, nem sempre se podem apagar as informações sobre um processo logo após sua morte. O mesmo permanece na tabela de processos, como um *zumbi*, até que o *Memory Manager* do pai envie um `FRESLT`, avisando que recebeu e interpretou corretamete o seu `REXIT`. A função executada ao receber uma mensagem `FRESLT` está mostrada na figura 6.27.

6.6 Desempenho do Sistema

Não existem muitas formas de se avaliar o desempenho do sistema, pois os laboratórios usados não possuiam nenhum sistema operacional para transputers — isto em parte foi o que motivo este projeto. Porém, pode-se avaliar o desempenho do sistema de forma preliminar, fazendo um programa de *benchmark* executar através um ambiente de programação qualquer e através do TRIX, comparando-se os resultados.

O ambiente de programação utilizado [LOG90] para confecção do TRIX não provê mecanismos de execução de novos processos e não executa os servidores do sistema nos transputers. Isso obrigatoriamente faz com que um programa de *benchmark* que não use as vantagens adicionais do TRIX, como a criação dinâmica de processos, corra mais devagar sob ele que no ambiente original.

As vantagens do TRIX sobre o ambiente de programação original são a criação e distribuição dinâmica de processos e acesso a dispositivos seguindo semântica idêntica ao UNIX, iclusive com *caches* nas operações de entrada e saída (o ambiente de programação original não possui as funções `fork()` ou `exec()`, por exemplo). Com isso, os motivos principais para o TRIX seja mais lento são:

- sobrecarga por haver vários processos em execução
- atendimento às interrupções temporizadas de *shadowing* e avaliação da carga de processamento
- cópias de memória devidos ao *shadowing*
- inclusão de cabeçalhos adicionais nas mensagens devido ao mecanismo de comunicação mais sofisticado

Tabela 6.1: Desempenho comparativo do TRIX

sistema	*dhrystones*/segundo
ambiente	3125
TRIX	2380

Foi executado o programa `dhrystone`, que é escrito em C e é usado comumente para medir o desempenho conjunto de compilador, sistema operacional e processador. Como o compilador e o processador usados nos dois casos foram os mesmos, pode-se garantir que a diferença do desempenho está no sistema operacional. A tabela 6.1 representa a comparação dos resultados da execução do processo no ambiente de programação e no sistema TRIX. Através dela se conclui que em aplicações com um único processo, que não use entrada nem saída, o desempenho do TRIX é 24 % inferior ao do ambiente.

```
PUBLIC int
do_rload()
{
 /* Receive information about the load in the neighbours.
  */
  int place, load, dist;
  message m;

  load = mm_in.m1_i1;
  place = mm_in.m1_i2;
  dist = mm_in.m1_i3 + 1;     /* this CPU is 1 hop farther */
  dont_reply = TRUE;

  /* check if a lower load or distance info came */
  if (load < load_info.min ||
     (load == load_info.min && dist < load_info.dist)  ) {
    load_info.min = load;
    load_info.place = place;
    load_info.dist = dist;
    propagate();
    return;
  }

  /* The load isn't lower. Check if local is lower */
  if (load > load_info.local) {
    load_info.min = load_info.local;
    load_info.place = 0;
    load_info.dist = 0;
    propagate();
    return;
  }
 /* No lower load, local isn't lower too.
  * Check if the load of the current lower has changed.
  */
  if (place == load_info.place) {
    load_info.min = load;
    load_info.dist = dist;
    propagate();
    return;
  }
 /* No lower load, no local lower, no change in the current
  * lower. It\'{}s not a lower load at all. Forget about it.
  */
}
```

Figura 6.17: Função que processa a informação sobre carga remota

```
PRIVATE int
propagate()
{
/* propagate the lowest known load to the neighbours
 */
  int n;
  message m;

  m.m_type = RLOAD;
  m.m1_i1 = load_info.min;
  if (load_info.place) m.m1_i2 = load_info.place;
  else  m.m1_i2 = this_processor;
  m.m1_i3 = load_info.dist;
  for (n=0; n<MAX_NEIGHBOURS; n++)
    if (neighbour[n])
      vsend(neighbour[n], &m, 28);
}
```

Figura 6.18: Função que informa os vizinhos a menor carga conhecida

```
PUBLIC int
do_rfork()
{
  if (não há memória disponível do processador local ) {
    i = EAGAIN;
    tell_mm(parent, RNFORK, 1, i);
    dont_reply = TRUE;
    return (OK);
  }
```

cria o processo localmente, como no MINIX

```
  /* in Minix: 1 < PID < 30000 */
  /* evaluate a global PID */
  rmc->mp_pid |= this_processor << 16;

  /* Send RYFORK to parent MM */
  tell_mm(parent, RYFORK, 2, ac_child, rmc->mp_pid);

  who = child_nr;
  return(OK);
}
```

Figura 6.19: Funções chamadas com a mensagem RFORK

```
PUBLIC int
do_ryfork()
{
/* A process has been forked and sent to another processor.
 * That processor started the new process and sent its
 * identification.
 */
  int parent, ac_child, cpid;

  parent    = mm_in.m1_i1;
  ac_child  = mm_in.m1_i2;
  cpid      = mm_in.m1_i3;
  mproc[dest_proc_number(parent)].mp_child = ac_child;
  who = parent;
  return (cpid);
}
```

Figura 6.20: Funções chamadas com a mensagem RYFORK

```
PUBLIC int
do_rnfork()
{
/* A process has forked and sent to another processor. That
 * processor unfortunately cannot fork so it sent it back.
 */

  /* set load_info to run locally */
  load_info.min = load_info.local;
  load_info.place = 0;
  load_info.dist = 0;

  /* setup variables to do_fork() */
  ac_who = mm_in.m1_i1;
  who = dest_proc_number(ac_who);
  mp = &mproc[who];
  do_fork();
  dont_reply = TRUE;
  return(OK);
}
```

Figura 6.21: Funções chamadas com a mensagem RNFORK

```
PUBLIC int
do_rexit()
{
/* A REXIT message has arrived: execute mm_rexit.*/
  mm_rexit(mm_in.m1_i1, mm_in.m1_i2, mm_in.m1_i3,
           (int)mm_in.m1_p1,  (int)mm_in.m1_p2);
  dont_reply = TRUE;
  return (OK);
}

PRIVATE void
mm_rexit(int who, int child, int cldpid, int retc, int brother)
{
  register struct mproc *parent;

  parent = &mproc[who=dest_proc_number(who)];
  if (!(parent->mp_flags & WAITING)) return; /* just forget */

  /* Wakeup the parent. */
  parent->mp_flags &= ~WAITING;
  reply(who, cldpid, retc, NIL_PTR);

  /* release child slot */
  if (dest_cpu(parent->mp_child) != this_processor)
    tell_mm(parent->mp_child, FRESLT, 2, child, brother);
  else
    mm_freslt(parent->mp_child, child, brother);

  /* check to see if it was the first child */
  if (parent->mp_child == child) parent->mp_child = brother;
}
```

Figura 6.22: Funções chamadas com a mensagem `REXIT`

```
PUBLIC int
do_wait()
{
/* A process wants to wait for a child to terminate.
 * If one is already waiting, go clean it up and let this
 * WAIT call terminate. Otherwise, really wait.
 */

  register struct mproc *rp;
  register int child;

  /* Is there a child waiting to be collected? */
  if (!mp->mp_child) return (ECHILD);

  mp->mp_flags |= WAITING;

  /* if there is any child, do RWAIT */
  if (dest_cpu(mp->mp_child) != this_processor)
       tell_mm(mp->mp_child, RWAIT, 0);
  else mm_rwait(mp->mp_child);

  dont_reply = TRUE;
  return(OK);
}
```

Figura 6.23: Função da chamada `WAIT`

```
PUBLIC int
do_rwait()
{
/* a message RWAIT arrived: execute mm_rwait */
mm_rwait(mm_in.m1_i1);
dont_reply = TRUE;
return (OK);
}

PRIVATE void
mm_rwait(int who)
{
/* RWAIT: the parent process isued a wait. Look to see
 * if the process is hanging. If not, pass the RWAIT
 * to the next brother.
 */
  struct mproc *rmp;
  int next, parent, r;

  while (who && dest_cpu(who) == this_processor) {
    rmp = &mproc[dest_proc_number(who)];
    if (rmp->mp_flags & HANGING) {
      parent = rmp->mp_parent;
      /* evaluate exit() code */
      r = rmp->mp_sigstatus & 0377;
      r = r | (rmp->mp_exitstatus << 8);
      if (dest_cpu(parent) != this_processor)
           tell_mm(parent, REXIT, 4, who, rmp->mp_pid,
                     r, rmp->mp_brother);
      else mm_rexit(parent, who, rmp->mp_pid,
                     r, rmp->mp_brother);
return;
    }
    who = rmp->mp_brother;
  }
}
```

Figura 6.24: Funções chamadas com a mensagem RWAIT

```
PUBLIC int
do_orphan()
{
/* a message ORPHAN arrived: execute mm_orphan */
  mm_orphan(mm_in.m1_i1);
  dont_reply = TRUE;
  return (OK);
}

PRIVATE void
mm_orphan(int who)
{
/* ORPHAN: the parent process has exited. New parent is init. */
  struct mproc *rmp;

  while (who && dest_cpu(who) == this_processor) {
    rmp = &mproc[dest_proc_number(who)];
    rmp->mp_parent = init_proc_nr;
    who = rmp->mp_brother;
  }
  if (who) tell_mm(who, ORPHAN, 0);
}
```

Figura 6.25: Funções chamadas com a mensagem ORPHAN

```
PRIVATE void
mm_inherit(int who, int head)
{
/* INHERIT: init has a bunch of new sons. Notice last son
 * of init to add the head of the new list to the tail of
 * the old list.
 */
  struct mproc *rmp;
  int next;

  rmp = &mproc[dest_proc_number(who)];
  /* The first element of the list is the init child.
   * After the first, the element is a brother.
   */
  if (who == init_proc_nr) next = rmp->mp_child;
  else next = rmp->mp_brother;

  /* while there are elements on the list */
  while (next) {
    if (dest_cpu(next) != this_processor) {
      tell_mm(next, INHERIT, 1, head);
      return;
    }
    who = next;
    rmp = &mproc[dest_proc_number(who)];
    next = rmp->mp_brother;
  }

  /* set the end of the list to be = head */
  if (who == init_proc_nr) rmp->mp_child = head;
  else rmp->mp_brother = head;
}
```

Figura 6.26: Funções chamadas com a mensagem `INHERIT`

```
PUBLIC int
do_freslt()
{
/* a message FRESLT arrived: execute mm_freslt */
  mm_freslt(mm_in.m1_i1, mm_in.m1_i2, mm_in.m1_i3);
  dont_reply = TRUE;
  return (OK);
}

PRIVATE void
mm_freslt(int who, int dead, int deadbro)
{
/* FRESLT: the death of a process has been already waited
 * by its parent, so MM of this process can release it
 * slot. FRESLT is sent through all the childs until it
 * reaches the destination.
 */
  struct mproc *rmp;
  int next;

  rmp = &mproc[dest_proc_number(who)];
  while (dest_cpu(who) == this_processor) {
    if (who == dead) {   /* reached the last mm */
      /* Update flags. */
      rmp->mp_flags = 0; /* release the table slot */
      procs_in_use--;
      return;
    }
    next = rmp->mp_brother;
    if (next == dead) rmp->mp_brother = deadbro;
    who = next;
  }
  tell_mm(who, FRESLT, 2, dead, deadbro);
}
```

Figura 6.27: Funções chamadas com a mensagem FRESLT

7 CONCLUSÃO

Este trabalho mostrou como foi feito o sistema operacional TRIX, como parte dos trabalhos do grupo de processamento paralelo do Instituto de Informática da UFRGS. Foi inicialmente constatado que graças à arquitetura do transputer, este sistema apresenta ganhos em desempenho em relação à versão original do MINIX. Isto se deve à capacidade de o transputer escalonar processos e enviar mensagens através de instruções específicas para tal fim. Estas características também permitiram que a parte escrita em linguagem *assembly* do MINIX desaparecesse quase que por completo, o que facilitou sua alteração e depuração.

Ao desmembrar o núcleo original do MINIX, obtiveram-se diversos processos independentes, cada um controlando um tipo de dispositivo do sistema. Esta alteração permitiu que uma *task* fosse reprogramada sem muita preocupação com o que estava programado em outras, alem de tornar seu código mais fácil de ser portado para novos dispositivos.

O sistema apresentado é compatível com o UNIX versão 7, podendo fazer uso da vasta biblioteca de programas existente para tal sistema. Além disso, comparado com as soluções mais freqüentemente encontradas para o controle de transputers, ele tem a vantagem de ser executado pelos próprios transputers. Em outros sistemas, quem executa o sistema operacional é o hospedeiro. Isto permite encarar os transputers como sendo os processadores mais significativos no sistema, posição que deveriam ocupar por seu alto desempenho. O processador mais simples do hospedeiro por sua vez, fica desobrigado de executar as pesadas tarefas de um sistema operacional, utilizando o seu tempo de processamento para atender com maior velocidade às operações de entrada e saída.

Em todo o trabalho, os pontos fracos apresentados foram a necessidade de fazer uso de mecanismos como *shadowing* para o disparo de processos e o baixo desempenho em relação a um ambiente de programação sem sistema

operacional. O primeiro problema estará resolvido no momento em que se tornar disponível comercialmente o novo modelo de transputer, que possui dispositivos para relocação de código. Quanto ao segundo revés, ele é o preço que se paga para se ter acesso à semântica completa de processos do UNIX. Ainda assim, algumas otimizações podem ainda ser feitas no código do sistema, como um algoritmo mais preciso e mais leve de equilíbrio de carga entre os processadores. A futura retirada do *shadowing* do sistema também melhorará o seu desempenho, já que menos mensagens serão periodicamente enviadas e menos cópias de memória serão necessárias.

O TRIX pode ser executado em um grupo de transputers com qualquer topologia, pois o roteamento de mensagens entre processos é feito fora do núcleo por um processo especializado. Este processo pode ser reescrito para usar qualquer dispositivo de comunicação, com as mais diversas topologias. Pode-se também dotar de roteamento dinâmico, por razões de desempenho ou tolerância a falhas.

O núcleo provê aos processos do sistema uma transparência de localidade ao converter automaticamente os endereços das mensagens de locais em globais. Os processos se comunicam sempre com servidores em endereços fixos, não importando a posição dos mesmos. Como os servidores respondem aos endereços que estão nos pedidos, não há problema se os mesmos forem endereços globais.

A informação sobre a árvore de processos é totalmente distribuída e pode ser usada por qualquer que seja o mecanismo de regulação da carga utilizado. A semântica UNIX foi mantida, mesmo sendo necessário existir uma árvore de processos, onde o número de ramos de cada nó é desconhecido.

Efetua-se a distribuição da carga de processamento de maneira simples, sem nenhum controle centralizado. Usando-se processadores mais modernos, com capacidade de relocação de código em execução, pode-se estender o

mecanismo para suportar migração de processos. Pode-se até mesmo reescrever totalmente o mecanismo de distribuição da carga, sem que seja necessário mudar o controle da árvore de processos.

Tendo sido completada sua implementação, estudam-se agora as possibilidades de dar continuidade ao projeto. O roteamento dinâmico de mensagens pode ser facilmente implementado dentro do processo de controle do *link*, assim como a reconfiguração da conexão dos processadores. Um sistema de arquivos distribuído, com *caches* em cada processador parece ser um dos maiores desafios, já que implicam em reescrever o servidor. O sistema serve bem para implementar e testar novos algoritmos de distribuição da carga de processamento, já que possui os mecanismos de mais baixo nível em operação.

BIBLIOGRAFIA

[ARS87] ARSTY, Y. et al. **Interprocess Communication in Charlotte. IEEE Software**, New York, v.4, n.1, p 25-32, 1987.

[BAC87] BACH, M. **The Design of the Unix Operating System**. Englewood Cliffs: Prentice-Hall, 1987. 471p.

[BAR90] BARCELLOS, A.M.P. et al. **DIX — Projeto e Implementação de um Sistema Operacional Distribuído para uma Rede de Estações de Trabalho**. Porto Alegre: CPGCC da UFRGS, 1990. 77p. (Projeto de Pesquisa).

[BEL93] BELMONTE FILHO, V.R., **Gerência de Processos em Sistemas Distribuídos Tolerantes a Falhas**. Porto Alegre: CPGCC da UFRGS, 1993. 186p. (Dissertação de Mestrado).

[BLA86] BLACK, J.P. et al. **The Architecture of Unix United**. Newcastle: University of Newcastle-upon-Tyne, 1986. 27p. (Technical Report).

[BOK93] BOKHARI, S.H. A Network Flow Model for Load Balancing in Circuit-Switched Multicomputers. **IEEE Transactions on Parallel and Distributed Systems**, New York, v.4, n.6, p.37-49, 1993.

[CAR89] CARRERAS, M. **Projeto e Implementação da Arquitetura M3P - Minix Multimicroprocessador**. Porto Alegre: CPGCC da UFRGS, 1989. 160p. (Trabalho Individual).

[CHA90] CHANDRAS, R.J. Distributed Message Passing Operating Systems. **Operating Systems Review**, New York, v.24, n.1, p.7-17, 1990.

[CHE88] CHERITON, D.R. The V Distributed System. **Communications of the ACM**, New York, v.31, n.3, p.314-333, 1988

[EAG86] EAGER, D.L. et al. Adaptive Load Sharing in Homogeneous Distributed Systems. **IEEE Transactions on SE**, New York, v.SE-12, p.662-675, 1986.

[FLY66] FLYNN, M.J. Very High Speed Computing Systems. **Proc.IEEE**, New York, v.54, p.1901-1909, 1966.

[GEI90] GEIHS, K.; HOLLBERG, U. Retrospective on DACNOS. **Communications of the ACM**, New York, v.33, n.4, p.439-448, 1990.

[HED88] HEDRICK, C. **Routing Information Protocol**. Rutgers: Rutgers University, 1988. 12p. (RFC 1058).

[HEL89] HELIOS Co. **Helios Technical Manual**. New York: Helios, 1989. 477p.

[HOA74] HOARE, C.A.R. Monitors, An Operating System Structuring Concept. **Communications of the ACM**, New York, v.17, n.10, p.549-557, 1974.

[INM87] INMOS Ltd. **Transputer Reference Manual**. Bristol: Inmos, 1987. 263p.

[INM88a] INMOS Ltd. **Transputer Databook**. Bath: Inmos, 1988. 461p.

[INM88b] INMOS Ltd. **OCCAM 2 Reference Manual.** Cambridge: Prentice-Hall, 1988. 328p.

[LEI91] LEISS, E.L.; REDDY, H.N. **Distributed Load Balancing algorithms: design and performance analysis**. Houston: W.M.Keck Research Computation Laboratory, University of Houston, 1991. v.5, p.205-269. (Annual Progress Review).

[LIN87] LIN, F.C.H.; KELLER, R.M. The Gradient Model Load Balancing Method. **IEEE Transactions on SE**, New York, v.SE-13, n.1, p.32-38, 1987.

[LOG90] LOGICAL SYSTEMS Ltd. **Logical Systems C for the Transputer.** Chicago: Logical Systems, 1990. 550p.

[MAS87] MASON, W.A. Distributed Processing: The State of the Art. **BYTE**. New York, McGraw-Hill, Nov. 1984.

[MCC90] MC CULLAGH, P.; SMIT G.DE V. Implementing UNIX in the INMOS Transputer. **South African Computer Journal**, Cape Town, South African Computer Society, n.2, 1990.

[MUL88] MULLENDER, S.J. Process Management in a Distributed Operating System. In: INTERNATIONAL WORKSHOP ON EXPERIENCES WITH DISTRIBUTED SYSTEMS, Sept. 28-30, 1987, Kaiserslautern. **Proceedings...** Berlin: Springer Verlag, 1988. p.38-51. (Lecture Notes in Computer Science, 309).

[MUL90] MULLENDER, S.J; van ROSSUM, G. Amoeba: A Distributed Operating System for the 1990s. **IEEE Computer**, Los Alamitos, v.23, n.5, p.44-53, 1990.

[NAV90] NAVAUX, P.O.A. et al. M3P - Máquina e Sistema Operacional Multiprocessadores. In: CONGRESSO NACIONAL DE INFORMATICA, 12. **Anais...**, Rio de Janeiro: SUCESU, 1990. p.455-472.

[OUS88] OUSTERHOUT, J.K. et al. The Sprite Network Operating System. **IEEE Computer**, Los Alamitos, v.21, n.2, p.25-35, 1988.

[PAS91] PASIN, M. **Especificação do TRIX: Um sistema Operacional Multiprocessado para Transputers**. Porto Alegre: CPGCC da UFRGS, 1991. 73p. (Trabalho Individual).

[POP81] POPEK, G. et al. LOCUS, A Network Transparent, High Reliability Distributed System. In: SYMPOSIUM ON OPERATING SYSTEM PRINCIPLES, 8. **Anais...** New York: ACM, 1981. Operating Systems Review, v.15, n.5, p.169-177.

[POU91a] POUNTAIN, D. The Chorus of Approval. **BYTE**, New York, McGraw-Hill, v.16, n.4, p.265-275, 1991.

[POU91b] POUNTAIN, D. The Transputer Strikes Back. **BYTE**, New York, McGraw-Hill, v.16, n.8, p.185-192, 1991.

[RAS86] RASHID, R. et al. Mach: A New Kernel Foundation for Unix Development, In: USENIX CONFERENCE, Summer 1986. **Proceedings...** San Francisco: Usenix, 1986. p.93-112.

[REK89] REKHTER, Y. **EGP and Policy Based Routing in the New NSFNET Backbone**. Armonk: IBM Research, 1989. 17p. (RFC 1092).

[SCH92] SCHEER, R.L. et al. M3P: Um Multiprocessador Fracamente Acoplado com Balanceamento de Carga e Migração de Processos. In: Simpósio Brasileiro de Arquitetura de Computadores, 4. **Anais...** São Paulo, SBC, 1992. p.69-81.

[SHE87] SHEPHERD, D. **Transputer Compiler Writer's Guide**. Bath: Inmos, 1987. 188p.

[SHI92] SHIVARATRI, N.G. et al. Load Distributing for Locally Distributing Systems. **IEEE Computer**, Los Alamitos, v.25, n.12, 1992.

[SMI89] SMIT, G.de V. et al. **DistriX, A Multiprocessor UNIX Workbench**. Cape Town: Cape Town Iniversity, 1989. 22p. (Research Report).

[STA84] STANKOVIC, J.A. A Perspective on Distributed Computer Systems. **IEEE Transactions on Computer Systems**, New York, v.c-33, n.12, 1984.

[STE89] STEIN, B.O.S. **Desenvolvimento do Módulo Ouvidor da Máquina M3P—Software**. Porto Alegre: CPGCC da UFRGS, 1989. (Trabalho Individual, 132).

[STE92] STEIN, B.O.S. **Projeto do Núcleo de um Sistema Operacional Distribuído**. Porto Alegre: CPGCC da UFRGS, 1992. 98p. (Dissertação de Mestrado).

[TAN85] TANENBAUM, A.S.; van RENESSE, R. Distributed Operating Systems. **Computing Surveys**, New York, v.17, n.4, p.419-470, 1985.

[TAN87] TANENBAUM, A.S. **Operating Systems: Design and Implementation**. Englewood Cliffs: Prentice-Hall, 1987. 717p.

[TAN90] TANENBAUM, A.S. et al. Experiences with the Amoeba Distributed Operating System. **Communications of the ACM**, New York, v.33, n.12, p.46-63, 1990.

[TAN92] TANENBAUM, A.S. **Modern Operating Systems**. Englewood Cliffs: Prentice-Hall, 754p. 1992.

[TEL91] TELMAT INFORMATIQUE. **T-node hardware manual.** Soultz: Telmat Informatique, 1991. v.1. 233p.

[WAL83] WALKER, B. et al. The LOCUS Distributed Operating System. In: SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 9. **Anais...** New York: ACM, 1983. Operating Systems Review, v.17, n.5, p.49-70.

Informática
UFRGS

TRIX, Um Sistema Operacional Multiprocessado para Transputers, com Gerência Distribuída de Processos.

Dissertação apresentada aos Senhores:

Prof. Dr. Celso Maciel da Costa

Prof. Dr. Newton Faller (COPPE-UFRJ)

Prof. Dr. Philippe O. A. Navaux

Prof. Dr. Wolfgang Pandikow

Vista e permitida a impressão.
Porto Alegre, 15/06/94.

Prof. Dr. Philippe O. A. Navaux,
Orientador.

Prof. Dr. Ricardo A. da L. Reis,
Coordenador do Curso de Pós-Graduação
em Ciência da Computação.