DCC – IMECC – UNICAMP

Núcleo Multiprocessador para Aplicações de Tempo-Real

Roberto André Hexsel

Agosto de 1988

Orientador: Prof. Dr. Rogério Drummond

Dissertação apresentada como requisito parcial para obtenção do Título de Mestre em Ciência da Computação pelo Instituto de Matemática Estatística e Ciência da Computação da Universidade Estadual de Campinas.

Dedicação

Este trabalho é dedicado a Bila e Conrado, meus pais.

Agradecimentos

Gostaria de externar a minha gratidão àquelas pessoas que de alguma forma influiram na realização deste trabalho:

A Luiz Gimeno y Latre e Mário Bento de Carvalho, Diretores do Instituto de Automação e a Jaime Szajner e Rubens Campos Machado, Chefes do Departamento de Controle de Processos do IA, pela possibilidade de participar do projeto que deu origem a este trabalho,

A Josué G. Ramos Jr. e Morio Hirama pela colaboração nos projetos do controlador de robô e do controlador da célula de manufatura, respectivamente,

A Adylson B. Lopes a a Juan M. Adan Coello pelo auxílio na preparação do material do Apêndice A,

A Hélio Azevedo, pela revisão do texto e pelo seu auxílio no uso do sistema de desenvolvimento,

A Fábio Q. Bueno da Silva, por iniciar-me nos mistérios do TEX e LATEX; a Isaías M. C. Ribeiro pelo auxílio na programação do filtro `ws2tex`; a Wang Chen Yu e a Paulo M. Janousek pela preparação do texto para processamento pelo LATEX,

Ao pessoal do Departamento de Controle de Processos do IA: Adylson, Aqueo, Beiral, Cidinha, Clarinda (UDD), Edeneziano, Eliane, Hélio, Helinho, Heloísa (UDD), Isaías, Juan, Juziani, Koyama, Leonardo, Marcelo, Maria Ruth, Olga, Paulo, Roberto, Reinaldo e Rubens,

A Arthur João Catto e Joni da Silva Fraga pelas valiosas contribuições ao conteúdo do texto desta dissertação,

A Rogério Drummond pelo muito que aprendi com ele,

A Helena.

Resumo

Esta Dissertação descreve o núcleo de tempo-real do Multiprocessador para Sistemas de Controle (MSC). O MSC foi desenvolvido no Instituto de Automação do Centro Tecnológico para Informática e possui características que o tornam adequado a aplicações em controle de processos e automação industrial. Estas aplicações exigem respostas rápidas a eventos externos e grande capacidade de processamento. O MSC pode ser configurado para satisfazer as mais diversas aplicações e diferentes níveis de desempenho, desde controladores de robô até gerenciadores de sistemas integrados de manufatura.

O MSC é baseado no barramento VME e pode conter um ou mais processadores MC 68000 e zero ou mais processadores Z80. O barramento VME suporta a comunicação de baixo nível entre os programas executados em paralelo pelos processadores. Cada processador (MC 68000) executa o código (aplicativos e NÚCLEO) armazenado em sua memória local, à qual é conectado por um barramento privativo (padrão VMX). A sincronização e comunicação de alto nível entre os programas ocorre através da memória global, a que os processadores tem acesso pelo barramento VME. Os processadores de Entrada/Saída (PE/S) são escravos dos demais processadores. Os PE/S são interligados ao VME por memória dual-port: uma das portas é mapeada como memória global (no VME) e a outra é ligada ao barramento interno do PE/S.

O NÚCLEO do MSC é estruturado segundo o modelo estratificado. A unidade lógica de computação é o processo. O mecanismo de sincronização interprocessos é uma extensão de semáforos para uso no ambiente multiprocessado do MSC. O mecanismo de comunicação interprocessos é a troca de mensagens através de caixas postais. A localização (em memória global ou local) dos objetos suportados pelo NÚCLEO é especificada somente na sua criação. Todas as outras operações referenciam apenas aos identificadores dos objetos (números inteiros). Chamamos esta característica de *Transparência de Multiprocessamento*. Graças a ela, a programação do MSC, um multiprocessador, oferece um grau de dificuldade não maior que aquele apresentado por sistemas multitarefa convencionais. Além disso, se todos os objetos necessários à execução de um programa forem criados num mesmo processo, um aplicativo pode ser transportado para uma máquina com um número diferente de processadores sem alterações no seu código (exceto nos locais onde os objetos/recursos são criados).

O NÚCLEO oferece primitivas para a criação e destruição de processos (locais e remotos), operações `P` e `V` sobre semáforos, operações sobre conjuntos de armazenadores e troca de mensagens através de caixas postais. Os semáforos possibilitam soluções simples e eficientes para problemas de exclusão mútua e sincronização condicional. As caixas postais permitem a implementação de vários paradigmas de comunicação interprocessos, tais como comunicação síncrona ou assíncrona, *pipes* e *rendezvous*.

Abstract

This dissertation describes the real-time kernel for a Multiprocessor aimed at Control Systems Applications (MSC for short). The MSC multiprocessor was developed at Instituto de Automação, Centro Tecnológico para Informática. It has features that make it suitable for applications in industrial process control. These applications demand high throughput and prompt response to external events. The MSC can be configured to solve a wide variety of problems in control and automation, from robot controllers up to supervising chores in flexible manufacturing systems.

The MSC is built around a VME bus and may have one or more MC 68000 based processors and zero or more Z80 based I/O processors. The VME bus supports the low level communication between the processors. Each processor (MC 68000) executes the code (application and KERNEL) stored in its local memory, to which it is connected by a private bus (VMX bus). The high level synchronization and communication between the programs occurs through the global memory, which is accessed by the processors via the VME bus. The I/O processors interface to the VME bus is a dual ported memory: one port is connected to the global bus (mapped as global memory) and the other is connected to the I/O processor's internal bus.

The MSC's KERNEL was designed as a layered structure. The logical unit of computation is the process. The interprocess synchronization mecanism is an extension to semaphores so they can be used in the distribuited MSC's environment. The interprocess communication mecanism is message exchange through mailboxes. The loci (in either global or local memory) of the objects supported by the KERNEL is specified only at its creation. All the other operations upon them refer just to their identifiers (which are integers). We call this feature *Multiprocessing Transparency*. Thanks to it, the MSC's (a multiprocessor) programming presents no more difficulty than that of conventional multitasking systems. Furthermore, if all the objects needed to a program's execution are created on one process, this program may be ported to a machine with a different number of processors without any changes to its code (except, of course, to the places where the objects are created).

The KERNEL supports primitives for the creation and killing of processes (local and remote ones), `P` and `V` operations on semaphores, operations on buffer pools (get, release) and message exchange through mailboxes. The semaphores provide simple and efficient solutions to mutual exclusion and condition synchronization problems. The mailboxes allow the implementation of many interprocess communication paradigms, such as synchronous and assynchronous communication, pipes and rendezvous.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Esta dissertação apresenta os resultados do desenvolvimento de um núcleo de sistema operacional para aplicações de tempo-real. Este núcleo suporta a execução de aplicativos "distribuídos" no Multiprocessador para Sistemas de Controle (MSC). O MSC foi desenvolvido no Instituto de Automação do Centro Tecnológico para Informática, tendo em vista aplicações em controle industrial e automação da manufatura. Sua arquitetura é flexível e permite a construção de máquinas com diferentes níveis de desempenho, desde controladores de robô até gerenciadores de sistemas integrados de manufatura. Estas aplicações exigem respostas rápidas a eventos externos e grande capacidade de processamento; isto é obtido no MSC pela exploração de paralelismo real na execução de programas.

As aplicações de controle em tempo-real impõem restrições sérias quanto ao desempenho dos programas que implementam os algoritmos de controle e sobre o sistema operacional que suporta estes programas. Os recursos e funções oferecidos pelo sistema operacional devem ser eficientes e rápidos, ou seja, a sobrecarga incorrida no uso dos serviços oferecidos pelo sistema operacional deve ser a menor possível. Além disso, os eventos externos devem ser atendidos e tratados com presteza, de forma que sua ocorrência não seja eventualmente ignorada ou percebida de maneira errônea.

Dependendo do tipo de sistema, a carga computacional imposta pelos algoritmos de controle e o padrão de ocorrência dos eventos externos podem ser tais que um único processador não é capaz de manter o sistema controlado funcionando de uma forma determinista. Neste caso, a capacidade de processamento do controlador pode ser aumentada pela inclusão de um ou mais processadores. Esta adição de poder computacional implica na redistribuição das tarefas de controle entre os processadores, de tal forma que o comportamento do sistema global venha a ser sempre determinista.

O NÚCLEO de tempo-real do MSC suporta a execução concorrente de programas através de um conjunto de funções que permitem a cooperação entre estes programas. Estas funções escondem do programador que o código produzido por ele[1] será executado num mono- ou num multiprocessador. Dessa forma, a "distribuição" de um programa em um ou mais processadores pode ser alterada com facilidade e a custos mínimos. A sintaxe das funções oferecidas pelo NÚCLEO torna transparente a localização dos objetos suportados por ele, isto é, a programação do MSC oferece um grau de dificuldade não maior que aquele associado a sistemas multitarefa convencionais.

O NÚCLEO foi projetado dentro de uma filosofia "minimalista", de forma a que o programador tenha plena liberdade de escolher as políticas de gerenciamento de recursos e de

[1] Os termos "programador", "operador" e "ele", por exemplo, são elementos puramente sintáticos. Sua semântica deve ser inferida do contexto de quem os lê.

controlar a dinâmica de funcionamento dos programas. O mecanismo de sincronização interprocessos[2] suportado pelo NÚCLEO permite a solução dos problemas de exclusão mútua e sincronização condicional de uma maneira eficiente e elegante. O mecanismo de comunicação é simples e possibilita a implementação de vários paradigmas de comunicação interprocessos.

O restante deste texto está organizado da forma descrita a seguir. O Capítulo 2 apresenta alguns conceitos básicos e terminologia relacionados a sistemas de controle e automação industrial. A estruturação dos sistemas hierárquicos de controle, bem como os requisitos funcionais dos computadores dedicados a aplicações industriais são discutidos.

O Capítulo 3 discute o enquadramento dos multiprocessadores no espectro dos sistemas computacionais e discute três arquiteturas comumente empregadas na implementação de multiprocessadores. O Capítulo 4 apresenta a arquitetura do Multiprocessador para Sistemas de Controle e descreve os módulos e barramentos de que é composto. Este capítulo ainda contém dois exemplos de aplicação do MSC: um controlador de robô e um controlador de célula flexível de manufatura.

O Capítulo 5 discute a filosofia de projeto e a estruturação do NÚCLEO do MSC. A semântica dos mecanismos de escalonamento, sincronização e comunicação interprocessos é discutida em detalhe. O Capítulo 6 descreve a implementação do NÚCLEO, a organização das suas estruturas de dados e a sintaxe de todas as suas primitivas.

O Capítulo 7 apresenta conclusões e sugestões para o aprimoramento e sintonia fina do desempenho do NÚCLEO. O poder semântico das primitivas de sincronização e comunicação é discutido no Apêndice A, através de um exemplo (a implementação de uma "camada" de comunicação intermódulos). O Apêndice B traz as medidas de tempo de execução de todas as primitivas do NÚCLEO.

Ao longo do texto, os trechos de código e pseudocódigo estão escritos de acordo com as construções e convenções da linguagem de programação C [KERN86]. As palavras reservadas são grafadas em **negrito**, as não-reservadas em `máquina de escrever`, os comentários em *itálico* e as constantes em letras MAIÚSCULAS.

[2]Veja nos próximos parágrafos em quais capítulos este e outros termos estão definidos.

Capítulo 2

Sistemas de Controle Industrial

Neste capítulo, introduzimos alguns conceitos relacionados ao controle de processos industriais. A discussão concentra-se em sistemas de controle baseados em computadores digitais, com abrangência sobre toda uma fábrica ou usina. Tais sistemas vêm sendo empregados cada vez com mais freqüência na completa automatização de siderúrgicas, refinarias, na indústria automobilística e de manufatura. Em alguns casos, o montante do investimento empregado na instalação de um destes sistemas é recuperado em períodos menores que um ano [WILL87].

A função de um sistema de controle é manter os equipamentos controlados trabalhando dentro de determinadas condições de consumo de energia e insumos e de produção. Conceitualmente, a interação do sistema de controle com o conjunto de equipamentos controlados é a mesma tanto para processos contínuos quanto para processos discretos ou de manufatura. *Processos contínuos* são aqueles onde as matérias primas são processadas continuamente e envolvem, por exemplo, o controle de vazão, temperatura e pressão (tais como os processos químicos na indústria petroquímica). *Processos discretos* são aqueles associados à manipulação de partes semi-trabalhadas com a finalidade de se obter partes mais complexas (tais como os processos de manufatura na indústria automobilística).

2.1 Variáveis de Processo, "Set Points" e Laços de Controle

O estado de um processo industrial de fabricação é descrito pelos valores de grandezas físicas como temperatura, vazão e pressão (as *variáveis controladas*). Algumas destas variáveis podem ser medidas diretamente por sensores enquanto que outras são obtidas indiretamente pela combinação de valores obtidos diretamente. Os valores medidos pelos sensores (tensões ou correntes elétricas) devem ser digitalizados, condicionados e convertidos para unidades de engenharia (graus centígrados, metros cúbicos por segundo, etc...) para posterior uso e/ou armazenagem. Cada variável corresponde a um *ponto* de amostragem; cada ponto deve ser amostrado periodicamente a uma taxa que depende do tipo de grandeza física e da dinâmica do processo. Além de variáveis analógicas, sinais digitais como chaves fim de curso e sinalizadores de estado (*on/off*) também são pontos de amostragem. Em geral, as interfaces de aquisição de sinais digitais somente acusam a ocorrência de mudanças no estado dos sinais através de interrupções. Além de sensores, *atuadores* interagem com as grandezas físicas envolvidas no processo controlado. Válvulas de ação linear e resistências de aquecimento são exemplos de atuadores.

As taxas de amostragens dos pontos se situam entre uma e 100 vezes por segundo, dependendo do tipo da variável medida e da dinâmica do processo. O número de pontos

varia de uma dezena a alguns milhares, dependendo da complexidade do processo e da sofisticação do sistema de controle.

A figura 2.1 mostra um *laço de controle* e os seus componentes principais: um conjunto de *set points*, um conjunto de variáveis de processo, um mecanismo de regulação e o processo controlado. O controle de um processo consiste da geração de sinais de controle em função dos valores medidos e/ou calculados de algumas das variáveis para que os valores desejados para outras delas (os seus *set points*) sejam atingidos e mantidos. Os algoritmos de controle determinam a intensidade e freqüência dos sinais de comando para os atuadores em função dos valores desejados, dos valores medidos no passado próximo e de alguma estratégia que leve à correção dos erros detectados entre os valores medidos e os *set points* (regulação).

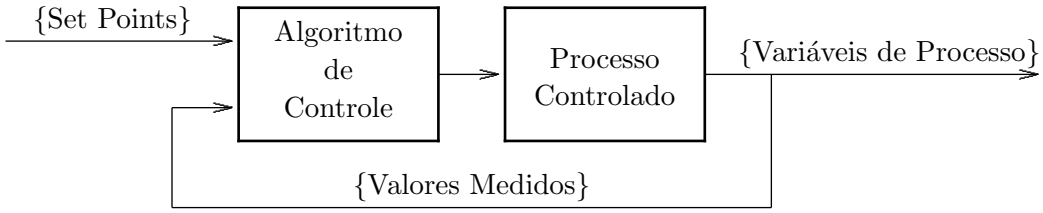

Figura 2.1: Laço de Controle

A interação entre os operadores e o sistema de controle ocorre através da interface de operação, consistindo de um quadro sinótico que descreve o estado de funcionamento da planta, e de um dispositivo de entrada que permita ao operador a alteração de parâmetros de operação do processo. O quadro sinótico é composto por um ou mais dispositivos de saída capazes de exibir o valor de variáveis selecionadas pelos operadores, condições de alarme e gráficos de tendência, por exemplo. Os valores das variáveis de processo e os parâmetros de operação ficam armazenados num banco de dados, para uso *on-line* pelo sistema de controle e para uso *off-line* pela administração e/ou engenharia de produção.

2.2 Componentes de um Sistema de Controle

Embora existam diferenças na operação e no controle de sistemas contínuos e discretos, um sistema projetado para controlar toda uma planta (uma fábrica ou usina) deve conter os seguintes componentes funcionais:

1. O controle estrito sobre cada unidade operacional da planta de tal forma que ela opere com máxima eficiência na produção e no uso de energia e matéria-prima. Os níveis de produção devem ser mantidos de acordo com o estabelecido pela *gerência de produção* (ítem abaixo). As situações de pane ou emergência que podem ocorrer na unidade operacional devem ser atendidas.

2. Um sistema de supervisão e coordenação que determine os níveis e otimiza a produção de todas as unidades operacionais sob sua área de atuação. Este subsistema coordena o uso de matéria-prima e/ou energia de todas as unidades da sua área, de tal modo que nenhuma delas exceda os níveis de demanda ou produção da área. Em caso de emergência ou pane em alguma das unidades, este subsistema é responsável, em conjunto com o controle local, pela redução da produção ou desligamento daquela e de outras unidades a ela relacionadas.

3. Um sistema de controle de produção capaz de transformar pedidos de clientes ou ordens da administração em comandos, de tal forma que os produtos desejados sejam produzidos com uma combinação ótima de tempo, energia, materiais e mão-de-obra de acordo com funções de custo adequadas.

4. Um método que assegure a confiabilidade e a disponibilidade de todo o sistema de controle, através de detecção de falhas, redundância e outras técnicas adequadas, incluídas na especificação e operação do sistema.

As três primeiras funções podem ser implementadas como três níveis de uma estrutura de controle hierárquico. Como o fluxo de informações entre os três níveis é muito grande, um sistema computacional distribuído, organizado segundo uma hierarquia de máquinas, é uma solução lógica para a implementação de um sistema de controle com abrangência sobre toda uma fábrica ou usina. Dependendo da complexidade da planta e do sistema de controle, as funções 1, 2 e 4 podem ser implementadas em microcomputadores ou supermicrocomputadores; a complexidade envolvida na função 3 exigiria um computador de médio ou grande porte.

2.3 Sistemas Hierárquicos – Implementação

Do início da década de 60 até meados da década de 70, os sistemas industriais de controle de produção eram construídos à volta de um computador que centralizava as funções de monitoração e controle regulatório de uma planta. Este computador era, em geral, de médio porte (para a época), caro e de operação e manutenção dispendiosas. Quando ocorria alguma pane no sistema de controle, as funções exercidas pelo computador eram momentaneamente transferidas para equipamentos analógicos ou operadores humanos, com freqüentes prejuízos para a produção.

Com a redução do custo de equipamentos digitais, a descentralização dos sistemas de controle tornou-se factível. Ao invés de um único computador controlar toda a planta, empregam-se vários computadores dedicados a controlar partes da planta. Atualmente, os sistemas de controle são organizados segundo uma hierarquia de funções de controle; em cada nível da hierarquia são empregados computadores com características diferentes, interligados por uma rede local (veja a figura 2.2 na página 6).

As funções mais simples e com maior dependência de tempo (aquisição de dados e controle regulatório) são executadas por microcomputadores dedicados e compõem o nível inferior da hierarquia. As funções mais complexas, relacionadas ao gerenciamento da planta, possuem menor dependência de tempo e são executadas por computadores de uso geral.

No nível mais baixo, os *Controladores Digitais Especializados* são microcomputadores dedicados ao controle de uma função específica, como por exemplo o controle da temperatura num reator químico ou o controle de um robô. No nível mais alto, os computadores do *Nível de Gerenciamento* executam o gerenciamento e supervisão das operações de toda uma fábrica. Neste nível, empregam-se computadores de uso geral.

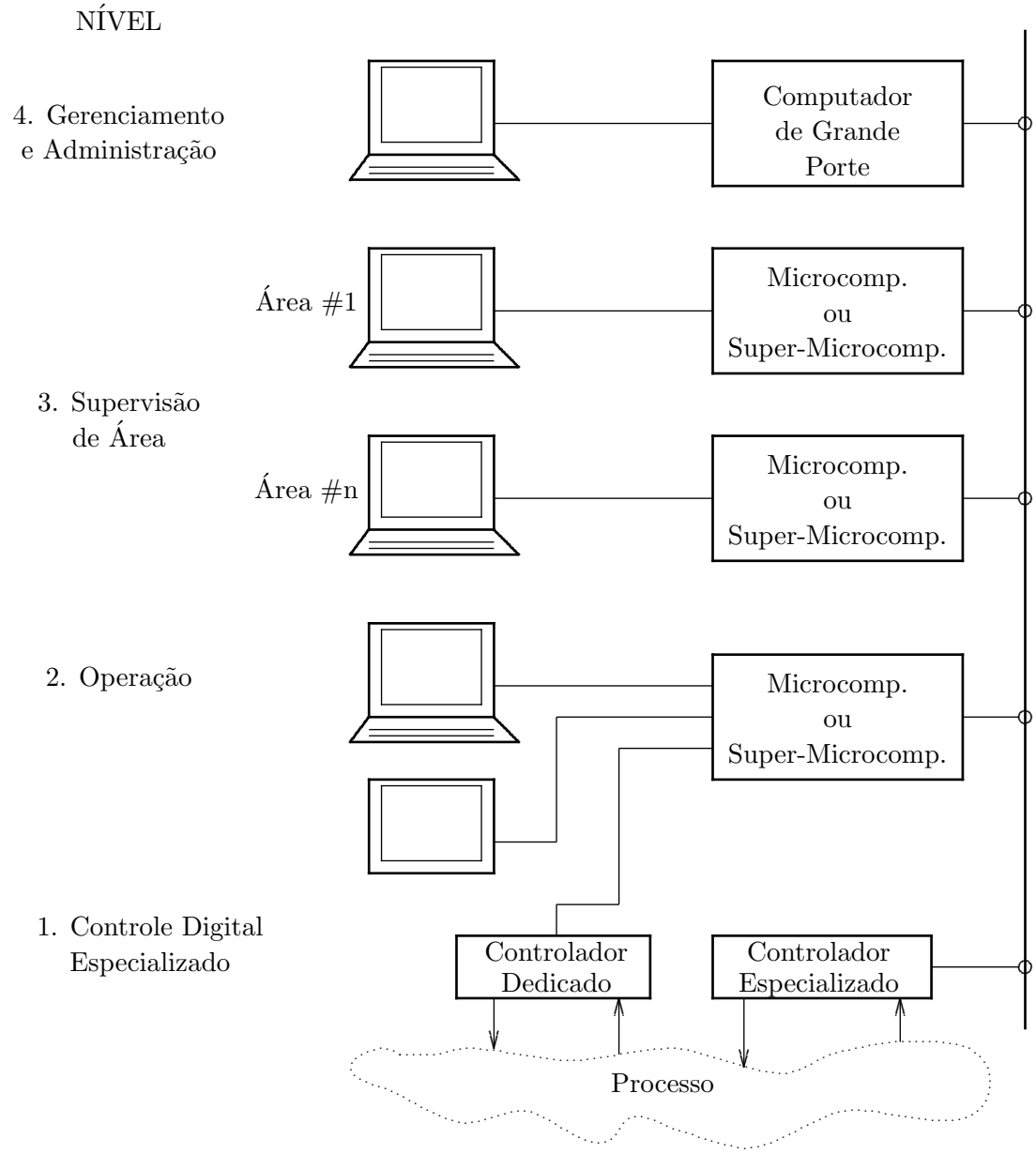

Figura 2.2: Sistema Hierárquico de Controle

Os computadores empregados nos níveis inferiores da hierarquia de controle, são dedicados a tarefas específicas de monitoração e/ou controle regulatório. Após a fase de depuração e testes de um determinado programa, ele é carregado na máquina (possivelmente em ROM) e só será alterado novamente no caso da detecção de um erro, alteração do algoritmo de controle ou alguma eventual modificação na planta. Em sistemas integrados de manufatura, os computadores de controle são interligados por uma rede local e carregados com programas de fabricação de peças através dela. Tipicamente, estes programas são carregados diariamente (ou semanalmente), dependendo dos pedidos de fornecimento recebidos pelo departamento comercial da empresa.

Nos níveis intermediários da hierarquia (coordenação e controle supervisório), a função dos computadores é a de supervisão sobre uma parte do processo de produção (uma *área*).

Estas máquinas devem manter atualizadas as bases de dados que descrevem o estado da área supervisionada, receber os comandos do nível superior, decodificá-los e disparar os comandos apropriados aos níveis inferiores. O conjunto de programas que executa estas operações também se mantém razoavelmente "estático" ao longo do tempo.

Nos níveis superiores (gerenciamento e programação da produção), são mantidas as informações relevantes ao gerenciamento de todo o processo de produção (toda a fábrica). Em função dos novos pedidos recebidos dos clientes e dos que já estão sendo atendidos, a programação de produção é atualizada. Isso provoca o envio de comandos aos níveis intermediários para que a nova programação seja cumprida; os estoques de matéria prima são avaliados e se for o caso, ordens de compra expedidas.

2.4 Sistemas de Tempo-Real

Os computadores dos níveis inferiores de um sistema de controle hierárquico e distribuído interagem diretamente com a planta através da medida de variáveis que descrevem o estado da planta e da atuação sobre ela. Os algoritmos de controle, em função do estado da planta, regulam a ação dos atuadores para que os valores desejados das variáveis controladas sejam atingidos e mantidos. Geralmente, estes computadores executam continuamente um conjunto fixo de programas e possuem interfaces especializadas para a aquisição de dados e atuação sobre a planta.

Estas máquinas devem monitorar e atuar sobre um grande número de eventos externos relacionados ao funcionamento da planta. Estes eventos podem ser periódicos ou esporádicos e isto impõe restrições temporais bastante sérias sobre os programas que implementam os algoritmos de controle e sobre os sistemas operacionais que os suportam. Os dados relativos a um determinado evento devem ser lidos antes que percam significado; uma ação de controle deve ser executada antes que as condições que a determinaram se modifiquem.

O parágrafo anterior caracteriza os Sistemas de Tempo-Real. Nestes sistemas, o atendimento consistente aos eventos relacionados com a dinâmica do processo controlado é fundamental. A cada evento do processo pode ser associada uma prioridade de tratamento, em função da sua importância tanto do ponto de vista de controle quanto do de segurança. O controlador deve possuir um sistema de interrupções com funcionalidade adequada ao reconhecimento destes eventos e os programas que ele executa devem ser simples e extremamente eficientes [WIRT77].

Um sistema operacional para suportar aplicações de tempo-real deve possuir características bem definidas quanto a tempo de resposta a eventos externos e quanto ao custo incorrido pelo uso de seus serviços. Os valores de tempo de resposta a interrupções e de custo dos serviços devem ser conhecidos para que seja possível a previsão do comportamento do controlador nas várias combinações possíveis de eventos e estados da planta.

O não-determinismo na ocorrência dos eventos faz com que o sistema como um todo tenha um comportamento potencialmente não-determinista. Uma das funções primordiais de um bom sistema operacional de tempo-real é minimizar o número de situações em que o não-determinismo dos eventos torna o comportamento do sistema global não-determinista, isto é, não-previsível, podendo portanto operar fora dos limites de tempo esperados em situações normais.

Nestas condições, a correção de um programa é essencialmente probabilista; isto pode ser parcialmente controlado pelos próprios programas de aplicação. Por exemplo, se a execução de uma operação dura um intervalo `t`, o programa deve supor que ela ocorre num

tempo `t + d`, prevendo a ocorrência de eventos de maior prioridade durante a execução desta operação. Quanto maior o valor de `d`, maior a probabilidade de correção do programa. Quanto menores forem os tempos de execução das funções suportadas pelo sistema operacional, maior será a probabilidade de correção dos programas porque a sobrecarga incorrida pela utilização dos seus serviços tende a influir menos no comportamento do sistema.

2.5 Especificação de um Equipamento

As seções anteriores deste Capítulo apresentaram as características mais importantes dos sistemas de controle industrial. Apesar de não discutido explicitamante, são muito importantes as questões de tolerância a falhas dos equipamentos industriais e sua capacidade de suportar agressões mecânicas e elétricas provenientes do meio ambiente industrial.

Como citado anteriormente, os computadores para aplicações industriais devem possuir interfaces de comunicação que permitam a construção de sistemas distribuídos fracamente acoplados[1], além de sistemas operacionais adequados a aplicações de tempo-real. Estas e outras características condicionam as decisões a serem tomadas pelo projetista de equipamentos e sistemas para aplicações industriais.

Estudos relacionados a sistemas avançados de controle industrial, realizados no Instituto de Automação do CTI, resultaram na especificação de um conjunto de módulos (processadores, memória, etc...) capazes de satisfazer a uma ampla gama de aplicações em controle e automação. Além do equipamento, as características de um sistema operacional para suportar os programas de aplicação foram levantadas.

O sistema de controle deveria atender aos seguintes requisitos:

- A carga computacional oferecida pelos algoritmos de controle e de otimização exige o uso de processadores de 16 ou 32 bits.
- A modularização do sistema possibilita a construção de máquinas para aplicações em todos os níveis da hierarquia de um sistema de controle industrial.
- A possibilidade de construção de multiprocessadores permite a obtenção de ganhos de desempenho sem aumento proporcional nos custos do equipamento.
- As tarefas de condicionamento de sinais e atuação direta sobre a planta podem ser efetuadas por subsistemas inteligentes de E/S, com ganhos no desempenho global do sistema.
- As máquinas devem possuir interfaces de comunicação via rede local, para a implementação de sistemas distribuídos.
- O sistema operacional das máquinas deve suportar programação de aplicativos com características de execução em tempo-real.

A partir destes requisitos, foi projetado no Instituto de Automação o *Multiprocessador para Sistemas de Controle* (MSC). A arquitetura do MSC e os módulos de que é composto são descritos no Capítulo 4.

[1] Este termo é definido no próximo Capítulo.

Capítulo 3

Multiprocessadores

Os *multiprocessadores* pertencem a uma classe de máquinas que explora o paralelismo de execução de programas para obter ganhos na velocidade de processamento. Existem muitas possibilidades de implementação de máquinas capazes de executar programas com alguma forma de concorrência. Este capítulo localiza os multiprocessadores no espectro dos sistemas computacionais através da apresentação de algumas das taxonomias que permitem o enquadramento de uma determinada máquina numa das classes possíveis. Feito isso, discutimos aspectos relativos à arquitetura e programação de algumas alternativas para a construção de multiprocessadores.

3.1 Taxonomias de Sistemas Distribuídos

Os sistemas computacionais podem ser grosseiramente divididos em dois grandes grupos: sistemas centralizados e sistemas distribuídos. Num sistema centralizado, os programas são executados por um único processador[1]; num sistema distribuído, a carga computacional é dividida entre mais de um processador. Os sistemas distribuídos podem ser classificados segundo várias taxonomias diferentes. Nas próximas seções, examinaremos algumas das mais difundidas.

3.1.1 Taxonomia Baseada na Granularidade das Interações

Os sistemas distribuídos podem ser classificados em função da granularidade das interações entre os componentes do sistema [JONE80]. Estas interações resultam do intercâmbio de dados e/ou da sincronização entre os processadores que cooperam na execução de programas. Em alguns sistemas as interações são infreqüentes mas envolvem grandes volumes de dados quando ocorrem. Noutros, as interações são muito freqüentes e a troca de dados ou sincronização ocorre a nível de instrução de máquina.

As *Redes de Computadores* são sistemas com *granularidade de paralelismo grossa*, onde cada processador é uma entidade independente dos demais e dedica apenas uma parte de seu tempo e recursos a tarefas comuns (bancos de dados, periféricos especiais, etc...). Os processadores são geograficamente dispersos e comunicam-se através de linhas seriais de alta velocidade. As interações entre as máquinas são pouco freqüentes mas envolvem grandes volumes de dados (transferência de arquivos, por exemplo).

[1] Alguns sistemas centralizados empregam, além da Unidade Central de Processamento, um ou mais microprocessadores para o controle de periféricos.

Os *Multi-computadores* são sistemas com *granularidade de paralelismo média*, onde cada processador pode executar um programa diferente mas o conjunto dos processadores forma uma entidade única. Todos os recursos do sistema são usados para a realização de uma mesma tarefa, sob o gerenciamento de um mecanismo de controle centralizado ou distribuído. A freqüência e a quantidade das informações trocadas entre os processadores são significativamante maiores que numa rede de computadores. A forma de comunicação e a topologia são características importantes da arquitetura do sistema.

Dois tipos de sistemas com *granularidade de paralelismo fina* são os *Processadores Vetoriais* e as *Máquinas de Fluxo de Dados*. Os Processadores Vetoriais ou Matriciais são máquinas projetadas para a solução de uma classe restrita de problemas. Os problemas em que estas máquinas são aplicadas com eficiência envolvem a manipulação de vetores ou matrizes. Os processadores vetoriais são construídos de tal forma que uma mesma operação pode ser executada simultaneamente sobre vários elementos de um vetor ou matriz. As Máquinas de Fluxo de Dados [GURD85] são uma classe de computadores projetados segundo uma filosofia diferente daquela de von Neumann. Nas máquinas de fluxo de dados, as instruções de um programa são executadas assim que seus operandos estiverem disponíveis, permitindo a exploração automática do paralelismo embutido nos programas e algoritmos. A granularidade da concorrência de execução nestas máquinas é extremamente fina.

3.1.2 Taxonomia Baseada em Grafos de Controle

Os sistemas computacionais podem ser classificados em função do grafo de controle típico dos programas que suportam [GAJK85, DUTR86]. Um programa pode ser mapeado sobre um grafo, de tal forma que os nós representem as operações sobre os dados e as arestas determinem uma ordem (parcial) em que estas operações devem ser executadas. As arestas podem representar dependências de dados (nós sucessores dependem dos valores produzidos em nós antecessores), ou dependências de controle (a ordem de "execução" dos nós depende da função do programa/algoritmo). Num modelo *serial* de computação, o grafo de controle é totalmente ordenado: os nós são executados serialmente. Num modelo *paralelo*, o grafo de controle é parcialmente ordenado: alguns nós podem ser executados paralelamente, desde que o ordenamento parcial do grafo seja obedecido e existam os recursos necessários para a execução em paralelo.

Esta classificação deve ser aplicada em função de quatro níveis de controle: programa, tarefa, processo e instrução. Um programa consiste de uma ou mais tarefas. Uma tarefa é a unidade de trabalho atribuída a um ou mais processadores. Uma tarefa, por sua vez, é composta por um ou mais processos. Finalmente, um processo é um conjunto de instruções que devem ser executadas em um único processador (os processos são indivisíveis quanto à sua alocação).

Tomando por base os grafos de controle e a forma como estes grafos são "executados", os sistemas computacionais podem ser classificados em, basicamente, quatro classes de arquitetura. A classe com *controle único serial* é formada pelos mono-processadores convencionais. Nestas máquinas, um programa consiste de uma única tarefa, executada serialmente por um único processador.

A classe com *controle duplo serial-paralelo* contém os processadores vetoriais e matriciais e aqueles que empregam unidades segmentadas em linha (*pipelines*)[2]. Os programas

[2]Um processador com unidades operacionais em linha opera da mesma maneira que uma linha de montagem de automóveis. Cada unidade da linha executa uma "parte" da instrução. Potencialmente, são

destas máquinas consistem de tarefas ou processos executados serialmente, com a execução das instruções em paralelo (potencialmente). Estas máquinas exploram o paralelismo de granularidade fina, inerente às operações sobre vetores, por exemplo.

A classe de arquiteturas com *controle duplo paralelo-serial* engloba os multi-computadores. Os programas nesta classe de máquinas são constituídos por tarefas ou processos com grafos de controle paralelos (o paralelismo de execução ocorre a nível de procedimento). A granularidade de paralelismo nestas máquinas é média ou grossa.

As máquinas da classe *controle triplo serial-paralelo-paralelo* são uma fusão das máquinas das duas classes anteriores. O grafo de controle dos programas destas máquinas é serial e cada nó representa uma tarefa. Cada tarefa pode ser dividida em processos; tanto os processos como as suas instruções podem ser executados em paralelo. Um exemplo de máquina nesta classe é o CRAY X-MP-1, que é composto por dois ou quatro processadores vetoriais executando partes de programas em paralelo.

3.1.3 Outras Taxonomias

Outra classificação de sistemas computacionais é aquela proposta por Flynn [FLYN72]. Esta classificação comporta três classes de sistemas:

- *Single Instruction Single Data* (SISD) - as máquinas desta classe são os monoprocessadores convencionais, onde um único fluxo de dados é processado por um único fluxo de instruções.

- *Single Instruction Multiple Data* (SIMD) – as máquinas desta classe são os processadores vetoriais. Neste caso, uma única instrução é executada sobre vários fluxos (paralelos) de dados.

- *Multiple Instruction Multiple Data* (MIMD) - nesta classe estão incluídos os multicomputadores. As máquinas desta classe são capazes de executar vários programas independentes, cada programa processando um conjunto de dados distinto dos demais.

Ainda outra classificação, proposta por Enslow [ENSL78], usa um espaço tridimensional para caracterizar os sistemas distribuídos. As três dimensões de análise são:

- A distribuição das unidades de processamento. Esta dimensão corresponde à organização física das máquinas, que pode variar de uma única UCP até um sistema geograficamente distribuído.

- A organização do controle. Esta dimensão pode variar de um sistema com as decisões de controle originárias de um único ponto fixo, até um sistema composto por um conjunto homogêneo de processadores com controle compartilhado entre eles.

- A distribuição dos dados. Os sistemas podem conter uma base de dados centralizada, num extremo, ou, completamente pulverizada, noutro.

Os multi-computadores, em particular, podem ser classificados em função do compartilhamento de um espaço de endereçamento pelos processadores. Os sistemas onde os processadores não tem acesso a uma área comum de memória e comunicam-se através de canais de Entrada/Saída são chamados *Sistemas Fracamente Acoplados*. Os sistemas onde

executadas várias instruções em paralelo ao longo da linha.

os processadores compartilham um espaço de endereçamento, isto é, uma área de memória comum, são chamados *Multiprocessadores* ou *Sistemas Fortemente Acoplados*.

Em geral, a inclusão de um sistema real dentro de uma das classificações anteriores não é simples. Num multiprocessador, o nível e a forma de cooperação entre os processadores são definidos tanto pela arquitetura do sistema como pelo sistema operacional e programas aplicativos. Os processadores podem executar programas onde o intercâmbio de dados é raro e segundo um modelo de troca de mensagens (mesmo que os processadores tenham acesso à memória compartilhada). Este mesmo sistema pode operar como uma linha de montagem sobre um conjunto de dados, com freqüentes trocas de resultados parciais (como numa máquina SIMD).

3.2 Multiprocessadores

Como citado na seção anterior, os multiprocessadores são um caso particular dos multicomputadores. Nestas máquinas, os processadores compartilham um espaço de endereçamento. O espaço comum de endereçamento contém um bloco de memória que pode ser acessado para escrita e/ou leitura por todos os processadores e que é o meio de comunicação de baixo nível entre eles. Um espaço de endereçamento compartilhado é implementado com um ou mais barramentos. Um *barramento paralelo* é uma estrutura física que permite aos módulos a ele conectados o acesso a um conjunto de linhas (ligações elétricas) que transportam dados, endereços e os sinais de controle que identificam o tipo de operação (leitura/escrita em memória, relógio,...). O barramento é chamado de paralelo porque todas as linhas transportam informação simultaneamente[3] . Um *barramento serial* transmite um único sinal por unidade de tempo através de uma única linha de sinal (dois fios). Um conjunto de sinais é serializado para a transmissão pelo barramento, isto é, em cada unidade de tempo, o valor de um dos sinais é aplicado ao meio de comunicação.

Tanto do ponto de vista lógico como do ponto de vista físico, um barramento é a forma de conexão mais simples entre processadores e memória. Quando mais de um processador tenta acessar a memória, ocorre um *conflito de acesso* ou *contenção* pelo acesso à memória. O protocolo de acesso ao barramento deve usar alguma política que permita a solução do conflito, de tal forma que os processadores acessem a memória de uma forma organizada. Um *árbitro de acessos* implementa a política que resolve qual das requisições simultâneas de acesso ao barramento é atendida. A figura 3.1 mostra a estrutura de um multiprocessador baseado em barramento.

Figura 3.1: Multiprocessador baseado em barramento.

O barramento é uma estrutura de interconexão que não permite transferências simultâneas entre mais de um par processador–memória, tornando-se facilmente o ponto

[3]Existem barramentos em que algumas linhas são usadas para o transporte de dados e endereços; um sinal de controle indica o tipo de informação válida em cada instante. Estes barramentos são chamados de "barramentos multiplexados".

de estrangulamento do sistema. Existem basicamente duas abordagens para a solução deste problema. A primeira delas consiste do uso de múltiplos barramentos, distribuindo através deles o tráfego de dados e/ou instruções. A segunda consiste do uso de técnicas de projeto de sistemas de memória que permitem uma redução de tráfego no barramento global. As próximas seções discutem estas abordagens.

3.2.1 Múltiplos Barramentos

Num sistema com múltiplos barramentos, o código dos programas reside em módulos de memória acessíveis somente ao processador que vai executá-lo, isto é, na memória local a cada processador (veja a figura 3.2). Nesta arquitetura, não ocorrem ciclos de busca de instruções no barramento global, que é usado apenas para transferências de dados entre os processadores ou entre estes e os periféricos. Os processadores compartilham de um espaço de endereçamento onde está localizada a memória global. Cada conjunto processador/memória local/periféricos locais é uma entidade independente dos demais; a cooperação entre estes conjuntos se dá através da memória global. Três sistemas construídos com esta arquitetura são o TOMP System [CONT85], o SUMUS [NETT86, CORS86] e o PP [CAVa87, CAVb87].

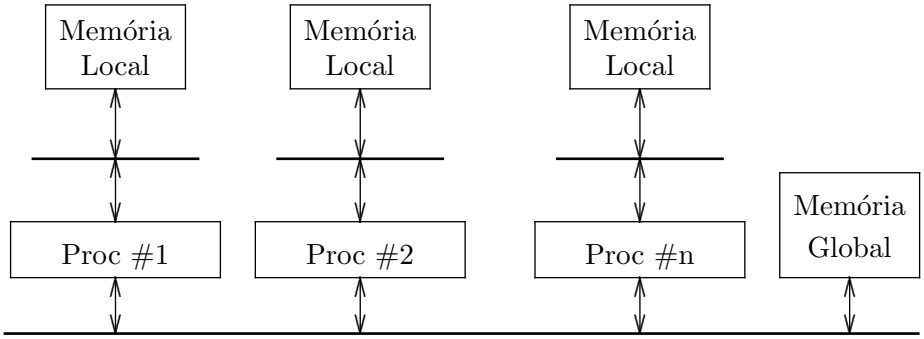

Figura 3.2: Arquitetura baseada em múltiplos barramentos.

O núcleo do sistema operacional do PP, chamado de PP-SO/P [CAVa87] é discutido abaixo. No PP, cada processador executa o código armazenado na memória interna ao módulo processador (a sua memória local); isto vale tanto para os programas de aplicação quanto para o código do PP-SO/P. O núcleo suporta as construções da linguagem de alto nível CHILL (CCITT High Level Language) com algumas extensões. CHILL é uma linguagem para a programação de sistemas de controle de tempo-real que devem operar com alto grau de confiabilidade (aplicações em telefonia).

A unidade lógica de computação no PP-SO/P é o processo. Um processo é identificado pela sua "instância" (veja a figura 3.3). Um programa é composto por um ou mais processos que podem executar concorrentemente em um ou mais processadores. Um mesmo processo (código) pode ser executado em várias encarnações diferentes. As instâncias de processos são criadas pela execução do comando START, que é similar ao `fork` do UNIX [RITC74]. Os processos podem pertencer a uma de três classes de prioridade; cada classe suporta oito níveis de prioridade.

A sincronização entre os processos pode ser obtida por meio dos comandos EVENT e CONTINUE; o acesso a "regiões" pode ser controlado pelos comandos LOCK e UNLOCK.

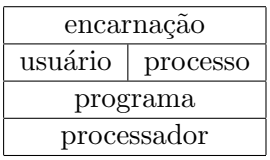

Figura 3.3: Instância de um processo no PP-SO/P.

A sincronização e a comunicação interprocessos ocorrem pela troca de mensagens, via comandos SEND, RECEIVE e RECEIVALL[4]. As mensagens são entregues aos destinatários independentemente da sua localização (o destinatário de uma mensagem é identificado pela sua instância). O PP-SO/P oferece comandos para a conexão de um processo a um relógio ou a uma interrupção. Ainda, existem comandos que permitem a suspensão temporária da execução de um processo (SLEEP) e o controle da ocorrência de interrupções (habilitando-as ou não). Os tratadores de Entrada/Saída são organizados segundo uma hierarquia de três níveis funcionais: o primeiro nível controla os dispositivos propriamente ditos; o segundo funciona como amortecedor entre as velocidades relativas entre o primeiro e o terceiro níveis; o terceiro, oferece uma interface de alto nível aos programas de aplicação. O sistema de arquivos é similar ao do UNIX e permite o acesso de um processador a arquivos locais ou remotos.

3.2.2 Barramento Global + Memória Cache

Uma alternativa de construção de multiprocessadores é aquela onde os processadores compartilham todo (ou quase todo) o seu espaço de endereçamento. Neste caso, os processadores executam o código e acessam os dados armazenados em memória compartilhada (veja a figura 3.1). O barramento global é o gargalo do sistema porque todos os acessos a instruções e dados ocorrem através dele. Esta situação pode ser resolvida pelo uso de *memória cache* para instruções ou para instruções e dados entre cada um dos processadores e o barramento global. Esta arquitetura é mostrada na figura 3.4.

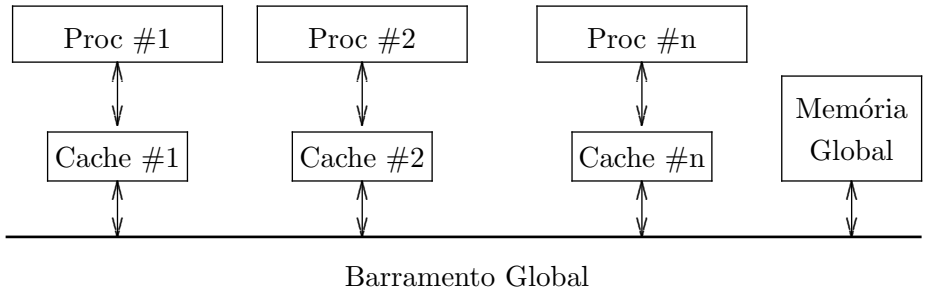

Figura 3.4: Multiprocessador baseado em memória cache.

Uma *memória cache* (ou, memória "escondida") é uma memória do tipo associativo[5], de altíssima velocidade de acesso, de projeto complexo e custo elevado. O seu tamanho varia usualmente de 4 a 32 Kpalavras. O seu funcionamento é baseado nos princípios da

[4]As referências sobre o PP e o PP-SO/P citadas acima, não contém nenhuma definição para o termo *region* nem para a semântica das primitivas. Supõe-se que seus significados sejam aqueles normalmente associados a estas palavras. A maneira como a "instância" de um processo é usada para identificá-lo também não é explicada.

[5]Endereçável pelo seu próprio conteúdo.

Localidade Espacial e da Localidade Temporal. O princípio da Localidade Espacial afirma que os últimos acessos executados por um processador, e os próximos a serem executados, são todos referências a uma faixa restrita de endereços (o contador de programa é incrementado de um pequeno valor, na grande maioria das instruções). O princípio da Localidade Temporal reza que os programas tendem a acessar periodicamente a um mesmo conjunto de endereços (*loops*). A finalidade de uma memória cache é permitir um acesso rápido às últimas posições referenciadas pelo processador. A *taxa de acerto* é a razão entre a quantidade de acessos que o processador executa e o número de vezes em que estes acessos são "atendidos" pela cache, sem a necessidade de um acesso à memória do sistema (que, em geral, tem tempos de acesso muito maiores que os da cache).

Nos mono-processadores, a finalidade de uma memória cache é diminuir o tempo de acesso à memória, que usualmente é o fator limitante do desempenho do sistema (em algumas implementações, a memória cache é capaz de antecipar-se ao processador na busca de instruções). Num multiprocessador, a finalidade da cache é reduzir o tráfego no barramento global, que é o fator limitante neste caso. As caches devem possuir um mecanismo que assegure a consistência entre os valores armazenados na memória global e na memória cache de cada um dos processadores, de tal forma que quando um valor presente na cache de um processador é atualizado, o novo valor seja disseminado para toda a memória do sistema (memória global e as outras caches).

O sistema operacional para um multiprocessador com esta arquitetura é mais complexo que o de um mono-processador. A integridade das estruturas de dados do sistema num mono-processador é facilmente mantida com o controle da ocorrência de interrupções. Num multiprocessador, as estruturas de dados armazenadas em memória global são acessadas concorrentemente pelos processadores e sua integridade depende de mecanismos mais complexos do que o controle das interrupções. Estes mecanismos podem ser implementados no árbitro do barramento ou por algoritmos que impeçam a manipulação de um conjunto de dados por mais de um processador a cada instante [BACH86].

Em alguns sistemas, somente um processador (o "mestre") executa o código do sistema operacional; a integridade dos dados é mantida pelo controle das interrupções neste processador. Os outros processadores (os "escravos") executam os programas de aplicação. Quando um aplicativo necessita de um serviço do sistema operacional, o mestre é notificado da requisição e a atende assim que possível. Na medida em que o número de processadores escravos presentes numa máquina aumenta, o seu desempenho como um todo passa a depender do desempenho do processador mestre.

Outra solução é aquela empregada nos *multiprocessadores homogêneos*. Nestas máquinas, todos os processadores podem executar o código do sistema operacional e dos aplicativos indistintamente. Um mesmo programa pode ser executado em processadores diferentes em instantes diferentes. Usualmente, estas máquinas são comercializadas com suporte a sistemas operacionais compatíveis com o UNIX, adaptados para a execução em multiprocessamento. A estação de trabalho Firefly [THAC87] e o Pégasus [FALL87] são exemplos destas máquinas. O Firefly suporta o sistema Ultrix e o Pégasus o Plurix.

O Firefly, por exemplo, explora paralelismo de execução com granularidades média e grossa porque permite a existência simultânea de vários espaços de endereçamento, com várias linhas de controle atuando sobre cada um deles. A comunicação entre os processadores e entre os espaços de endereçamento é suportada de uma maneira uniforme por chamadas remotas de procedimento (chamadas de procedimento que são atendidas remotamente, isto é, em outro espaço de endereçamento ou por outro processador). O ambiente de execução do Ultrix permite uma única linha de controle num espaço de endereçamento

(granularidade grossa). O sistema Topaz, que oferece o suporte de execução para o Ultrix, permite a existência de mais de uma linha de controle num espaço de endereçamento (granularidade média) e implementa o mecanismo de chamada remota de procedimento. O Topaz ainda oferece um sistema de arquivos remotos e um gerenciador de janelas que pode ser compartilhado por todos os processos.

3.2.3 Barramentos Comutados

Além das alternativas das seções anteriores, existe outra possibilidade de construção de multiprocessadores, baseada em barramentos comutados (*crossbar switch*) – veja a figura 3.5. Estas máquinas possuem dois tipos de barramentos: *de processador* e *de memória*. A figura mostra os barramentos de processador nas linhas e os de memória nas colunas. Os "comutadores" ou "chaves", colocadas em cada cruzamento, podem fazer a ligação física entre dois barramentos, um de cada tipo.

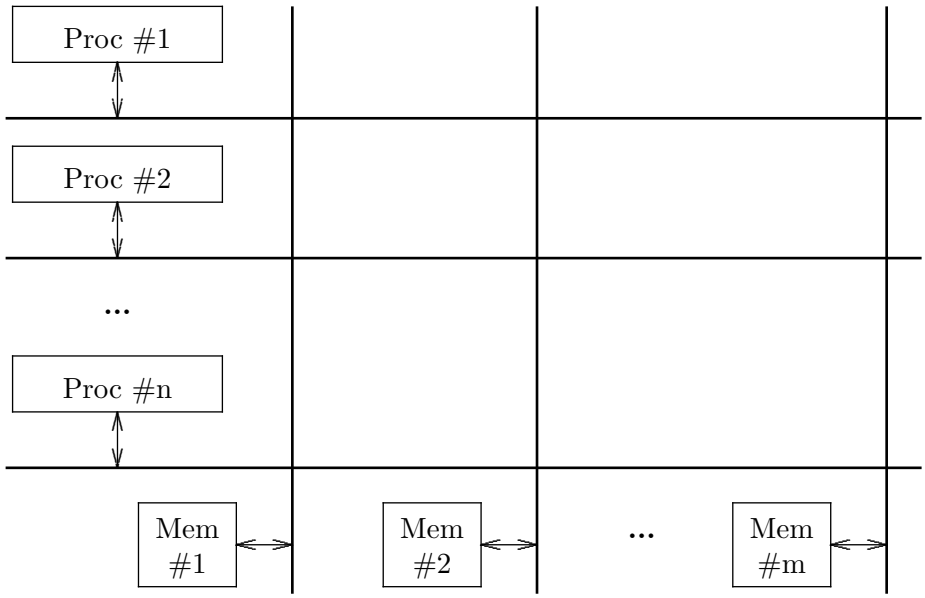

Figura 3.5: Multiprocessador baseado em barramentos comutados.

Os barramentos horizontais contém um processador, memória local e periféricos locais. Estes barramentos dispensam um árbitro de acessos porque apenas um processador é conectado a eles. Os barramentos verticais contém os módulos de memória que são compartilhados pelos processadores; estes barramentos são multimestre e necessitam, portanto, de árbitros de acesso. As chaves fazem a conexão entre os dois barramentos necessários à formação de um caminho entre um processador e um módulo de memória.

Um exemplo clássico desta arquitetura é o C.mmp (Carnegie-Mellon Multi-Mini-Processor) [WULF74]. O C.mmp é composto por 16 processadores Digital Equipment Corporation PDP 11/40, 16 módulos de memória com 32 Kpalavras e uma matriz de 16 x 16 chaves. Um processador pode gerar interrupções nos demais processadores num de quatro níveis de prioridade. Um mecanismo de relocação em cada um dos barramentos de processador faz o mapeamento dos endereços virtuais nestes barramentos para os endereços físicos na memória compartilhada.

O núcleo de sistema operacional do C.mmp, chamado HYDRA [WULF74], oferece

uma série de mecanismos que suportam a abstração de *recursos* e *objetos* (encarnações de recursos). Estes mecanismos permitem a criação e representação de novos tipos de recursos e a definição de operações sobre eles, acesso protegido a objetos e domínios de execução. A proteção no acesso à objetos é obtida pelo uso de *capabilities*. O acesso a um objeto só é permitido se a capacidade para tanto existir; uma *capability* especifica quais as operações que seu dono pode executar sobre o objeto especificado nela. Deve existir uma *capability* para cada objeto passível de ser referenciado por um programa, "protegendo" os objetos contra acessos indevidos. Um *domínio de execução* é o ambiente de execução de um procedimento. A geração de domínios é dinâmica. Cada procedimento possui um conjunto de *capabilities* internas que forma o "gabarito" do seu domínio. Na invocação de um procedimento, as *capabilities* passadas como parâmetro mais aquelas do gabarito formam o novo domínio de execução. Quando um procedimento é invocado, o domínio de execução muda na sua interface, isto é, cada procedimento tem o seu próprio conjunto de *capabilities*.

Os procedimentos suportados pelo HYDRA (código + dados + *capabilities*) são de natureza estritamente sequencial. A unidade de processamento assíncrono é o *processo*. Um processo é a menor unidade lógica que pode ser executada individualmente por um processador. A execução de um processo consiste de um registro das mudanças de domínios de execução introduzidos por uma seqüência de chamadas de procedimento. A comunicação interprocessos é suportada por filas de mensagens; a sincronização por semáforos [DIJK68].

3.2.4 Aplicações e Programação

As arquiteturas apresentadas neste capítulo tem seus nichos de aplicação e modos de programação razoavelmente distintos. Os multiprocessadores baseados em múltiplos barramentos são usados principalmente em sistemas dedicados de controle ou supervisão. Os programadores devem conhecer a arquitetura da máquina para programá-la de modo a tirar máximo proveito do paralelismo real de execução (programação em baixo nível). Os multiprocessadores baseados em memória compartilhada oferecem uma boa relação custo/desempenho e eles são muito usadas em aplicações de processamento de dados (muitos deles suportam sistemas derivados do UNIX, por exemplo).

O maior problema dos sistemas baseados em barramentos chaveados é o custo e a complexidade das chaves. Uma chave deve interconectar dois conjuntos de linhas de dados (2x16 ou 2x32), de linhas de endereço (2x16 ou 2x24 ou 2x32) e linhas de controle. Mesmo que as chaves sejam implementadas em circuitos integrados VLSI, seu custo e a latência introduzida pelas conexões e desconexões desestimulam novos desenvolvimentos com esta arquitetura.

Na medida em que o desempenho dos dispositivos eletrônicos se aproxima dos limites do fisicamente realizável[6], maior tem sido a ênfase na pesquisa de sistemas de processamento paralelo; e as propostas de arquiteturas paralelas tem proliferado nos últimos anos [KUNG88]. Algumas destas propostas frutificaram, como por exemplo o Hypercube [SEIT85], o Transputer [WILS83] e Warp Machine (iWarp) [KUNG88]. Outras, ainda estão em fase embrionária, como as redes neuronais [TREL88], por exemplo. Segundo [TREL88], as grandes tendências no desenvolvimento de sistemas computacionais são a proliferação dos *Parallel Unix Systems*[7] e a consolidação de Programação Funcional [MEIR88] como uma ferramenta de programação.

[6] Folclore tecnológico: *light travels one feet per nanosecond*.

[7] O Firefly e o Pégasus são exemplos destes sistemas.

Capítulo 4

A Arquitetura do MSC

A arquitetura do MSC foi concebida no Instituto de Automação do CTI, tendo como base os requisitos funcionais descritos no Capítulo 2. O MSC é baseado em três tipos de módulos que podem ser combinados de maneiras diferentes possibilitando a construção de toda uma família de máquinas. Os módulos básicos são: *processador* baseado no microprocessador Motorola MC 68000, *processador de E/S* (PE/S) baseado no microprocessador Zilog Z80 e *memória*.

Máquinas com diferentes capacidades de processamento podem ser construídas a partir da combinação dos módulos básicos. Para aplicações que demandem muita capacidade de processamento (níveis superiores da hierarquia de controle, por exemplo), uma configuração com vários processadores e um PE/S seria adequada. Noutro extremo, uma aplicação com muita interação com o mundo externo e carga computacional relativamente baixa, seria atendida por uma configuração com um processador e vários PE/S. Nas duas últimas seções deste capítulo, discutimos propostas de implementação de controladores usando estes módulos.

Deste ponto em diante, sempre que nos referirmos ao MSC, estaremos supondo uma configuração com no mínimo dois processadores e um ou mais PE/S.

4.1 A Arquitetura do MSC

O MSC é uma máquina com "controle duplo paralelo serial" porque o paralelismo de execução ocorre a nível de procedimento. Ainda, o MSC pertence à categoria das máquinas MIMD (*Multiple Instruction Multiple Data*) porque seus processadores executam concorrentemente programas distintos em espaços de endereçamento disjuntos (parcialmente). Sob outro ponto de vista, o MSC é um Sistema Distribuído Fortemente Acoplado ou Multiprocessador porque seus processadores compartilham de um mesmo espaço de endereçamento.

A descrição a seguir da arquitetura do MSC, é baseada na notação MSBI (*Master Slave Bus Interface*) [CONT85]. Esta notação evidencia o fluxo de dados e instruções entre os processadores e a memória mostrando as interações entre os componentes de um sistema como *requisições de acesso à memória*. Os componentes de um sistema são modelados como *mestres* ou *escravos*. Um mestre efetua uma requisição de acesso que é atendida por um escravo, isto é, o sentido de uma requisição é sempre do mestre para o escravo independentemente de o acesso ser uma leitura ou escrita em memória.

Além de mestres e escravos esta notação representa dois outros elementos: *barramentos* e *interfaces*. Um barramento suporta o fluxo de dados resultante da comunicação entre os

mestres e os escravos interconectados por ele. Uma interface "transporta" uma requisição de acesso entre dois barramentos (um local e outro externo ao módulo que contém a interface) e possui uma regra de tradução de endereços (do lado do mestre para o lado do escravo). No que concerne a requisições de acesso, a interface é sempre unidirecional. As entidades representadas nesta notação são interligadas por setas que indicam o sentido das requisições de acesso e não o sentido do fluxo de dados, que, em princípio, é bidirecional (escritas e leituras). Outros mecanismos de controle dos barramentos, tais como o subsistema de interrupções e o de arbitragem da posse do barramento são discutidos posteriormente. A figura 4.1 mostra os símbolos das entidades e a figura 4.2 mostra a arquitetura do MSC modelada de acordo com esta simbologia.

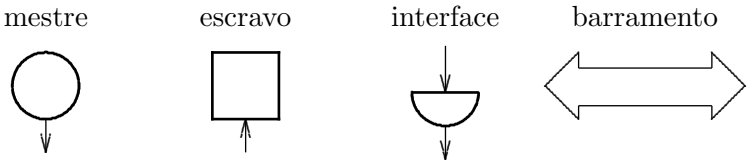

Figura 4.1: Símbolos da notação MSBI.

O MSC é estruturado sobre três barramentos distintos e dois *tipos* de memória. O barramento VME interliga os processadores com a *memória global*; o barramento VMX é uma extensão do barramento interno do módulo ao qual está conectado; o barramento ECB interliga os módulos que compõem um subsistema de Entrada/Saída.

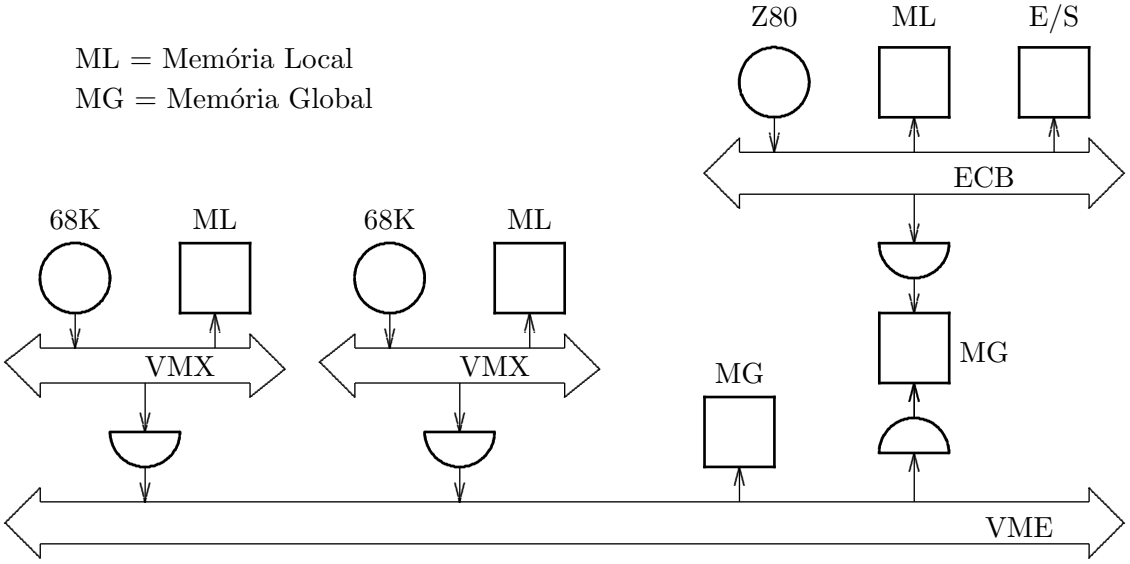

Figura 4.2: Arquitetura do MSC.

A memória do MSC é logicamente dividida em dois espaços de endereçamento: *global* e *local*. A memória global é acessada por todos os processadores através do barramento VME. A memória local somente é acessada pelo processador ao qual está ligada pelo VMX. Os processadores executam o código armazenado na sua memória local (não ocorrem ciclos de busca de instruções na memória global). A memória global contém as estruturas de dados que podem ser acessadas por todos os processadores do sistema. Esta distribuição foi escolhida porque minimiza o número de conflitos de acesso ao barramento global, o que

deve influir positivamente no desempenho do sistema. Análises de desempenho realizadas num multiprocessador semelhante ao MSC suportam esta afirmação [CONT85].

Existem dois mecanismos de comunicação de baixo nível entre os processadores e entre estes e os processadores de E/S: um processador pode sinalizar a ocorrência de um evento através de uma interrupção no VME, ou, pode depositar uma mensagem num armazenador pré-estabelecido na memória global ou na memória dual-port de um dos PE/S. Se o mecanismo de comunicação de baixo nível for baseado em interrupções, o limite máximo é de seis processadores e/ou PE/S em qualquer combinação porque são seis as linhas de interrupção mascaráveis no 68000.

Se o mecanismo de comunicação for a troca de mensagens em memória global ou dual-port, o número de processadores e/ou PE/S é limitado pelo número de posições disponíveis no barramento VME (20 posições). *Troca de mensagens* está sendo usado como um nome genérico para a comunicação entre processadores; as mensagens podem ser requisições de serviços depositadas num armazenador, mantido como uma lista encadeada de requisições e respostas. A integridade das listas é facilmente garantida pelos acessos em exclusão mútua possíveis tanto no VME como na memória dual-port dos PE/S. O mecanismo baseado em interrupções é mais eficiente que a troca de mensagens porque esta, necessariamente, envolve alguma forma de *polling* nas filas de mensagens.

Os conflitos de acesso dos processadores ao VME são resolvidos pelo árbitro (veja abaixo), instalado na posição 1 do barramento. Se o barramento estiver livre, o árbitro entrega a posse a quem solicitar; se ocorrer uma tentativa de acesso quando o barramento estiver sendo usado, o árbitro só entrega a posse após o final do acesso ou grupo de acessos corrente. Isso aumenta, em média, o tempo de acesso à memória global pelos processadores.

4.2 Componentes do MSC

As próximas sub-seções descrevem os componentes do MSC, quais sejam, os barramentos VME, VMX e ECB e os módulos Processador (CPU VME 68K), Processador de E/S (CPU ECB/VME) e Memória VME/VMX.

4.2.1 Barramento VME

O barramento padrão VME foi desenvolvido em 1981 pelas empresas Mostek, Motorola e Signetics para suportar aplicações com os microprocessadores de 16 e 32 bits como o Motorola MC 68000. O resultado é um barramento de alta velocidade, com capacidade de multiprocessamento e um eficiente subsistema de interrupções.

O padrão VME define as dimensões físicas das placas de circuito impresso, do *backplane* (o barramento propriamente dito) e do bastidor que as abriga. O dimensionamento das placas segue o padrão Eurocard duplo (233,7 x 160 mm) e elas possuem dois conectores do tipo *pin & plug* de 96 pinos (Euroconnector). O bastidor segue o padrão IEC para bastidores de 19 polegadas e pode conter conectores para até 20 placas.

São definidas três versões para o padrão: a versão *standard*[1] possui um espaço de endereçamento de 16 Mbyte e 16 linhas de dados; a versão estendida possui 4 Gbyte de espaço de endereçamento e 32 linhas de dados; a versão reduzida, 64 Kbyte de espaço de endereçamento e 16 linhas de dados. O conector superior da placa (P1) contém todos

[1] O MSC é construído com a versão *standard*.

os sinais da versão standard; o conector inferior (P2) contém as linhas adicionais de endereço e dados da versão estendida e 64 pinos que podem ser definidos pelo usuário para Entrada/Saída.

O padrão VME define um barramento paralelo (não-multiplexado) onde as transferências de dados ocorrem segundo um protocolo mestre-escravo assíncrono. Uma transferência de dados através do barramento consiste de duas fases: o mestre que deseja transferir os dados deve solicitar ao árbitro do barramento a sua posse; quando obtida, o escravo é selecionado e ocorre então o ciclo de leitura ou escrita[2]. Além das linhas de endereço, existem 6 linhas de *modificadores de endereço* que identificam o tipo de ciclo de barramento que está ocorrendo. Alguns dos códigos identificam os acessos pela largura da palavra (16/32 bits) ou pelo modo (supervisor/usuário).

A arbitragem do barramento é efetuada segundo uma de três políticas: *daisy-chain*, prioridade fixa e prioridade rotativa. Existem quatro linhas para a requisição do barramento pelos mestres: BR0$*$, BR1$*$, BR2$*$ e BR3$*$. Na política de prioridade fixa, requisições na linha BR3$*$ são as de maior prioridade de atendimento. Na prioridade rotativa, a prioridade mais alta "circula" pelas quatro linhas. Na política *daisy-chain*, a prioridade dos mestres depende da sua posição física no barramento; todas as requisições ocorrem na linha BR3$*$ e quando o árbitro entrega a posse do barramento, o mestre que estiver fisicamente mais próximo do árbitro pode interceptar o sinal da entrega e tomar a si o direito de acesso.

O subsistema de interrupções do VME possui sete linhas de interrupção com sete níveis de prioridade (IRQ1$*$ a IRQ7$*$). Um escravo (periférico) pode interromper um mestre (processador) usando uma destas linhas. Pode haver apenas um tratador e um ou mais geradores de interrupções em cada nível. Durante o atendimento de uma interrupção, o mestre que vai atendê-la solicita ao árbitro a posse do barramento e realiza um acesso semelhante a um ciclo de leitura. Três linhas de endereço identificam o nível da interrupção que está sendo atendida e o escravo coloca um vetor (um byte) nas linhas de dados para ser usado pelo mestre na identificação do escravo que originou a interrupção.

4.2.2 Barramento VMX

O barramento VMX foi especificado pelo mesmo grupo que especificou o VME (o *VME Manufacturers Group*) e possui características que o tornam atrativo como um barramento para extensão da memória local a um processador. As linhas de sinal do barramento VMX ocupam as 64 linhas do conector inferior (P2) do VME que são dedicadas a E/S e definidas pelo usuário. O padrão permite a interconexão de um ou dois mestres e de um a cinco escravos (memória ou periféricos mapeados em memória), através de um *backplane* ou cabo paralelo de 64 vias.

O VMX é bastante simples: possui um mecanismo de arbitragem para somente dois mestres (processador e controlador de Acesso Direto à Memória) e não possui subsistema de interrupções. O VMX permite transferências em 32 linhas de dados segundo um protocolo mestre-escravo assíncrono. O espaço de endereçamento é de 16 Mbyte com as linhas de endereço multiplexadas em dois grupos de 12 linhas.

[2]Isto é o que ocorre num sistema multimestre; num sistema com somente um processador, o acesso do processador ao barramento pode ficar permanentemente habilitado.

4.2.3 Barramento ECB

O barramento ECB é definido num formato mecânico similar ao do VME, com o mesmo conector *pin & plug* e cartões do tipo Eurocard simples (160 x 100 mm). As linhas de sinal são um superconjunto dos sinais do microprocessador Zilog Z80. O espaço de endereçamento é de 1 Mbyte (o do Z80 é de 64 Kbyte) e as transferências ocorrem em 8 linhas de dados segundo um protocolo síncrono. Este barramento comporta mais de um mestre e oferece duas linhas de interrupção, uma delas não-mascarável. A prioridade no atendimento das interrupções mascaráveis é definida por *daisy-chain*. Os periféricos são mapeados num espaço de endereçamento especial para E/S com 256 posições.

4.2.4 Processador CPU VME 68K

O módulo processador CPU VME 68K é o componente principal do MSC. Ele é baseado no microprocessador MC 68000 da Motorola e possui interfaces para os barramentos VME e VMX. Além do microprocessador, o módulo contém um processador de ponto flutuante MC 68881, que executa operações aritméticas, trigonométricas e transcedentais em ponto flutuante. O relógio de tempo-real do módulo é implementado com um *Programable Timer Module* – MC 6840 que contém três temporizadores que podem ser usados individualmente ou em cascata. Uma interface RS 232C (MC 6850 – *Asynchronous Communications Interface Adapter*) pode ser usada para depuração de programas com um terminal de vídeo ou comunicação por linha serial. A figura 4.3 mostra um diagrama de blocos do módulo CPU VME 68K.

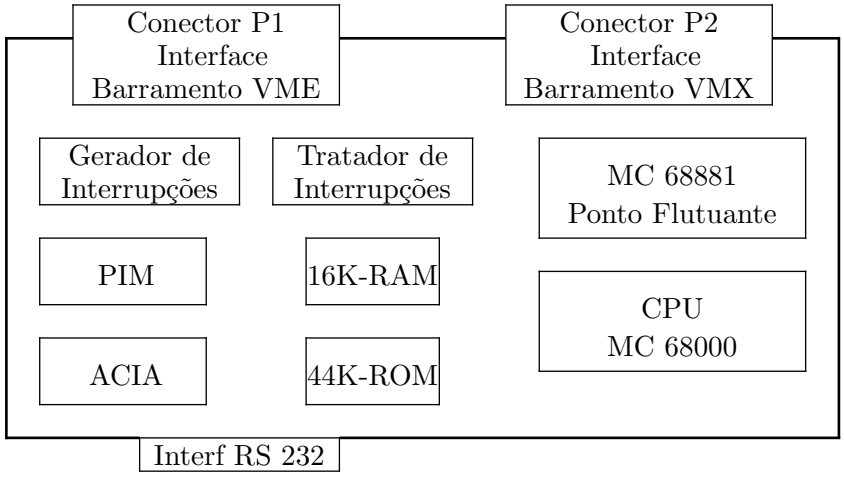

Figura 4.3: Módulo Processador CPU VME 68K.

A interface VME é do tipo mestre, capaz de endereçar 16 Mbyte com transferências em 8 ou 16 bits. Ela contém um árbitro do tipo *daisy-chain* que pode ser usado como o árbitro do sistema se o módulo for instalado na posição 1 do barramento. Uma máquina de estados implementa um gerador de interrupções capaz de provocá-las em qualquer uma das sete linhas com a seleção do nível por programa. A interface VMX também é do tipo mestre, capaz de endereçar 16 Mbyte e fazer transferências em 8 ou 16 bits. As parcelas do espaço de endereçamento do 68000 acessíveis via VME ou via VMX são programadas numa memória PROM que contém o mapeamento de endereços.

O módulo contém 44 Kbyte de memória EPROM que podem ser utilizados para a

inicialização "a frio" do processador. 16 Kbyte de memória RAM estão disponíveis e mapeados na parte inferior do espaço de endereçamento do 68000 de tal forma que os vetores de atendimento de interrupções ficam contidos em RAM e podem ser alterados durante a execução de programas. Toda a área de RAM pode ser protegida contra escrita para que os vetores de atendimento de exceções não sejam corrompidos acidentalmente.

4.2.5 Processador de Entrada/Saída CPU ECB VME

O processador de E/S (PE/S) é baseado no microprocessador Zilog Z80 e possui interfaces para os barramentos VME e ECB. Este módulo contém um controlador de acesso direto à memória (ADM) Z80-DMA capaz de controlar transferências entre a memória instalada no módulo, memória instalada no ECB e periféricos locais ao módulo ou instalados no ECB. Duas interfaces RS 232C síncronas ou assíncronas são implementadas pelo Z80 SIO – *Serial Input Output*; estes dois "canais" seriais podem ser alimentados ou descarregados pelo controlador de ADM. O relógio de tempo-real do módulo é implementado por um dos três temporizadores do Z80 CTC – *Counter Timer Circuit*. Estes três periféricos podem solicitar a atenção da CPU Z80 através de interrupções. A figura 4.4 mostra o diagrama de blocos do módulo CPU ECB VME.

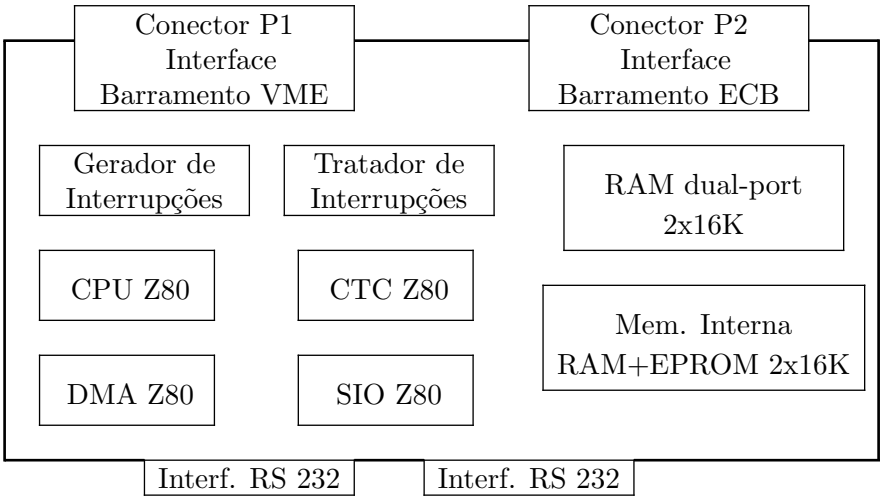

Figura 4.4: Módulo Processador de Entrada/Saída CPU ECB VME.

O espaço de endereçamento do Z80 é de 64 Kbyte e o do ECB é de 1 Mbyte. Esta ampliação é obtida dividindo-se o espaço de 1 Mbyte em 64 páginas de 16 Kbyte cada uma. Em cada instante, um conjunto de 4 páginas, totalizando 64 Kbyte, pode ser acessado diretamente pelo Z80. Esta forma primitiva de gerenciamento de memória é obtida com uma memória RAM de acesso rápido com 4 palavras de 6 bits para a geração das 6 linhas de endereço mais significativas. A memória local do módulo CPU ECB VME contém 4 páginas; duas compõem a memória dual-port VME-barramento interno ao módulo; as outras duas são acessadas pela CPU Z80 exclusivamente através do barramento interno.

A interface com o VME contém um mestre e um escravo. A interface se comporta como mestre durante o atendimento das interrupções geradas por outros mestres ou escravos no VME. Durante a geração de interrupções no VME pelo PE/S e nos acessos à memória dual-port por algum mestre no VME, a interface se comporta como escravo. A interface com o barramento ECB torna disponíveis no conector P2 do módulo todos os sinais de

controle do Z80 (devidamente reforçados). Isto permite a instalação de outro mestre no ECB (outra CPU ou controlador de ADM) e a requisição de interrupções não-mascaráveis e mascaráveis pelos periféricos.

4.2.6 Módulo de Memória VME-VMX

Este módulo pode conter 256 Kbyte de memória RAM estática (6164) ou 512 Kbyte de EPROM (27128) ou combinações de RAM e EPROM até um limite de 512 Kbyte. O módulo possui interfaces para os barramentos VME (conector P1) e VMX (conector P2). As duas interfaces são do tipo escravo e permitem acessos em 8 ou 16 bits – a largura da palavra é 16 bits.

O módulo pode operar somente com a interface VME ou somente com a interface VMX ou com ambas simultaneamente. Um árbitro localizado no módulo decide qual interface será habilitada em caso de conflito (acesso simultâneo pelo VME e VMX). O modo de operação, isto é, dual-port ou mono-port, é selecionado por *jumpers*. No modo dual-port, o módulo pode ser mapeado em áreas diferentes no VME e VMX de tal forma que uma mesma palavra pode ser referenciada em dois endereços diferentes, dependendo da porta por onde ocorrer o acesso.

4.3 Exemplos de Aplicações do MSC

Nesta seção discutimos dois exemplos de aplicação do MSC. A distribuição dos componentes funcionais em três placas de circuito impresso, permite a construção de máquinas com diferentes capacidades de processamento sem aumento proporcional no custo do equipamento. Na discussão que segue, apresentamos a arquitetura e a implementação de dois sistemas de controle com o MSC.

4.3.1 Controlador de Robô

A arquitetura do controlador de robô que discutiremos a seguir é semelhante àquela da figura 4.2, com a adição de mais cinco processadores de E/S. A especificação funcional do controlador é para um robô articulado com seis graus de liberdade (seis juntas) empregando os módulos do MSC. Este robô deve poder ser programado em linguagem de alto nível concorrentemente com a execução de movimentos pelo braço.

O controlador possui um processador (68000 e processador de ponto flutuante) para o cálculo de trajetórias, um processador (68000) para a programação do robô, um processador de E/S para cada junta e um módulo de memória global para a comunicação entre os processadores. Cada processador (68000) possui a sua memória local que contém o código e os dados dos programas que ele executa. A figura 4.5 mostra o controlador e as interconexões entre os processadores de junta e as juntas do robô.

O processador dedicado à programação (processador de programação) executa o interpretador da linguagem de programação de robô; o interpretador aceita os comandos de alto nível digitados pelo usuário e gera os códigos internos ao controlador que descrevem as operações e as posições relacionadas aos comandos. O processador de programação oferece ao usuário uma interface bastante amigável e com recursos gráficos de auxílio à programação. Por exemplo, pode mostrar uma simulação dos movimentos programados na tela de um terminal de vídeo.

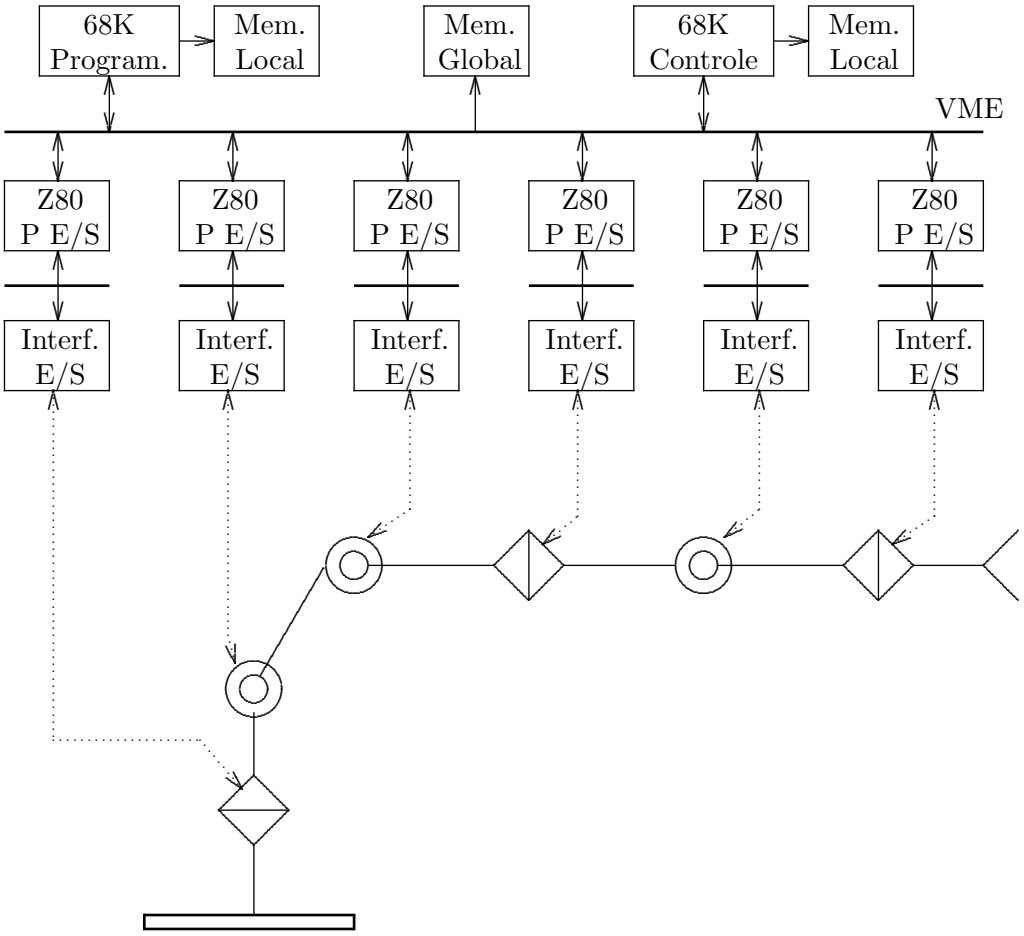

Figura 4.5: Controlador de Robô.

O processador dedicado ao cálculo de trajetórias (processador de trajetória) executa os comandos de movimentação conforme o programado pelo usuário. Estes comandos consistem de uma *operação* e da *posição* associada à operação. Uma matriz de transformação [PAUL81] relaciona o sistema de coordenadas da garra com o sistema de coordenadas da base fixa do robô, descrevendo a posição e orientação do braço e da garra. A partir dos valores contidos na matriz de transformação, os novos ângulos das juntas que satisfazem ao próximo comando a ser disparado são calculados e transferidos aos processadores que controlam as juntas (*set points*). O cálculo dos *set points* envolve a avaliação de funções trigonométricas que deve ser efetuada em números no formato de ponto flutuante para que a precisão dos movimentos do braço seja mantida. O processador de ponto flutuante (MC 68881) permite a execução destes cálculos em tempos muito curtos (`sen(x)` em 5 microssegundos).

Além do cálculo de trajetórias, este processador é o responsável pela sincronização do braço com o meio ambiente onde o robô opera, seja através de sensores instalados nos processadores das juntas, seja através de comunicação (via linha serial) com o controlador da célula de manufatura ou outro sistema de controle de área onde o robô está instalado.

Os processadores das juntas executam os algoritmos de controle de posição e velocidade das juntas, obedecendo aos *set points* recebidos do processador de trajetórias. Estes algo-

ritmos são do tipo PID[3] com tempos de amostragem da ordem de 30 milissegundos. Além dos algoritmos de controle propriamente dito, os processadores de juntas devem executar o algoritmo de interpolação que gera os valores intermediários aos set points fornecidos pelo processador de trajetória. É responsabilidade de alguns dos processadores das juntas a monitoração do meio ambiente através de sensores de proximidade, luz ou tato, por exemplo.

4.3.2 Controlador de Célula Flexível de Manufatura

Nesta seção apresentamos a implementação de um controlador de Célula Flexível de Manufatura (veja a figura 4.6) que emprega os módulos do MSC para obter paralelismo de processamento nas diferentes funções associadas ao controle e monitoração da célula. Nesta discussão, consideramos uma célula com um robô para manipulação de peças e duas máquinas-ferramenta com controle numérico. Esta célula pertence a uma linha de produção e é alimentada por uma correia transportadora. A linha de produção é completamente automatizada e seus componentes são interligados por uma rede local.

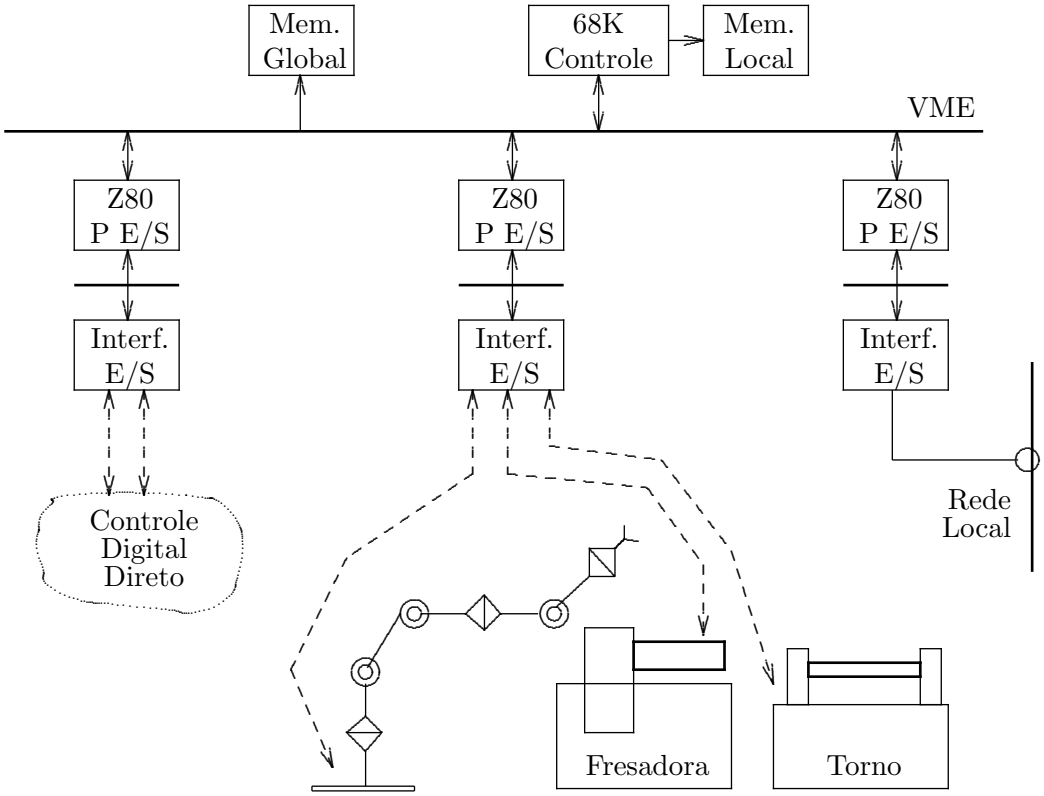

Figura 4.6: Controlador de Célula de Manufatura.

O controlador é composto por um processador (68000) que executa as funções de controle de alto nível, um processador de E/S (Z80) para interface com o meio ambiente, um PE/S (Z80) para o controle dos componentes da célula e um PE/S (Z80) para a interface com a rede local.

O processador (68000) dedicado ao controle da célula (processador de controle) executa um interpretador de Redes de Petri ou de *Mark Flow Graph* [MASU80] que descreve a

[3]Controle Proporcional, Integral e Derivativo.

sequência de operações que deve ser executada sobre a peça pelas máquinas. A Rede de Petri ou o *Mark Flow Graph* de cada peça deve conter uma lógica de intertravamento que mantenha os componentes da célula sempre numa condição segura (evitando colisões do braço do robô com as máquinas, por exemplo).

Além do controle local sobre a célula, o processador de controle deve manter os demais membros da linha de produção e os níveis superiores da hierarquia de controle informados sobre o seu estado de funcionamento. Estas informações permitem a sincronização da célula à linha e a tomada das medidas cabíveis em caso de pane ou acidente. O processador de controle também é o responsável pela decodificação dos comandos recebidos dos níveis superiores da hierarquia e pela transformação destes comandos em ordens específicas aos componentes da célula. Além disso, o processador de controle deve manter atualizadas as bases de dados locais e remotas que contém informações sobre o nível de produção e o estado das máquinas.

O processador de E/S responsável pela interface com o meio ambiente deve fazer um pré-tratamento das informações colhidas pelos sensores e sinalizar ao processador de controle a ocorrência dos eventos relevantes. Um evento importante é a chegada de novas peças na correia transportadora, por exemplo. Este PE/S também é o responsável pela atuação direta sobre dispositivos especiais que por ventura façam parte da célula.

O PE/S dedicado ao controle do robô e das máquinas com controle numérico é o responsável pela transmissão das ordens geradas pelo processador de controle às maquinas e ao robô, via interface série. As mudanças de estado de funcionamento das máquinas (peça pronta, carregando peça, etc...) e as panes devem ser sinalizadas ao processador de controle. Se as máquinas forem programáveis remotamente, este processador efetua a carga dos programas de fabricação de peças, obedecendo aos comandos do processador de controle.

O PE/S com a interface de rede local [APPE87] controla o fluxo de mensagens geradas pelo controlador ou endereçadas a ele. Este processador oferece uma interface de alto nível ao processador de controle para a transmissão e recepção de mensagens através da rede local.

Capítulo 5

O Núcleo de Tempo-Real do MSC

A especificação funcional de um sistema pode ser encarada como um conjunto de exigências e limitações que devem ser cumpridas para que o produto resultante da implementação esteja de acordo com os requisitos. O projeto de um sistema computacional consiste da solução do problema proposto na sua especificação funcional. Esta tarefa é simplificada se o problema proposto for reduzido a um conjunto de subproblemas.

A solução dos subproblemas envolve a escolha de mecanismos e políticas que satisfaçam aos requisitos de projeto. Estas escolhas implicam em compromissos entre os vários aspectos envolvidos. Um exemplo clássico é aquele do relacionamento entre tempo e espaço, isto é, tempo de processamento e espaço de memória: a redução de um implica no aumento do outro. Em geral, a opção entre tempo de processamento e espaço de memória é facilitada pelos requisitos funcionais do sistema. Outras opções envolvem fatores com ponderabilidade mais sutil que tempo e espaço.

O núcleo de um sistema operacional para o MSC foi projetado para aplicações em controle industrial e automação da manufatura. Alguns de seus requisitos são:

- deve suportar a programação de sistemas de tempo-real,
- deve oferecer suporte à programação de aplicativos distribuídos em mais de um processador,
- deve oferecer ao programador um conjunto de primitivas que permitam a geração de programas eficientes e rápidos e
- as primitivas devem simplificar a tarefa do programador (e não o contrário).

As próximas seções discutem as soluções adotadas para que os requisitos funcionais e de desempenho fossem atendidos.

Algumas das premissas consideradas na fase inicial do projeto e que tiveram influência no resultado foram:

- os computadores para aplicações em controle são usualmente dedicados a um conjunto fixo de tarefas de supervisão e/ou controle;
- a interação com o meio ambiente pode envolver a manipulação direta de dispositivos de Entrada/Saída pelos programas de aplicação;
- os programas executados nestes computadores, uma vez depurados, permanecem em execução contínua por longos períodos de tempo;

- a programação de tempo-real impõe restrições sérias sobre a eficiência dos programas.

O NÚCLEO do MSC oferece ao projetista de sistemas de controle um conjunto de mecanismos que permitem a construção de sistemas que atendem às condições do parágrafo anterior. O conjunto de primitivas do NÚCLEO implementa os mecanismos básicos para o controle da execução dos programas. Estes mecanismos são simples e permitem ao projetista a escolha das políticas mais adequadas à sua aplicação para o gerenciamento dos recursos do sistema — processadores e periféricos.

A filosofia minimalista aplicada ao projeto do NÚCLEO é suficientemente flexível para permitir que as políticas de gerenciamento dos recursos do sistema sejam tão simples ou tão complexas quanto o necessário. A implementação de políticas simples a partir de mecanismos complexos não é trivial. Se as necessidades de uma determinada aplicação puderem ser satisfeitas diretamente com mecanismos e políticas simples, o seu uso não implica em nenhum ônus provocado por "peso morto". A recíproca não é verdadeira, no caso de mecanismos complexos sub-utilizados.

5.1 Organização do Núcleo

O NÚCLEO possui características de multiprocessador e de sistema distribuído. Por razões de eficiência, não há compartilhamento de código na memória global. Cada processador executa os programas armazenados na sua memória local de forma mais ou menos independente do que ocorre nos outros processadores. Esta é uma característica de sistema distribuído. A interação entre os programas acontece através da memória global, pelo compartilhamento de dados entre os programas nos vários processadores. Esta é uma característica de multiprocessador.

Cada processador executa uma cópia privativa do código do NÚCLEO, armazenada em sua memória local. O NÚCLEO oferece serviços que permitem a interação entre os programas, de tal forma que o resultado da computação num processador pode interferir de uma maneira controlada nos resultados de computações nos outros processadores do sistema. Existem duas possibilidades para a interação (sincronismo e comunicação) entre dois programas em execução no MSC: (1) os dois programas residem na memória local de um processador, ou (2) os dois programas residem na memória local de diferentes processadores. No primeiro caso, a interação entre os programas é semelhante à que ocorre num sistema multitarefa convencional. No segundo caso, os dois processadores envolvidos devem trocar informações através da memória global via barramento VME.

Do ponto de vista da eficiência, é interessante que as interações do primeiro tipo ocorram completamente na memória local ao processador onde os dois programas residem. O custo dos acessos ao VME é maior, na média, que o dos acessos locais devido à competição entre os processadores pela posse do barramento (além do ciclo normal para escrita ou leitura pode ser necessário um ciclo de arbitragem).

O capítulo anterior discutiu duas características interessantes da arquitetura do MSC: o árbitro do barramento VME, que permite acessos à memória global em exclusão mútua, e o subsistema de interrupções do 68000, que possibilita a comunicação de baixo nível entre os processadores (192 eventos distintos podem ser codificados no vetor de interrupções). Isto viabiliza a implementação eficiente de mecanismos de sincronização e comunicação entre os programas que vão ser executados pelos processadores do MSC.

5.2 Processos

Uma das abstrações fundamentais oferecidas pelos sistemas multiprogramados é aquela de *processo*. Um programa é um conjunto de instruções na memória de um processador, enquanto que um processo é um trecho de programa que está sendo executado pelo processador. Pode-se dizer ainda que o processo é o meio lógico e o processador o meio físico necessários para a execução de um programa. Neste e nos próximos capítulos, os termos *tarefa* e *processo* têm o mesmo significado, isto é, um programa em execução. *Processo* é usualmente associado a sistemas operacionais multiusuário enquanto que *tarefa* é associado a sistemas operacionais de tempo-real ou sistemas multitarefa.

A correção de um programa sequencial não é afetada se ocorrerem pausas no processamento das suas instruções e, nestas pausas, uma quantidade suficiente de informações for salva, de tal forma que o estado de execução do programa possa ser reconstruído posteriormente. Obviamente, os dados usados pelo programa devem permanecer inalterados durante as pausas. Esta discussão baseia-se na suposição de que as pausas acontecem somente entre a execução de duas instruções consecutivas, isto é, a execução das instruções é atômica. O MC 68000 não suporta o truncamento da execução de uma instrução.

As informações salvas nas pausas da execução de um processo compõem o seu *contexto*. Quando um processo tem sua execução paralizada momentaneamente, seu contexto é trocado pelo contexto de outro processo que será executado pelo processador até a próxima troca de contexto. O contexto de um processo é mantido num *descritor de processo*. O contexto consiste dos valores do contador de programa, apontador de pilha, registradores de trabalho e outras informações. Esta definição de contexto de execução pressupõe que a linguagem de programação empregada para escrever os programas mantenha as variáveis locais a cada procedimento alocadas na pilha (C ou Pascal, por exemplo) e que cada processo possua uma pilha de seu uso exclusivo.

O contexto de um processo é composto por um conjunto de informações voláteis e não-voláteis. A parte volátil do descritor contém as informações que se alteram à medida que o processador executa o código do processo (contador de programa, apontador de pilha e registradores de trabalho). A parte não-volátil contém informações que permitem associar o processo a um programa (endereço do início do código e base da sua pilha) e outras informações, tais como o nome do processo. Na parte volátil, algumas das informações se alteram mais rapidamente que outras: o contador de programa se altera a cada instrução mas o seu valor só é copiado no descritor em instantes determinados. Por outro lado, a prioridade de um processo é alterada algumas poucas vezes durante toda a sua vida.

Um processo pode estar ativo (vivo) mesmo que o processador não esteja executando seu código. O *estado* de execução de um processo indica o tipo de atividade em que ele está envolvido num dado instante (o estado é um componente do descritor). O processo pode estar *morto* se nenhum programa foi associado ao seu descritor. O processo pode estar *executando* se o seu código está sendo executado pelo processador. O processo que está executando é chamado de *processo corrente*. O processo está *pronto* se está esperando que o processador pare de executar instruções de outro processo (possivelmente, um de maior prioridade) e retome a sua execução. O processo pode estar *esperando* pela ocorrência de algum evento que não dependa diretamente da sua própria dinâmica (a resposta de um periférico, por exemplo).

5.2.1 Processos no Núcleo do MSC

Os processos são identificados univocamente em todo o sistema pelo índice do seu descritor na *tabela de (descritores de) processos*. Quando um processo é criado, o seu identificador[1], isto é, o seu "nome", passa a ser o índice de um elemento não-ocupado da tabela de descritores. A tabela de processos é distribuída: na memória local a cada processador, são preenchidos apenas os descritores dos processos executáveis pelo processador; os restantes permanecem vazios. A tabela de processos possui um dual em memória global que é a tabela de *pseudo-descritores* de processo. Estas duas tabelas tem o mesmo número de elementos e, a cada descritor corresponde um pseudo-descritor de mesmo índice. O descritor de um processo mantém o seu contexto, conforme exposto anteriormente, enquanto que o pseudo-descritor contém o identificador do processador em cuja memória local o processo reside.

A primitiva de criação de processos permite a passagem de um número variável de parâmetros ao processo que está sendo criado. Desta forma, o processo "pai" e seus "filhos" podem todos compartilhar de tantos recursos quantos necessários, desde que os identificadores dos recursos sejam passados como parâmetros aos filhos, na sua criação. O NÚCLEO não impõe nenhuma hierarquia entre processos (o processo pai pode "morrer" sem que isso afete a execução dos seus filhos[2]). Processos que executam na memória local de um processador são chamados de *processos internos* a este processador; processos que executam remotamente, isto é, na memória local de outros processadores são chamados de *processos externos*.

5.3 Modelo Estratificado

O NÚCLEO foi projetado de acordo com o modelo estratificado [DIJK68, COME84]. Este modelo leva, naturalmente, ao projeto de sistemas inerentemente simples e bem estruturados. Cada estrato ou camada implementa uma máquina virtual mais poderosa que aquela da camada imediatamente inferior, acrescendo novas abstrações ao conjunto oferecido pelas camadas sobre as quais é construída. A abstração implementada numa camada é, em geral, composta por uma estrutura de dados e um conjunto de rotinas que permitem a consulta e a alteração do seu conteúdo. Estas rotinas são chamadas de *primitivas* ou *serviços*. As operações sobre as estruturas de dados escondem os detalhes da sua implementação, oferecendo ao programador uma visão clara das abstrações contidas nelas.

A figura 5.1 mostra o modelo estratificado do NÚCLEO; os próximos parágrafos descrevem brevemente as abstrações e os serviços oferecidos pelas camadas. Uma discussão detalhada sobre as primitivas segue no próximo capítulo. A figura 5.1 mostra também uma estimativa do tempo médio de execução não-interruptível do código das primitivas de cada uma das camadas.

A Camada 0 não pertence propriamente ao NÚCLEO mas é sobre ela que este é construído. A abstração deste nível é a execução das instruções de uma forma contínua. Os detalhes internos ao processador ("pre-fetch", arbitragem do barramento, etc...) e do funcionamento dos periféricos (linha serial com interface RS 232, 8 bits por caracter, etc...) ficam "invisíveis" para as camadas superiores. O montador e o compilador permitem a programação em níveis cada vez mais altos, tornando invisíveis a linguagem de máquina e a linguagem de montagem do 68000, respectivamente.

[1] Neste texto, a palavra "identificador" significa o nome lógico de objetos/recursos.

[2] A morte de um processo não provoca a destruição automática dos recursos criados por ele.

		não-interruptível
6	comunicação interprocessos	60%
5	gerenciamento de armazenadores	60%
4	sincronização interprocessos	80%
3	processo	80%
2	operações sobre filas de processos	100%
1	gerenciamento de memória	100%
0	MSC, montador, compilador	

Figura 5.1: Modelo estratificado do NÚCLEO do MSC.

A Camada 1 – gerenciamento de memória – mantém a lista de memória disponível aos processos (a lista livre de memória). Acima deste nível, os processos requisitam áreas de memória para seu uso ("heap") sem conhecer os detalhes envolvidos na alocação e desalocação de blocos de memória.

Acima da Camada 2 – gerenciamento de filas de processos – os descritores de processo podem ser inseridos e retirados de listas encadeadas, filas de prioridade ou filas do tipo FIFO ("First In First Out"). As operações de inserção e retirada escondem a implementação: todas as filas de processos são montadas sobre uma mesma estrutura de dados.

A Camada 3 implementa a abstração de processo. Esta camada esconde a multiplexação de um processador entre os vários processos na sua memória local fazendo com que cada processador torne-se um recurso compartilhado entre os processos. O *escalonador* é o algoritmo que escolhe a qual dos processos prontos para executar será entregue a posse do processador. Ainda, a Camada 3 oferece primitivas para criar e matar processos e para provocar algumas das alterações possíveis no estado de um processo.

Na Camada 4 estão as primitivas que tornam possível a sincronização entre processos. Estas primitivas são `P` e `V` [DIJK68] sobre semáforos estendidos para o ambiente multiprocessador do MSC e a que denominamos *semáforos distribuídos* [HEXS88]. Nesta camada estão ainda as primitivas que permitem que um processo suspenda a sua execução por um intervalo de tempo determinado.

Acima da Camada 5 – gerenciamento de armazenadores ("buffers") – os processos podem requisitar blocos de memória para a formação de conjuntos de armazenadores ("buffer pools"). As primitivas desta camada controlam o uso de armazenadores pelos processos para impedir que toda a memória disponível seja consumida por um processo em detrimento dos demais.

Acima da Camada 6 – comunicação interprocessos – os processos podem enviar e receber mensagens de/para outros processos invocando as primitivas que implementam *caixas postais* ("mailboxes") [HABE76]. Estas primitivas provocam a sincronização entre os produtores e os consumidores de mensagens, impedindo a retirada de uma caixa vazia ou o envio através de uma caixa cheia.

Sobre a Camada 6 podem ser construídas novas camadas, tais como um servidor de arquivos, ou, programas de aplicação. As Camadas 7 e seguintes podem usar diretamente os serviços oferecidos pelas camadas 1, 3, 4, 5 e 6. As operações sobre filas de processos são de uso exclusivo das primitivas do NÚCLEO.

5.4 Transparência de Multiprocessamento

O NÚCLEO suporta a programação de sistemas de alto desempenho pela exploração de paralelismo real na execução de programas. A programação concorrente impõe alguns requisitos ao núcleo de um sistema operacional: ele deve possibilitar a sincronização e a comunicação entre os programas que suporta. A programação de sistemas de tempo-real impõe, como um requisito, que o suporte à programação concorrente seja eficiente e rápido. O NÚCLEO atende a um requisito adicional: a programação concorrente no MSC – um multiprocessador – oferece um grau de dificuldade não maior que a programação de sistemas multitarefa convencionais [HEXS88].

A *Transparência de Multiprocessamento* é a característica do NÚCLEO que satisfaz o requisito adicional do parágrafo anterior. Ela permite que todas as operações, exceto a criação, sobre os objetos ou recursos suportados diretamente pelo NÚCLEO referenciem apenas os seus identificadores (os seus "nomes"). As primitivas escondem do programador a localização do objeto sobre o qual operam: os objetos podem ou não residir na memória local de um mesmo processador, as primitivas são invocadas com a mesma sintaxe. A localização de um objeto é explicitada somente na sua criação.

O custo das operações *locais* é menor que o das *remotas*. Numa operação local, o código da primitiva é executado pelo processo que a invocou. Numa operação remota, o código da primitiva é executado pelo núcleo do processador onde o "alvo" da primitiva reside. Note que o custo de uma operação remota depende de fatores como, por exemplo, o tráfego no barramento global e a carga suportada pelo processador alvo. A próxima seção discute a questão do "tamanho" dos processos e sua relação com o desempenho do sistema.

Um programa distribuído é executado no MSC como um conjunto de processos. As interações entre eles são suportadas pelos mecanismos de sincronização e comunicação oferecidos pelo NÚCLEO. Um processo em cada processador, chamado de *Configurador*, tem a função de "criar" os processos e os recursos necessários à sua execução. O Configuradores determinam que processos serão criados em qual processador e quais dos recursos serão compartilhados por quais processos. A existência do Configurador, que pode ser uma parte de `main()`[3], não é uma imposição do núcleo embora torne os programas para o MSC portáteis: se todos os recursos compartilhados pelos processos que compõem um programa forem criados pelos Configuradores, um aplicativo pode ser transportado de um sistema com `n` processadores para outro com `m` processadores, $\mathtt{m} \neq \mathtt{n}$, necessitando de alterações apenas e tão somente no código dos Configuradores (re-especificando a localização dos recursos). Os demais componentes do programa não necessitam ser recompilados. Isto só é possível graças à Transparência de Multiprocessamento.

5.5 Escalonamento

O *escalonador* é o algoritmo que escolhe, dentre os prontos, um processo para ter suas instruções executadas pelo processador. O escalonador pode ser invocado explicitamente pelos processos ou implicitamente, em função de alguma mudança no estado do sistema. Esta escolha é baseada no mecanismo de prioridades, que faz parte do escalonador.

Os processos prontos para execução são mantidos numa fila ordenada por prioridade – a *fila de pronto*. O início, ou a cabeça, desta fila contém sempre o processo de maior

[3] `main()` é o procedimento (subrotina) que inicia um programa em C.

prioridade. A função básica do escalonador é, quando invocado, decidir se a prioridade do processo corrente é maior que aquela do processo no início da fila. Se o processo na cabeça da fila tiver prioridade maior, o contexto do processo que estava executando é salvo no seu próprio descritor, o contexto do novo processo é recomposto e o processador (re)inicia a execução das instruções do novo.

A fila de pronto pode ser implementada como uma única fila. Neste caso, os processos são inseridos na posição adequada, de tal forma que sua prioridade seja maior que a de seu sucessor e menor ou igual que a de todos os seus antecessores, no momento da inserção (numa mesma prioridade a ordenação é FIFO). Noutra implementação, a fila de pronto é, na verdade, um conjunto de filas, uma para cada nível de prioridade (usualmente, de quatro a seis níveis). A fila de cada prioridade é FIFO e o processo de maior prioridade está na cabeça da fila de maior prioridade não-vazia. A fila de pronto do NÚCLEO é do primeiro tipo.

O escalonador pode ser *preemptivo* ou *não-preemptivo*. No escalonamento preemptivo, o processador está sempre executando o processo de maior prioridade, dentre os prontos. Se, devido a alguma mudança nas condições internas (estado da computação) ou externas (periféricos) ao sistema, um processo com prioridade maior que a do processo corrente tornar-se apto a executar, o escalonador é invocado para forçar a troca de contexto, voltando o processo de menor prioridade para a fila de pronto (de onde sairá quando sua prioridade tornar-se novamente a maior). No escalonamento não-preemptivo, o escalonador não é invocado automaticamente quando um processo de maior prioridade que a do corrente torna-se apto a executar. O escalonador é invocado para efetuar trocas de contexto apenas em pontos explicitamente definidos no código dos programas.

A programação de sistemas de tempo-real impõe o seguinte requisito, além daqueles usuais em programação concorrente: o processamento relativo a um evento externo ao sistema deve ser concluído antes da ocorrência de outro evento ou antes que expire um certo intervalo de tempo. A *latência* do sistema com relação a um dado evento é o intervalo decorrido entre a percepção pelo sistema da sua ocorrência e a conclusão do processamento a ele associado [ABBO84]. O comportamento do sistema de tempo-real pode ser especificado e avaliado em termos da *latência máxima permitida* para alguns dos eventos tratáveis pelo sistema. A latência máxima permitida é um *critério dinâmico* para o projeto de sistemas e se refere a cada evento, isoladamente.

Além do critério dinâmico, pode ser definido um *critério estático* de projeto. Este critério determina se um sistema é capaz de executar em tempo-real por longos períodos de tempo. O *tempo total de execução* é a somatória dos tempos de execução seqüencial de todos os processos de um sistema. O critério estático é atendido se: (1) a freqüência com que os eventos ocorrem é tal que seu tratamento consome uma percentagem relativamente pequena do tempo total de execução, e (2) o critério dinâmico de todos os eventos é atendido. Se o critério estático não for atendido, um único processador é incapaz de executar, em tempo-real, o tratamento de todos os eventos relevantes ao sistema.

Chamamos de *tempo de execução contínua* de um processo ao intervalo em que ele pode executar sem ser interrompido por um reescalonamento. No escalonamento não-preemptivo, o programador deve projetar o sistema para que o tempo de execução contínua de cada um dos processos seja arbitrariamente menor que a menor latência máxima permitida de todos os eventos tratáveis pelo sistema (a determinação deste "arbitrariamente menor" não é uma tarefa trivial). Além disso, a latência enfrentada por um processo de alta prioridade depende do tempo de processamento de *todos* os processos, de quaisquer prioridades. Processos que fazem um tratamento de dados complexo e demorado devem

ser "quebrados" em (sub)processos mais rápidos para que o critério da latência máxima seja atendido. Esta redução forçada no tamanho dos processos implica em sobrecarga adicional para a transmissão de resultados parciais entre os (sub)processos ou para o salvamento destes resultados entre duas ativações consecutivas de um mesmo processo, se este é interrompido por reescalonamento.

Uma desvantagem do escalonamento preemptivo é o custo incorrido pelo número potencialmente grande de trocas de contexto que pode ocorrer em determinadas circunstâncias. Não é difícil de se imaginar um cenário onde um sistema esteja sobrecarregado pelo atendimento de eventos externos, o que provoca muitas trocas de contexto. Isso retarda o progresso no tratamento dos eventos, resultando em maiores latências e piora no desempenho. A escolha das prioridades e da periodicidade de ativação (tempo de ciclo) dos processos que devem executar ciclicamente não é tarefa simples. Podem ocorrer combinações patológicas destas grandezas onde, por exemplo, o tempo de execução contínua de um processo de alta prioridade é maior que o tempo de ciclo de um de baixa prioridade.

Em média, o escalonamento não-preemptivo previlegia os processos de baixa prioridade em detrimento dos de alta. O escalonamento preemptivo possibilita um grau de "atendimento" pelo processador aos eventos e processos que é proporcional às suas prioridades. No escalonamento preemptivo, a computação progride em função da ocorrência dos eventos ("event driven") ao passo que no não-preemptivo o atendimento aos eventos é retardado, em média, pela metade do tempo de execução contínua dos processos ("task driven").

5.5.1 Escalonamento no Núcleo do MSC

A escolha do tipo de escalonamento pode depender do tipo de sistema controlado. Sistemas onde a latência máxima permitida para algum evento é tão pequena que impediria a programação de processos com mais que algumas poucas instruções, são bons candidatos ao escalonamento preemptivo. Sistemas "bem comportados" onde os eventos acontecem com freqüências relativamente baixas e razoavelmente constantes são bons candidatos ao não-preemptivo. No NÚCLEO, esta escolha é facultada ao programador através de compilação condicional.

Do ponto de vista do escalonador, o MSC é um sistema distribuído onde cada processador executa um conjunto de programas de forma autônoma. A memória global é o meio físico de comunicação entre os processos nos vários processadores. Quando uma primitiva é invocada para alterar o estado de execução de um processo noutro processador, o que ocorre é uma "chamada remota de procedimento"[4]. O processador onde o processo "alvo" reside atende a requisição de serviço originada pelo processo que invocou a primitiva (a primitiva é invocada num processador e executada noutro). Sua ação pode ou não resultar num reescalonamento, dependendo do tipo de primitiva e do estado dos processos no processador alvo. Se ocorrer um reescalonamento, o escalonador percebe apenas a mudança no estado dos processos internos mas ignora a sua causa.

5.6 Sincronização Interprocessos

Uma das motivações para o uso de programação concorrente é o ganho de velocidade advindo da execução paralela de vários processos cooperando entre si. A cooperação entre

[4]O funcionamento das "chamadas remotas de procedimento" é descrito adiante, através de exemplo.

processos concorrentes acontece através de comunicação e sincronização. A comunicação é o que permite que a execução de um processo influencie a execução de outro. A comunicação interprocessos pode ser realizada pelo compartilhamento de variáveis ou por troca de mensagens.

Para que ocorra a comunicação, um processo deve executar uma ação que seja detectada pelo outro, como a alteração do valor de uma variável compartilhada ou o envio de uma mensagem. A execução dos processos progride com velocidades variáveis e imprevisíveis. Isso torna necessário o uso de um mecanismo de sincronização para garantir que os eventos "executa a ação" e "percebe a ação" aconteçam nesta ordem. Assim, a sincronização interprocessos pode ser encarada como uma restrição no ordenamento de eventos [ANDR83].

A execução de uma instrução pelo processador (68000) é uma *operação atômica*, isto é, indivisível. A execução concorrente de um grupo de processos pode ser encarada como uma seqüência de operações atômicas. Esta seqüência é formada pelo entrelaçamento das seqüências de operações atômicas de cada um dos processos, tomado individualmente. Uma *seção crítica* é um segmento de código que deve ser executado como uma única instrução. Não pode ocorrer entrelaçamento durante a execução de uma seção crítica: seu código deve ser executado em *exclusão mútua* pelos processos.

Um dos problemas básicos que pode ser resolvido com sincronização interprocessos é o da exclusão mútua. Se vários processos desejam executar simultaneamente o código de uma seção crítica, um mecanismo de sincronização pode garantir que no máximo um processo o faça a cada instante; os demais têm sua execução bloqueada enquanto esperam pela saída do primeiro da seção crítica (quando então o acesso será franqueado a outro processo). Os acessos às seções críticas são os eventos ordenados pelo mecanismo de sincronização.

O outro problema básico é o dos pares *produtor/consumidor*. Neste caso, um processo produz um determinado objeto e deposita-o num armazenador de tamanho limitado. Outro processo consome os objetos que retira do armazenador. A ordenação dos eventos consiste em garantir que os objetos sejam consumidos somente depois de produzidos. Este problema envolve outro tipo de sincronismo, dissemelhante da exclusão mútua, que é o sincronismo condicional ("condition synchronization"). No caso do produtor/consumidor, o estado do armazenador (cheio ou vazio) é uma condição pela qual o produtor e o consumidor devem se sincronizar. O mecanismo de sincronização deve impedir que o produtor deposite objetos no armazenador, quando ele está cheio, e que o consumidor retire objetos do armazenador, quando vazio.

5.6.1 Semáforos

O semáforo é um mecanismo de sincronização com características que o tornam muito interessante para aplicações em programação concorrente. Um *semáforo* é uma variável inteira não-negativa sobre a qual são definidas duas operações: `P` e `V` [DIJK68]. Dado um semáforo `s`, a execução de `P(s)` por um processo bloqueia sua execução até que $s > 0$, quando ele então decrementa `s` de 1 (o teste e o decremento são executados atomicamente). `V(s)` incrementa o semáforo de 1 (atomicamente). As primitivas `P` e `V` também são conhecidas pelos nomes `wait` e `signal`, respectivamente. Um semáforo cujo campo de valor pode assumir apenas os valores `0` e `1` é chamado de *semáforo binário*. Se o campo de valor puder conter qualquer valor inteiro não-negativo, o semáforo é do tipo *contador*.

A implementação de um semáforo é *justa* se garante que um processo que executou

`P(s)` não ficará bloqueado eternamente se a operação `V(s)` for executada com freqüência. Isso é facilmente obtido se os processos forem liberados na mesma ordem em que foram bloqueados. Quando um processo executa `P(s)` e fica bloqueado esperando pela execução, por outro processo, do `V(s)` correspondente, ele entrega voluntariamente a posse do processador. O escalonador escolhe então um outro processo, dentre aqueles que estão prontos para executar. Isso elimina a necessidade de sincronização com *espera ocupada*[5] (*busy waiting*), aumentando o compartilhamento do processador entre os processos.

Uma solução com semáforos para o problema da exclusão mútua é mostrada a seguir. Em cada processo, a seção crítica é precedida por uma operação `P` e seguida por uma operação `V`. Todas as seções críticas mutuamente exclusivas usam o mesmo semáforo, inicializado em `1`. A solução para dois processos é mostrada na figura 5.2. Note que os protocolos de entrada e saída da seção crítica são simples e simétricos. Esta solução garante a exclusão mútua e a ausência de impasse (*deadlock*) [ANDR83].

```
struct semaforo mutex;

p1()
{
      while(TRUE) {
            P(mutex);                    /* protocolo de entrada */
            seção crítica1;
            V(mutex);                    /* protocolo de saída */
            seção não-crítica1;
      }
}

p2()
{
      while(TRUE) {
            P(mutex);                    /* protocolo de entrada */
            seção crítica2;
            V(mutex);                    /* protocolo de saída */
            seção não-crítica2;
      }
}
```

Figura 5.2: Exclusão mútua com semáforos.

O semáforo é um mecanismo de sincronização versátil e potente. Os problemas clássicos de sincronização *exclusão mútua*, *leitores e escritores*, *armazenador limitado* e o *jantar dos filósofos* podem todos ser resolvidos com semáforos [PETE83, BARI82]. Além do seu poder semântico, esta primitiva pode ser implementada de uma forma simples e eficiente. Contra o emprego de semáforos, argumenta-se que seu uso não favorece à produção de código bem estruturado e que pequenos enganos do programador podem levar impasses de difícil detecção.

O uso de construções estruturadas, como *monitores* [HOAR74] e seus descendentes, em linguagens que trazem a sincronização embutida em construções de alto nível como mo-

[5] *Espera cupada* é um mecanismo (ineficiente) de sincronização, onde uma dada condição é continuamente testada. Enquanto a condição for "falsa", o processador não executa nenhum "trabalho útil".

nitores ou módulos (em Modula [WIRT77]), oferece maiores facilidades de programação. O sincronismo entre os processos é controlado pelo código gerado pelo compilador. Por outro lado, o semáforo, por ser uma construção de mais baixo nível, permite ao programador um controle mais estrito sobre a execução dos processos. Além disso, as construções mencionadas neste parágrafo podem ser implementadas com semáforos.

5.6.2 Semáforos Distribuídos

Como discutido na seção anterior, o semáforo é um mecanismo potente e flexível para a sincronização interprocessos, permitindo ao programador a implementação das políticas que julgar conveniente para o controle da execução dos processos. O NÚCLEO oferece um conjunto de operações sobre semáforos que escondem a sua localização do programador, de tal forma que os programas de aplicação podem ser escritos como o seriam para executar em ambientes multitarefa convencionais. A sintaxe das operações sobre semáforos é a mesma independentemente de a operação envolver processos executando num único ou em vários processadores.

Os semáforos do NÚCLEO são de dois "tipos": *locais* e *globais*. Os *semáforos locais* são usados para a sincronização de processos executando na memória local de um mesmo processador. Os *semáforos globais* são empregados na sincronização de processos residentes em processadores distintos. A diferenciação entre semáforos locais e globais é aparente apenas na sua criação, quando o programador especifica o seu tipo. Todas as outras operações sobre semáforos referenciam apenas seus identificadores. Obviamente, processos residentes na memória local a um mesmo processador podem sincronizar-se através de semáforos globais, arcando com o custo mais elevado destas operações.

Os próximos parágrafos discutem um exemplo de criação e utilização de semáforos por processos executando em processadores distintos. A figura 5.3 mostra a criação de dois semáforos, um local e outro global e sua utilização por três processos, dois deles executando num mesmo processador e o terceiro, noutro [HEXS88]. Os parâmetros da primitiva `criasem()` são a localização do semáforo e o seu valor inicial. A primitiva que cria um processo (`cria()`) recebe como parâmetros a localização do código do processo (criação local ou remota), o endereço do código (se criação local) ou o nome do programa[6] (criação remota), prioridade inicial, tamanho de pilha, o número de parâmetros de ativação do processo e os parâmetros, se for o caso. A execução dos processos inicia somente após a sua "ativação", por `ativa()`.

Os semáforos globais são implementados de forma distribuída. Uma parte do descritor de semáforo global reside em memória global e a outra parte é distribuída na memória local a cada processador. A *parte global* consiste do valor do semáforo e de uma fila de pseudo-descritores de processo. A *parte local* consiste de uma fila de descritores de processo. Quando é realizada uma operação num semáforo global, tanto a parte global quanto a parte local (onde reside o processo que está ou ficará bloqueado) são consultadas e/ou atualizadas.

A figura 5.4 mostra o estado de parte das estruturas de dados (filas) do NÚCLEO durante a execução do exemplo da figura 5.3. `sem2` é a parte local de `s2`, `gsem2` é a sua parte global. `p1`, `p2` e `p3` são os descritores de `pid1`, `pid2` e `pid3`, respectivamente. `CPU1` corresponde ao pseudo-descritor de `pid1`. Quando um processo executa `P(sem)` e deve ficar bloqueado, seu descritor é enfileirado na parte local do semáforo e seu pseudo-descritor na parte global. Quando um semáforo global é sinalizado, o primeiro pseudo-descritor é retirado da fila de

[6] Uma cadeia de até sete caracteres.

```
/* Configurador executando na CPU1 */
 ⋮
s1 = criasem(LOCAL,0);
s2 = criasem(GLOBAL,2);
 ⋮
pid1 = cria(CPU1,prog1,...,2,s1,s2);
pid2 = cria(CPU2,"prog2",...,1,s2);
pid3 = cria(CPU1,prog3,...,1,s1);
 ⋮
ativa(pid1);
ativa(pid2);
ativa(pid3);
 ⋮
```

/* na CPU1 */	/* na CPU1 */	/* na CPU2 */
/* proc. pid3 */	/* proc. pid1 */	/* proc. pid2 */
prog3(s1)	prog1(s1,s2)	prog2(s2)
int s1;	**int** s1,s2;	**int** s2;
⋮	⋮	⋮
⋮	P(s2);	⋮
P(s1);	⋮	V(s2);
⋮	V(s1);	⋮
⋮	⋮	⋮

Figura 5.3: Exemplo de uso de semáforos distribuídos.

processos bloqueados naquele semáforo e o processador onde o processo reside é notificado da liberação de um de seus processos por uma interrupção via VME.

A memória local a cada processador contém uma tabela de descritores de semáforos. Os elementos desta tabela com índices [0...NSEMGLB-1] são alocados aos semáforos globais e os restantes, aos locais ([NSEMGLB...NSEM-1]). A memória global contém uma tabela de descritores de semáforos com NSEMGLB elementos. O índice de um semáforo global identifica-o univocamente em todo o sistema. O NÚCLEO determina o tipo do semáforo pelo valor de seu identificador e executa as ações pertinentes em cada caso. Desta forma é que é obtida a *Transparência de Multiprocessamento* nas operações sobre semáforos. Note que uma posição da tabela de semáforos, alocada como local, corresponde a um semáforo diferente em cada processador: um mesmo identificador pode corresponder a vários semáforos locais independentes. Isso não causa nenhum problema porque as operações sobre semáforos locais ocorrem totalmente em memória local.

5.7 Comunicação Interprocessos

A comunicação entre processos cooperantes é o que permite que os resultados da computação efetuada num processo possam interferir na execução de outro. A comunicação pode ocorrer de duas formas: por variáveis compartilhadas ou troca de mensagens. A comunicação por variáveis compartilhadas implica no uso de mecanismos de sincronização para garantir a integridade dos valores e o ordenamento das consultas e atualizações. Na

	Mem. Local	Mem. Global
inicialmente:	`sem2→nil` `sem1→p3`	`gsem2→nil`
pid1 executa P(s2):	`sem2→p1` `sem1→p3`	`gsem2→CPU1`
pid2 executa V(s2):	`sem2→nil` `sem1→p3`	`gsem2→nil`
pid1 executa V(s1):	`sem2→nil` `sem1→nil`	`gsem2→nil`

Figura 5.4: Filas do NÚCLEO durante operações sobre semáforos.

troca de mensagens, a comunicação e a sincronização entre os processos ocorre pelo envio e recepção de mensagens. A sincronização é um efeito colateral da comunicação: uma mensagem só pode ser recebida depois de ter sido enviada.

Um *canal de comunicação* deve ser estabelecido entre a fonte e o destino das mensagens, isto é, entre o processo que envia as mensagens e o que as recebe. A comunicação pode ser do tipo *direta*, se os nomes dos processos fonte e destino deverem ser especificados para que o canal seja estabelecido. A comunicação direta pode ser implementada de uma forma muito simples, mas é também muito pouco flexível. Em relações do tipo *cliente/servidor*, onde um ou mais processos usam os serviços proporcionados por um ou mais servidores, um canal deve ser estabelecido entre cada um dos possíveis pares cliente/servidor.

Outro modo de se estabelecer um canal de comunicação é pelo uso de *nomes globais* ou *caixas postais*. Uma caixa postal pode ser o destino das mensagens enviadas pelos processos e a fonte das mensagens recebidas (qualquer processo pode "depositar" uma mensagem numa caixa postal e qualquer processo pode "retirá-la"). Os clientes enviam as requisições de serviço para uma caixa postal; os servidores recebem as requisições dela. Quando somente um processo pode retirar mensagens de uma caixa postal, ela é chamada de *porta* ("port"). A porta é a solução ideal para relações onde um único servidor deve atender a múltiplos clientes.

Os canais de comunicação podem ser estabelecidos *estática* ou *dinamicamente*. No caso estático, a fonte e o destino das mensagens são determinados em tempo de compilação. No caso dinâmico, os canais de comunicação são estabelecidos e removidos em tempo de execução.

As primitivas que implementam a troca de mensagens podem ser do tipo *bloqueante* ou *não- bloqueante*. Uma primitiva do tipo bloqueante pode suspender temporariamente a execução do processo que a invocou; isto nunca acontece com primitivas não-bloqueantes. Quando as mensagens são enfileiradas entre o envio e a recepção, o envio de mensagens é dito *assíncrono*, isto é, o processo que envia mensagens nunca é bloqueado (pelo menos, enquanto houver espaço na fila de mensagens). O envio assíncrono permite que a execução do processo fonte das mensagens progrida independentemente da execução do processo destino. Neste caso, as mensagens podem conter informações desatualizadas. Se as mensagens não forem enfileiradas, o envio deve ser bloqueante. O processo que envia

uma mensagem fica bloqueado até que ela seja recebida pelo processo destino. Esta forma de comunicação é dita *síncrona*. Na comunicação síncrona, as mensagens sempre contém informações atualizadas[7].

5.7.1 Caixas Postais

No NÚCLEO, uma caixa postal é composta por uma fila de mensagens e dois semáforos. As mensagens têm capacidade para conter um inteiro (32 bits). Um dos semáforos controla a retirada de mensagens da caixa, impedindo que um processo tente receber mensagens de uma caixa vazia (bloqueando-o até que outro processo deposite alguma mensagem na caixa). O outro semáforo, controla o envio, sincronizando o(s) produtore(s) de mensagens ao(s) consumidore(s). Se a velocidade dos produtores é muito maior que a dos consumidores, os produtores devem eventualmente ser bloqueados, para impedir que os produtores avancem infinitamente sobre os consumidores (dessa forma permitindo que os consumidores processem as mensagens pendentes).

A recepção de mensagens é bloqueante. O envio de mensagens pode ser síncrono ou assíncrono, dependendo da capacidade da fila de mensagens. O envio é síncrono se a fila tiver capacidade para uma mensagem apenas. O assincronismo depende da capacidade da fila e da velocidade relativa da execução dos processos que enviam e recebem mensagens. A capacidade da fila de mensagens é especificada na criação de cada caixa postal (elas podem ser criadas e destruídas dinamicamente). Da mesma forma que com os semáforos, a localização de uma caixa postal (em memória local ou global) é explicitada somente na sua criação. Todas as outras operações referenciam apenas os seus identificadores.

O mecanismo de comunicação interprocessos do NÚCLEO permite a implementação de vários paradigmas de sincronização entre processos comunicantes. O sincronismo na troca de mensagens pode ser controlado como descrito no parágrafo anterior. Comunicação através de *rendezvous* [BARI82] pode ser implementada com duas caixas postais com capacidade para uma mensagem e um semáforo: o processo fonte envia uma mensagem e espera pela resposta; o processo destino recebe uma mensagem, processa-a e envia uma resposta ao processo fonte. O semáforo torna o "rendezvous" uma operação atômica. Se a mensagem de resposta não é necessária, um semáforo pode ser usado para o bloqueio do processo fonte até que a mensagem seja processada pelo destino. Um *pipe* [RITC74] pode ser implementado com uma caixa postal entre cada dois processos componentes do "tubo".

Como citado anteriormente, uma mensagem consiste de um inteiro. Para a maioria das aplicações, a sobrecarga incorrida na troca de quantidades "úteis" de dados com mensagens de apenas uma palavra é insuportável. Isto é remediado de uma forma simples: o inteiro da mensagem pode ser um apontador para um armazenador ou para um bloco de memória. No primeiro caso, os processos que necessitam de comunicação entre si criam um conjunto de armazenadores com tamanho adequado (mensagens de tamanho fixo). No segundo caso, os processos requisitam blocos de memória, usam-nos como armazenadores temporários e devolvem-nos à lista de memória livre (mensagens de tamanho variável). O programador pode escolher o tipo de mensagem em função dos requisitos da aplicação (tamanho fixo ou variável). Mensagens de tamanho variável simplificam a programação mas são mais onerosas em tempo de processamento.

[7] Como contrapartida, a comunicação síncrona reduz o grau de paralelismo de execução.

Capítulo 6

A Implementação do Núcleo

Neste capítulo, discutiremos a implementação do NÚCLEO do MSC. Inicialmente, mostraremos uma visão de conjunto das estruturas de dados evidenciando a sua localização: quais residem em memória local e quais em memória global. Feito isso, as estruturas de dados e os serviços de cada uma das camadas são descritos.

6.1 Estruturas de Dados

As estruturas de dados do NÚCLEO do MSC são distribuídas, parte em memória global e parte na memória local de cada um dos processadores. A figura 6.1 mostra, esquematicamente, as estruturas em memória global e na memória local a um dos processadores.

A *lista de memória livre* mantém uma lista encadeada de blocos de memória disponíveis para uso pelos processos. Existe uma lista livre para memória global e uma lista livre na memória local a cada processador. As estruturas `q[ ]`, `gq[ ]` e `t[ ]` suportam as filas de processos. Estas estruturas permitem que as filas de processos (fila de pronto, fila de dormindo, filas dos semáforos) sejam todas montadas sobre um mesmo tipo de estrutura. As listas são duplamante encadeadas e permitem a implementação de filas do tipo FIFO, pilhas e listas ordenadas por prioridade.

As *tabelas de descritores de processo* e a *tabela de pseudodescritores* suportam a abstração de processo. A tabela de descritores mantém o contexto dos processos e a associação de processos a programas. A tabela de pseudodescritores associa os processos aos processadores que os executam. A *tabela de semáforos* mantém o estado dos semáforos. A tabela de semáforos globais é distribuída. Parte da descrição de um semáforo distribuído é mantida em memória global (valor e fila de processos bloqueados). Os descritores dos processos bloqueados num semáforo global são enfileirados na parte local a cada processador. Cada *conjunto de armazenadores* é definido num elemento do vetor de descritores de conjuntos de armazenadores. Existe um vetor na memória global e um na memória local a cada processador. As *caixas postais* são definidas nos elementos dos vetores de descritores de caixas postais. Existe um vetor na memória global e um na memória local a cada processador.

6.2 Camada Um - Gerenciamento de Memória

As primitivas de gerenciamento de memória da Camada Um não controlam o consumo de memória pelos processos. Um processo pode consumir toda a memória disponível do sistema, privando os demais deste recurso. Neste nível, as requisições de memória pelos processos são atendidas ou recusadas em função da disponibilidade de memória livre. O

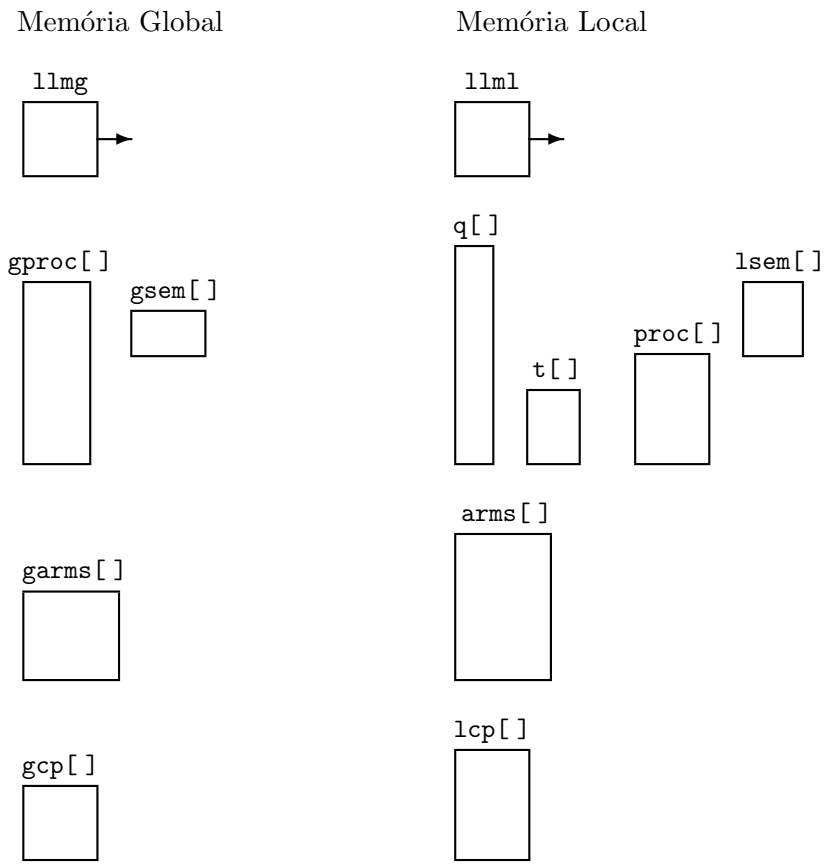

Figura 6.1: Estruturas de dados do NÚCLEO.

algoritmo que mantém a lista de memória é do tipo *first fit* e o número de bytes requisitados ou devolvidos à lista livre deve ser especificado na requisição e na devolução. O custo de uma requisição é variável e depende do estado de fragmentação da memória e do tamanho do bloco requisitado.

6.2.1 Implementação

Cada elemento da lista livre tem o seguinte formato:

```
struct      mbloco  {              /* cabeçalho de bloco de memória */
    struct  mbloco  *mprox;        /* aponta próximo bloco na lista */
    int     tam;                   /* tamanho do bloco, em bytes */
    int     b[tam];                /* bloco de memória com tam bytes */
}
```

A primitiva `aloca()` remove um bloco com `nbytes` da lista livre indicada (global ou local) e retorna um apontador para o seu início. Se a lista livre não contiver um bloco de tamanho suficiente, `aloca()` retorna a constante `SYSERR` (`SYSERR` é um número negativo cuja representação em complemento de 2 é um valor maior que qualquer endereço válido para o 68000).

```
int   *aloca(onde, nbytes)          /* requisita bloco de memória */
```

```
int   onde;                          /* em memória local ou global */
int   nbytes;                        /* tamanho do bloco, em bytes */
```

`desaloca()` testa se o valor de `*bloco` está dentro de limites válidos. Se estiver, o bloco é reinserido na lista livre. Se não estiver, `dealoca()` retorna `SYSERR`. Se o bloco que estiver sendo devolvido for adjacente a outro bloco livre, os dois blocos são agrupados, formando um bloco maior.

```
int   desaloca(onde, bloco, tam)     /* devolve bloco à lista livre */
int   onde;                          /* em memória local ou global */
int   *bloco;                        /* aponta início do bloco */
int   tam;                           /* tamanho do bloco, em bytes */
```

A alocação de área para a pilha de cada processo é obtida com os procedimentos `alocap()` e `dealocap()`. Estes procedimentos são de uso interno ao NÚCLEO, sendo empregados na criação e destruição de processos. A área alocada para pilha é retirada da lista livre de memória local. Enquanto `aloca()` inicia a busca de blocos livres pelos endereços mais baixos, `alocap()` inicia a busca pelos mais altos.

6.3 Camada Dois - Filas de Processos

Quando um processo espera pela ocorrência de um evento, ele o faz numa fila de processos. Cada semáforo possui uma fila de processos; a fila de processos prontos para executar é, como diz o seu nome, uma fila. Todas as filas de processos do NÚCLEO são implementadas num mesmo tipo de estrutura [COME84]. Tanto as filas de descritores de processo em memória local (`q[ ]` e `t[ ]`) quanto as filas de pseudodescritores (`gq[ ]`), em memória global, têm a mesma implementação. O que varia é o número de elementos em cada caso. Discutiremos apenas a implementação da estrutura `q[ ]` e as operações sobre ela, a estrutura e operações sobre `gq[ ]` e `t[ ]` são similares.

O vetor `q[ ]` é composto por elementos com o formato

```
struct    qelmto {       /* elemento dos vetores q[ ], t[ ] e gq[ ] */
     int  chave;         /* chave para inserção (prioridade) */
     int  prox;          /* índice, na estrut.,do próximo elmto. nesta fila */
     int  ant;           /* índice, na estrut.,do elmto. anterior nesta fila */
}
```

Uma cabeça de fila é implementada com dois elementos, um para o início e outro para o final da fila. A tabela de descritores de processo e as estruturas de filas são organizadas de tal forma que o índice do descritor de um dado processo tem o mesmo valor do índice do elemento em `q[ ]` a ele associado. Quando um processo está numa fila, o elemento de `q[ ]` com índice igual ao identificador do processo é que está inserido na fila. As operações sobre a estrutura `q[ ]` são discutidas a seguir. As mesmas operações são disponíveis sobre `gq[ ]`. Sobre `t[ ]`, apenas `enfileira()`, `retira()` e `primeiro()` (as demais são desnecessárias).

```
enfileira(proc,final)          /* insere proc no final da fila indicada */
int   proc;                    /* processo a inserir */
int   final;                   /* índice do elmto. em q[ ], com o final da fila */
```

```
int   retira(proc)              /* retira processo da fila onde estiver incluído */
int   proc;

insere(proc, inicio, chave)     /* insere processo na fila especificada, ... */
                                /* ... no lugar indicado por chave */
int   proc;                     /* processo a inserir */
int   inicio;                   /* índice do elmto. em q[ ], com a início da fila */
int   chave;                    /* chave para inserção (prioridade) */

int   primeiro(inicio)          /* retira primeiro processo da fila indicada */
int   inicio;                   /* índice do elmto. em q[ ], com o início da fila */

int   ultimo(final)             /* retira último processo da fila indicada */
int   final;                    /* índice do elmto. em q[ ], com o final da fila */

int   novafila()                /* inicializa dois elementos consecutivos de ... */
                                /* ... q[ ] como cabeça de fila */
```

A estrutura `q[ ]`, em memória local, possui um elemento para cada processo, dois para cada semáforo local, dois para cada semáforo global e dois para a fila de pronto. A estrutura `gq[ ]`, em memória global, possui um elemento para cada processo, dois para cada semáforo global e dois para cada processador no sistema (*fila de sinais pendentes* – veja na seção 6.5.1). A estrutura `t[ ]`, em memória local, possui um elemento para cada processo mais dois elementos para a fila *dormindo* (seção 6.5.2).

6.4 Camada Três - Processo

Como discutido no capítulo anterior, um descritor de processo é composto de duas partes: o *descritor*, em memória local, contém as informações que associam o processo a um programa; o *pseudodescritor*, em memória global, associa cada processo ao processador onde executa. Os descritores de processo formam a *tabela de processos* (`proc[NPROC]`), os pseudodescritores formam a *tabela global de processos* (`gproc[NPROC]`). O identificador de um processo é o índice de seu descritor na tabela de processos (a cada descritor é associado um pseudodescritor de mesmo índice). O formato dos descritores e pseudodescritores é mostrado a seguir.

```
struct     pelmto {          /* descritor de processo - memória local */
     int   estado;           /* estado do processo */
     int   modo;             /* executa como "user" ou "supervisor" (68000) */
     int   prio;             /* prioridade */
     int   *ctxt;            /* contexto na pilha – regs. do 68000 */
     char  nome[8];          /* nome do processo */
     int   basep,            /* endereço da base da pilha privativa */
     int   tampl,            /* tamanho da pilha */
     int   codigo;           /* endereço do código do processo */
}

struct     gproc {           /* pseudodescritor de processo - memória global */
     int   cpu;              /* identificador da cpu onde executa */
}
```

A tabela de pseudodescritores faz a ligação dos processos aos processadores que os executam. Quando um processo é criado, o seu identificador passa a ser o índice de um pseudodescritor não- ocupado. O descritor de mesmo índice é então preenchido com as informações pertinentes. Num dado processador, apenas os descritores de seus processos internos são preenchidos e usados. Os descritores vazios podem conter processos externos (executando noutro processador) ou podem estar livres (não há nenhum processo associado ao descritor). Em média, a tabela de processos de cada processador contém mais elementos vazios que ocupados. Este "desperdício" de espaço é compensado pela simplicidade e eficiência com que os processos são identificados univocamente em todo o sistema pelo índice do seu descritor.

6.4.1 Grafo de Estados

Os processos no NÚCLEO podem estar em um de oito possíveis estados de execução. Um processo pode estar *morto* se o seu descritor não foi associado a nenhum programa (seu descritor está livre). Um processo está *suspenso* se ele já foi criado mas não pode (ainda) ser executado. Um processo está *pronto* para executar se espera que o escalonador o escolha para execução. O processo *corrente* é aquele cujo código está sendo executado pelo processador. Um processo que executa `P` num semáforo com valor não-positivo fica *bloqueado* até que outro processo execute o `V` que lhe permita seguir executando. Um processo está *temporizado* se executou um `P` com limite de tempo para liberação. Um processo está *dormindo* se suspendeu sua própria execução por um intervalo de tempo limitado.

A figura 6.2 mostra o grafo de estados dos processos. Os vértices do grafo representam os estados e as arestas as transições possíveis entre os estados. A cada aresta são associadas uma ou mais operações (primitivas ou funções internas) que podem provocar a mudança de estado. Embora não esteja representado na figura, um processo pode ser morto em qualquer estado em que se encontre.

Note que o fato de o MSC ser um multiprocessador não foi citado na descrição do grafo de estados. A Transparência de Multiprocessamento permite que o MSC seja programado como um sistema multitarefa convencional. Isto é refletido no seu grafo de estados.

6.4.2 Primitivas e Funções da Camada Três

As primitivas desta camada podem ser divididas em dois grupos. No primeiro grupo estão as primitivas que atuam sobre o estado dos processos. O segundo grupo é composto

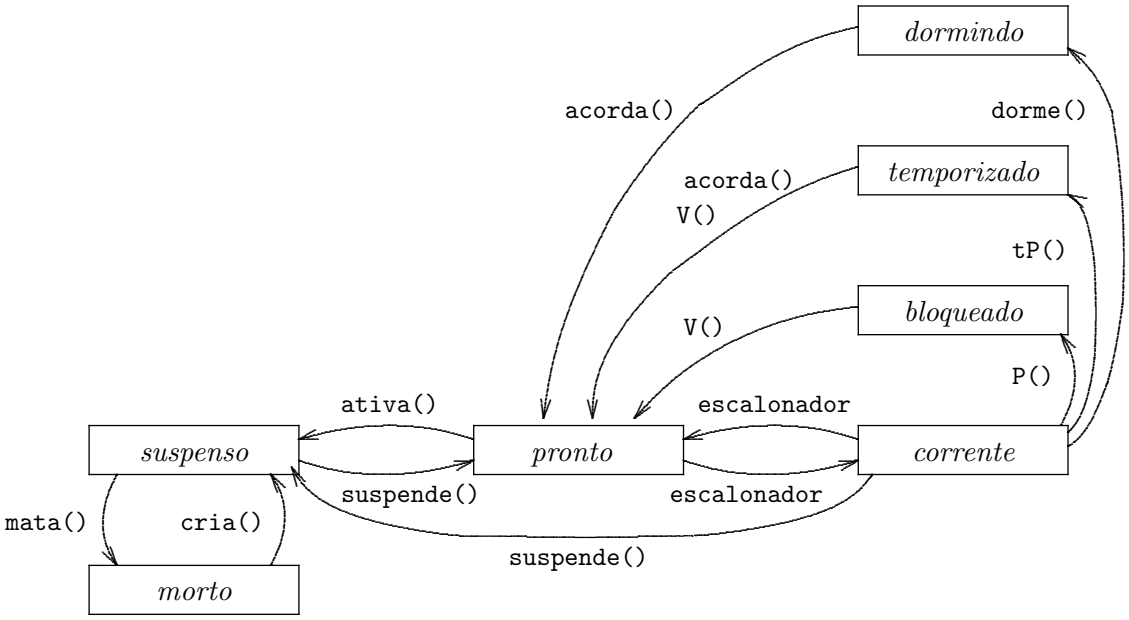

Figura 6.2: Grafo de estados dos processos do NÚCLEO do MSC.

pelas funções internas que implementam o escalonador. O código a ser executado por um processo é o de um procedimento codificado em C ou linguagem de montagem (que deve obedecer às convenções do C para a chamada e retorno de subrotinas). `cria()` permite que um número qualquer de parâmetros seja passado ao processo com o mesmo protocolo da chamada e retorno de subrotinas em C. `nargs` indica o número de parâmetros colocados na pilha pelo programador. `cria()` retorna o identificador do novo processo.

```
int   cria(onde, nomeprog, prog, prio, pilha, nomeproc, nargs, args)
int   onde;                    /* criação do processo interno ou externo */
char *nomeprog;                /* nome do programa a ser executado */
int  *prog;                    /* endereço do código, se interno */
int   prio;                    /* prioridade inicial */
int   pilha;                   /* tamanho da pilha privativa do processo */
char *nomeproc;                /* nome do novo processo */
int   nargs;                   /* número de argumentos na pilha */
int   args;                    /* argumentos de ativação */
```

O programador deve "publicar" os programas passíveis de serem transformados em processos por execuções remotas de `cria()`. Se `onde` é especificado como `EXTERNO` e `nomeprog` contém o nome de um programa previamente publicado, `cria()` efetua a criação do processo `nomeproc` na memória do processador onde o código do programa indicado reside. Se `onde` é especificado como `INTERNO`, o código do processo é aquele com início apontado por `prog`. Esta diferença no tratamento de processos internos e externos é causada pela diferença de custo entre uma operação local e uma remota. Além disso, a criação local dispensa a publicação dos programas que são visíveis apenas internamente.

Um programa é "publicado" com uma execução de `publica()`. O NÚCLEO mantém, em memória global, uma tabela com o nome do programa, endereço do código e processador onde reside, para cada um dos programas que podem ser remotamente transformados em

processos. A chave de busca nesta tabela é o nome do programa; o índice de um elemento da tabela é calculado segundo uma função "hash" [AAHO82].

```
publica(nomeprog, endprog, cpu)   /* possibilita a criação remota */
char *nomeprog;                   /* nome do programa */
int  *endprog;                    /* end. do programa na memória onde reside */
int  cpu;                         /* processador em cuja memória local reside nomeprog */
```

A execução de `mata()` remove definitivamente do sistema o processo especificado (seja ele interno ou externo). Esta primitiva efetua todas as ações necessárias para que a consistência das estruturas de dados do NÚCLEO seja preservada (os processos podem ser mortos em qualquer estado de execução). A morte de um processo não provoca a destruição automática dos recursos criados por ele.

```
mata(proc)              /* mata o processo identificado por proc */
int  proc;
```

`suspende()` e `ativa()` permitem um controle menos drástico da execução de processos do que `cria()` e `mata()`. `suspende()` torna o processo especificado inelegível para execução até que ele seja (re)ativado. A semântica das outras três primitivas é aquela indicada nos comentários.

```
suspende(proc)          /* suspende execução do proc */
int  proc;

ativa(proc)             /* permite execução de proc, que estava suspenso */
int  proc;

int  queprio()          /* retorna a prioridade do proc. corrente */

mudaprio(proc, nova)    /* muda a prioridade de proc para nova */
int  proc;
int  nova;

int  meuid()            /* retorna identificador do proc. corrente */
```

As funções `reescal()` e `apronta()` e a primitiva `troca()` implementam o escalonador. `reescal()` e `apronta()` são de uso interno ao NÚCLEO; `troca()` é invocada explicitamente pelos processos para provocar um reescalonamemto. `reescal()` escolhe o processo de maior prioridade para executar e efetua a troca de contexto, se for o caso, entre o processo corrente e o que foi escolhido. `apronta(proc)` insere o processo indicado na fila de pronto, numa posição que depende da sua prioridade.

O modo de operação do escalonador é definido por compilação condicional: ele pode ser preemptivo ou não preemptivo. A possibilidade de preempção de um processo pelo esgotamento de seu *quantum* é controlada implicitamente pelo programador. O *quantum* é o intervalo (número de "tics" do relógio de tempo-real) no qual um processo pode executar continuamente sem sofrer um reescalonamento provocado pelo NÚCLEO (mantidas as condições externas ao processo). O valor do quantum pode ser ajustado de tal forma que, em condições normais, nenhum processo o esgote. O tamanho do quantum pode ser feito tão pequeno ou tão grande quanto se queira, desta forma minimizando a interferência do reescalonamento periódico na execução dos processos.

6.5 Camada Quatro - Sincronização

O capítulo anterior discutiu a semântica das operações `P` e `V` sobre *semáforos distribuídos*. Nesta seção, apresentamos as estruturas de dados que os suportam e as operações sobre elas. A *estrutura de semáforos globais* (`gsem[ ]`), em memória global, contém um descritor para cada semáforo global. A *estrutura de semáforos* (`lsem[ ]`) contém um descritor para cada semáforo global e um para cada semáforo local. Localmente, os semáforos são diferenciados pela sua posição (índice) na estrutura de semáforos. Veja a figura 6.3.

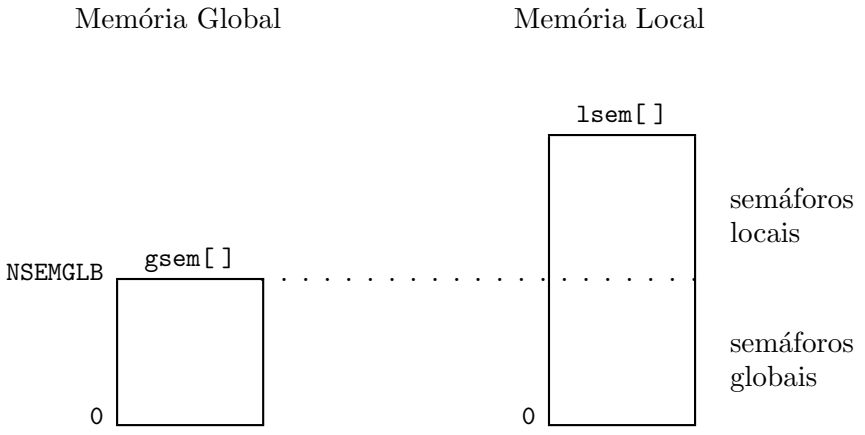

Figura 6.3: Estruturas de descritores de semáforos.

Os elememtos inferiores do vetor local (`lsem[ ]`) são alocados aos semáforos globais e os restantes aos locais. A cada semáforo global corresponde um descritor de mesmo índice na memória local. Os descritores locais e globais tem um mesmo formato:

```
struct    semaforo {        /* descritor de semáforo */
      int  estado;          /* descritor livre ou usado */
      int  valor;           /* valor do semáforo */
      int  qinicio;         /* índices da cabeça deste semáforo na ... */
      int  qfim;            /* ... estr. q[ ] , se local, ou gq[ ], se global */
}
```

Nos semáforos locais, todos os campos do descritor são usados. Nos semáforos globais, apenas a cabeça da fila de descritores de processo é usada nos descritores em memória local; os campos de valor e fila de pseudodescritores são usados nos descritores em memória global. Quando um processo está bloqueado num semáforo global, o seu descritor está inserido na fila em memória local e seu pseudodescritor está inserido na fila em memória global.

6.5.1 Operações Sobre Semáforos

A semântica das operações `P` e `V` sobre semáforos distribuídos foi discutida no capítulo anterior. Nesta seção, o funcionamento destas primitivas é discutido em mais detalhe.

Um semáforo distribuído é composto por uma parte global (elemento de `gsem[ ]`) e por uma parte local na memória de cada processador (elemento de `lsem[ ]`). Quando o processo `A`, executando no processador `CPU1`, executa `P` sobre um semáforo global e deve ficar bloqueado, o descritor de `A` é inserido na fila em memória local do semáforo (`q[ ]`);

o pseudodescritor de `A` é inserido na fila em memória global do semáforo (`gq[ ]`). Quando o processo `B`, executado pelo processador `CPU3`, efetuar o `V` que libera o processo `A`, o pseudodescritor de `A` é retirado da fila do semáforo e inserido na *fila de sinais pendentes* de `CPU1` (`gq[ ]`). `CPU1` é informado da liberação de um de seus processos por uma interrupção no VME, causada por `CPU3`. Quando esta interrupção for tratada por `CPU1`, o pseudodescritor de `A` é retirado da fila de sinais pendentes; o descritor de `A` é retirado da fila do semáforo (em memória local) e inserido na fila de pronto. Dependendo do modo de operação do escalonador, `V` pode ou não provocar um reescalonamento. Operações sobre semáforos mantém o seguinte invariante:

> *"um semáforo com valor não-negativo tem sua fila de processos vazia; um valor negativo igual a* `n` *indica que* `n` *processos estão bloqueados no semáforo."*

A primitiva `criasem()` preenche um descritor do tipo indicado (local ou global) e retorna seu identificador. `matasem()` libera os processos que eventualmente estejam bloqueados no semáforo e marca seu descritor como livre. A primitiva `valorsem()` retorna o valor do semáforo. `semreset()` reinicializa a contagem do semáforo: a execução de `semreset()` é equivalente à execução atômica de `matasem()` e `criasem()` sobre um mesmo semáforo. `multiV()` equivale à múltiplas execuções de `V` sobre um semáforo.

```
P(sem)                          /* bloqueia processo se sem.valor ≤ 0 */
int   sem;

V(sem)                          /* libera um proc. (se) bloqueado em sem */
int   sem;

int   criasem(onde, valor)      /* cria um sem. do tipo e com o valor indicados */
int   onde;                     /* memória local ou global */
int   valor;

matasem(sem)                    /* mata semáforo */
int   sem;

int   valorsem(sem)             /* retorna valor do semáforo */
int   sem;

semreset(sem, valor)            /* matasem(sem) + criasem(onde, valor) */
int   sem;
int   valor;                    /* novo valor do semáforo */

multiV(sem, n)                  /* equivale a n execuções de V(sem) */
int   sem;
int   n;
```

Um processo que executa `tempP()` e fica bloqueado, passa para o estado *temporizado*. Neste estado, um processo fica bloqueado até que seja sinalizado ou que o intervalo especificado expire. Este é o único estado em que um processo fica simultaneamente em mais de uma fila. Seu descritor é inserido na fila do semáforo, como na operação `P` normal (em `q[ ]`, ou `q[ ]` e `gq[ ]`); o descritor também é inserido na fila dormindo (`t[ ]`). Quando o processo é liberado por tempo ou sinalizado, seu descritor é retirado das duas filas. Se a liberação ocorreu por "decurso de prazo", `tempP()` retorna `TIMEOUT`, senão, retorna `OK`.

```
int    tempP(sem, ntics)  /* bloqueio temporizado */
int    sem;
int    ntics;             /* intervalo máximo de espera pela sinalização */
```

6.5.2 Operações de Temporização

Um processo pode suspender a sua própria execução durante um intervalo definido. Quando um processo está no estado *dormindo*, seu descritor está inserido na *fila de dormindo* (`t[ ]`). Os processos são inseridos nesta fila na posição determinada pelo intervalo de suspensão. O tamanho do intervalo de espera (número de tics) é mantido no campo `chave` dos elementos de `t[ ]`. O campo `chave` é ajustado de tal forma que o número de tics que um processo deve esperar para ser acordado é a somatória do número de tics de todos os processos que o antecedem na fila, mais a diferença entre o intervalo desejado e o valor da somatória. Desta forma, o tratador da interrupção do relógio de tempo-real examina e decrementa apenas o valor de `chave` do primeiro processo na fila.

```
dorme(ntics)          /* adormece o proc. corrente ntics do relógio de tempo-real */
int  ntics;
```

A função `acorda()` é o tratador das interrupções do relógio de tempo-real. Se o intervalo de espera do processo no início da fila de dormindo já expirou, ele é inserido na fila de pronto. Isto pode ou não provocar um reescalonamento, dependendo do modo de operação do escalonador e da prioridade do processo que acordou. Se o processo que acordou está bloqueado num semáforo (por `tempP()`), seu descritor é retirado da fila do semáforo e o valor do semáforo incrementado (mantendo o invariante).

6.6 Camada Cinco - Armazenadores

As primitivas da camada cinco permitem a implementação de conjuntos de armazenadores. O tamanho e o número de armazenadores num dado conjunto são especificados na sua criação. O consumo de armazenadores pelos processos é controlado por um semáforo: quando o conjunto está vazio, os processos com requisições pendentes ficam bloqueados até que armazenadores sejam devolvidos ao conjunto. Os armazenadores desocupados ficam reunidos numa lista de armazenadores livres; se o conjunto não estiver vazio, o custo de uma requisição é constante; se estiver vazio, o custo depende do tempo de utilização dos armazenadores pelos processos. Note que o dimensionamemto correto do tamanho de um conjunto permite que todas as requisições sejam atendidas sem que ocorram bloqueios (e a um custo constante e baixo).

Um conjunto de armazenadores é definido pela estrutura abaixo. Existe um vetor de conjuntos em memória global e um na memória local a cada processador. A figura 6.4 mostra os vetores de conjuntos de armazenadores em memória global e na memória local a um dos processadores.

```
struct      carm {        /* conjunto de armazenadores */
      int   tam;          /* tamanho dos armazenadores deste conjunto, em bytes */
      char  *prox;        /* lista de armazenadores livres */
      int   sem;          /* identificador do semáforo do conjunto */
}
```

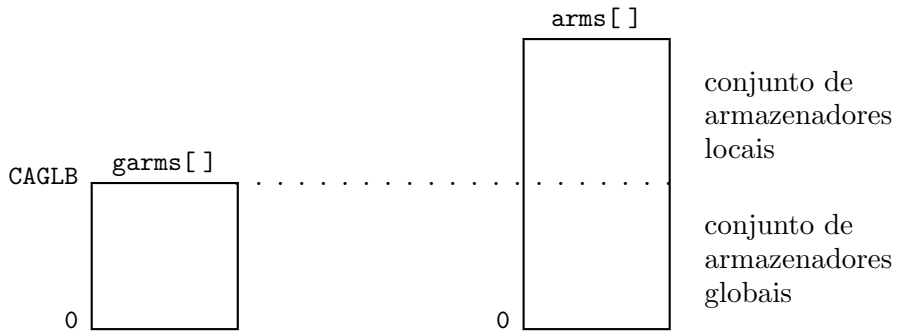

Figura 6.4: Conjuntos de Armazenadores.

Os vetores `garms[ ]` e `arms[ ]` são compostos por elementos do tipo `carm`, mostrado acima. Os conjuntos em memória local são diferenciados daqueles em memória global pelo seu índice no vetor `arms[ ]`. Os elementos de `arms[ ]` com índice `0` a `CAGLB` não são usados localmente porque correspondem a armazenadores em memória global, isto é, estes índices identificam os conjuntos em memória global. Os elementos de `arms[ ]` de índice maior que `CAGLB` descrevem os conjuntos em memória local.

A primitiva `criaarm()` requisita um bloco de memória (local ou global) de tamanho suficiente para conter `num` armazenadores com `nbytes` cada um. Além disso, `criaarm()` cria um semáforo do tipo indicado por `onde` (local ou global) com valor inicial `num` e retorna o identificador do conjunto na estrutura de armazenadores (`arms[ ]`, se local ou `garms[ ]`, se global).

```
criaarm(onde, nbytes, num)    /* cria conjunto de armazenadores */
int     onde;                 /* em memória global ou local */
int     nbytes;               /* tamanho dos armazenadores do conjunto */
int     num;                  /* quantidade de armazenadores no conjunto */
```

Os armazenadores podem ser requisitados e devolvidos pela invocação de `reqarm()` e `libarm()`. `reqarm()` retira um armazenador da lista livre e retorna um apontador para seu início. Se não houver nenhum armazenador livre, o processo é bloqueado até que algum armazenador seja devolvido ao conjunto. `libarm()` devolve o armazenador apontado à lista livre do seu conjunto (o identificador do conjunto só é explicitado na requisição de armazenadores). Os armazenadores desocupados são mantidos na lista livre apontada por `carm.prox`.

```
int   *reqarm(conj)      /* requisita um armazenador do conjunto conj */
int   conj;

libarm(arm)              /* devolve o armazenador arm ao seu conjunto */
int   *arm;              /* aponta ínicio do armazenador */
```

As operações sobre um dado conjunto de armazenadores mantém o invariante abaixo:

> *"o valor do semáforo* `sem` *é igual ao número de armazenadores livres no conjunto se* `sem.valor` > 0*; ou ao número de processos bloqueados esperando por armazenadores se* `sem.valor` < 0*".*

6.7 Camada Seis - Comunicação

A semântica das operações de envio e recepção de mensagens através de caixas postais foi discutida no capítulo anterior. Nesta seção, descrevemos as estruturas de dados que implementam as caixas postais e as operações sobre elas. As mensagens suportadas pelas primitivas desta camada têm o tamanho de um inteiro (32 bits). As mensagens pertencem a conjuntos de mensagens invisíveis ao programador (um global e outro local a cada processador). Cada mensagem tem o formato abaixo.

```
struct      msg {          /* mensagem suportada pelo núcleo */
     int    msgm;          /* valor da mensagem */
     struct msg *prox;     /* aponta próxima mensagem na lista/fila */
}
```

Uma caixa postal é descrita pela estrutura abaixo. As tabelas de caixas postais são mostradas na figura 6.5.

```
struct      cp {           /* descritor de caixa postal */
     int    estado;        /* livre/alocada/limbo */
     int    esem;          /* indentificador do semáforo de envio */
     int    rsem;          /* indentificador do semáforo de recepção */
     int    cap;           /* capacidade da fila de mensagens */
     struct msg *inicio;   /* fila de mensagens pendentes */
     struct msg *fim;
}
```

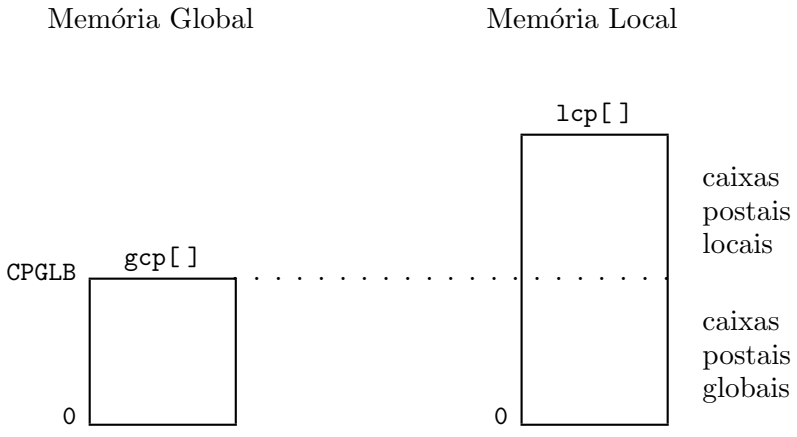

Figura 6.5: Tabelas de caixas postais.

A localização de uma caixa postal (global ou local) é determinada pelo seu índice na tabela `lcp[ ]`. Se o índice estiver entre `0...CPGLB`, a caixa postal é em memória global; senão, ela é em memória local (ou o índice é inválido). Os elementos com índice na faixa `0...CPGLB`, na tabela de caixas postais em memória local não são usados porque correspondem às caixas postais em memória global. Da mesma forma que com a tabela de conjuntos de armazenadores, o espaço desperdiçado em `lcp[ ]` é compensado pela simplicidade e eficiência com que as caixas postais são diferenciadas em locais e globais.

A primitiva `criacp()` cria uma caixa postal com a capacidade especificada (`nmsgs`) e retorna seu identificador. `criacp()` cria os semáforos que controlam a recepção (valor inicial 0) e a transmissão (valor inicial `nmsgs`) de mensagens. `destroicp()` libera os processos eventualmente bloqueados nos semáforos e entrega as mensagens pendentes para a rotina especificada; os semáforos são então destruídos. `resetcp()` equivale a uma execução atômica de `destroicp()` e `criacp()`. Uma caixa postal que está sendo destruída ou reinicializada é colocada no "limbo" e nenhuma operação sobre ela é possível enquanto a destruição ou reinicialização não estiverem completas.

```
criacp(onde, nmsgs)      /* cria uma caixa postal */
int   onde;              /* memória local ou global */
int   nmsgs;             /* capacidade da fila de mensagens */

destroicp(cp, dispoe)    /* destói a caixa postal cp */
int   cp;
int   (*dispoe)();       /* dispõe as mensagens pendentes */

resetcp(cp, dispoe)      /* reinicializa a caixa postal cp ... */
int   cp;                /* ... sem alterar parâmetros anteriores */
int   (*dispoe)();       /* dispõe as mensagens pendentes */
```

`examcp()` retorna o valor do semáforo de recepção da caixa indicada: se maior que `0`, indica o número de mensagens na fila; se menor que `0`, indica o número de processos bloqueados à espera de mensagens.

```
int   examcp(cp)         /* "examina" caixa postal */
int   cp;
```

`envia()` deposita uma mensagem na caixa postal especificada. O processo que executa `envia()` pode ou não ser bloqueado, dependendo da capacidade da fila de mensagens. Se a fila tiver capacidade para uma mensagem apenas, o envio é síncrono. O assincronismo do envio depende da capacidade da fila e da velocidade relativa entre a execução do(s) processo(s) que envia(m) e recebe(m) mensagens.

```
envia(cp, msg)           /* deposita msg em cp */
int   cp;
int   msg;               /* mensagem */
```

`recebe()` retira a primeira mensagem da fila de mensagens da caixa postal indicada. Se não houver nenhuma mensagem à espera, o processo que executa `recebe()` é bloqueado até que alguma mensagem seja depositada na caixa.

```
int   recebe(cp)         /* retira mensagem de cp */
int   cp;
```

As primitivas `tenvia()` e `trecebe()` são similares a `envia()` e `recebe()`, respectivamente. O que as diferencia é que as primeiras impedem que um processo fique eternamente bloqueado à espera de espaço na fila de mensagens (`envia()`) ou à espera de que uma mensagem seja depositada na caixa postal (`recebe()`). Se o intervalo expirar, estas primitivas retornam `TIMEOUT`. Processos bloqueados a espera de mensagens, devido a execução destas duas primitivas, ficam no estado *temporizado*.

```
tenvia(cp, msg, ntics)          /* envio temporizado de mensagens */
int   cp;
int   msg;
int   ntics;                    /* intervalo máximo de espera */

trecebe(cp, ntics)              /* recepção temporizada de mensagens */
int   cp;
int   ntics;
```

6.8 Primitivas e Escalonamento

O modo de operação do escalonador pode ser selecionado como preemptivo ou não-preemptivo, dependendo de uma opção de compilação condicional. Algumas primitivas sempre provocam um reescalonamento, devido à sua própria natureza ou devido a seus parâmetros. Outras, somente provocam um reescalonamento se o modo de operação do escalonador é preemptivo (neste caso, o processo corrente é sempre aquele de maior prioridade dentre os prontos para executar).

As primitivas do NÚCLEO que sempre provocam reescalonamento são

`troca()`, `suspende(p-corrente)`, `mata(p-corrente)`, `P()`, `tP()` e `dorme()`.

As primitivas que provocam reescalonamento em modo preemptivo, dependendo das prioridades relativas do processo corrente e dos processos prontos para executar, são

`mudaprio()`, `reativa()`, `V()`, `matasem()`, `multiV()`, `semreset()` e `acorda()`.

6.9 Inicialização do Núcleo

As estruturas de dados do NÚCLEO são inicializadas em duas etapas. Na primeira, o processador instalado na posição 1 do barramento VME (o controlador do barramento) inicializa todas as estruturas em memória global. Feito isso, os outros processadores são sinalizados com interrupções no VME. Esta interrupção libera os processadores, que podem então inicializar as estruturas do NÚCLEO em suas memórias locais.

Após a inicialização, o NÚCLEO cria um processo de usuário em cada um dos processadores – o Configurador. O que acontece a partir deste momento depende da aplicação. Tipicamente, os Configuradores criam todos os recursos necessários à execução do aplicativo, publicam os programas elegíveis à criação remota de processos e criam os processos que compõem o programa (distribuído) de aplicação.

Capítulo 7

Conclusão

A primeira seção deste capítulo descreve a estratégia de desenvolvimento que empregamos na implementação do NÚCLEO. A seção seguinte discute a otimização de seu código, com vistas a ganhos em velocidade de execução e propõe uma revisão na estrutura do NÚCLEO. Finalmente, a Coda expõe nossa opinião sobre os resultados obtidos com este trabalho.

7.1 Estratégia de Desenvolvimento

Como parte da estratégia de desenvolvimento, o sistema XINU [COME84] foi inicialmente transportado do processador DEC LSI 11 para os processadores do MSC (MC 68000). Feito isso, o XINU foi adaptado para processamento paralelo: o código das primitivas foi reescrito para acomodar a semântica das operações "distribuídas" e então otimizado para funcionamento dentro dos requisitos de tempo-real. O núcleo do XINU foi usado como um protótipo; algumas de suas partes e idéias foram aproveitadas, tais como a estrutura em camadas e algumas das estruturas de dados; outras, adaptadas ou completamante reprojetadas, como a troca de contexto (`ctxsw()`) e as estruturas em memória global, por exemplo.

O núcleo original do XINU é composto por 2200 linhas de código fonte (comentários inclusive), com 11% em linguagem de montagem do LSI 11. Estas 2200 linhas excluem os "drivers" de tty e as funções de biblioteca (como `printf()`, por exemplo). O NÚCLEO do MSC contém 4600 linhas de código fonte, com 9% em linguagem de montagem do MC 68000. O código executável ocupa 18.800 bytes.

7.2 Otimização

O código do NÚCLEO necessita de alguma "sintonia fina" para que o tempo de execução das primitivas seja reduzido. A Seção 8 do Apêndice discute uma forma simples de melhorar a velocidade de execução de algumas das primitivas: a otimização manual do código gerado pelo compilador (o sistema de desenvolvimento que empregamos na implementação do NÚCLEO não possui um otimizador de código). Este tipo de ajuste piora a portabilidade do código mas possibilita bons resultados com um esforço relativamente pequeno (redução de 30 a 40% nos tempos de execução das primitivas assim otimizadas).

Um problema com as estruturas de filas de processos da Camada Dois é a grande freqüência com que as funções que manipulam as filas de processos são invocadas. O protocolo de entrada e de saída das funções codificadas em C consome muito tempo: todos os registradores usados pela função devem ser empilhados na sua entrada e desempilhados

na saída (retorno). O MC 68000 possui oito registradores de dados (D0 a D7, D7 é de uso reservado pelo compilador) e oito registradores de endereço (A0 a A7, A7 é o apontador de pilha); o salvamento ou a carga de cada um dos registradores implica em dois acessos à memória (16 + 16 bits).

Uma revisão na estrutura do NÚCLEO, com a recodificação ou a eliminação da Camada Dois (filas de processos), traria ganhos substanciais na velocidade de execução. A implementação das filas de processo através de índices nas estruturas `q[]`, `gq[]` e `t[]` é muito interessante do ponto de vista organizacional mas o custo em termos de desempenho é alto. A inserção de um processo numa fila, da forma como atualmente implementado, envolve o cálculo de quatro endereços, no melhor caso. A alternativa seria a implementação das filas de processos com apontadores. Isto evita o cálculo de endereços, agilizando as operações sobre filas de processos.

A Camada Dois poderia ser recodificada, mantendo o mesmo conjunto de funções. A sintaxe destas funções seria ligeiramente alterada uma vez que seus parâmetros reais passariam a ser apontadores e não mais índices (números inteiros). Por outro lado, a Camada Dois poderia ser eliminada e a manipulação de filas de processos dispersa pelo código das primitivas. Isto reduziria substancialmente o número de invocações de procedimentos, com a contrapartida de um aumento também substancial na entropia do código do NÚCLEO.

7.3 Coda

O NÚCLEO de tempo-real do MSC oferece ao programador uma série de facilidades que lhe permitem programar o MSC de uma maneira eficiente e simples. A Transparência de Multiprocessamento simplifica a programação de aplicativos "distribuídos". Se todos os recursos necessários à execução de um programa forem criados somente no(s) Configurador(es), o transporte deste aplicativo para uma máquina com um número diferente de processadores tem um custo mínimo uma vez que os nomes dos objetos compartilhados (semáforos, armazenadores e caixas postais) são parâmetros dos procedimentos que se tornarão os processos do programa concorrente. Ainda, a Transparência de Multiprocessamento torna a programação do MSC, que é um multiprocessador, não mais complexa que a de um sistema multitarefa convencional. A localização dos objetos suportados pelo NÚCLEO é especificada somente na sua criação. Todas as outras operações sobre eles referenciam apenas aos seus identificadores.

O projeto do NÚCLEO segundo um modelo estratificado torna a sua implementação inerentemente simples e bem estruturada. A ampliação do sistema, isto é, a adição de novas camadas, consiste apenas da combinação dos serviços oferecidos pelas camadas inferiores de uma forma direta, sem perda de eficiência nem generalidade (como demonstrado no Apêndice). Outra característica interessante do NÚCLEO é a simplicidade dos mecanismos suportados por ele. Isto permite a implementação das políticas de controle da execução dos programas que forem mais convenientes à cada aplicação.

Referências Bibliográficas

[AAHO82] A. V. Aho, J. E.Hopcroft, J. D. Ullman; *Data Structures and Algorithms*; Addison-Wesley; 1982.

[ABBO84] C. Abbot; *Intervention Schedules for Real-Time Programming*; IEEE Trans. on Software Engineering, SE-10(3); Maio 1984.

[ADAN86] J. M. Adán Coello; *Suporte de Tempo-Real para um Ambiente de Programação Concorrente*; Dissertação de Mestrado em Eng. Elétrica, FEC UNICAMP; Agosto 1986.

[ANDR83] G. R. Andrews, F. B. Schneider; *Concepts and Notations for Concurrent Programming*; ACM Computing Surveys, 15(1); Março 1983.

[APPE87] L. Appezato; *Um Processador de Comunicação de Rede Local para Controle de Processos*; V Simp. Bras. de Redes de Computadores, São Paulo; Abril 1987.

[BACH86] M. J. Bach; *The Design of the UNIX Operating System*; Prentice-Hall Software Series; 1986.

[BARI82] M. Ben-Ari; *Principles of Concurrent Programming*; Prentice-Hall; 1982.

[CAVa87] E. Cavalli, M. C. Zabeu; *Sistema Hardware para Processamento Paralelo*; I Simp. Bras. de Arquitetura de Computadores - Processamento Paralelo, Gramado; Maio 1987.

[CAVb87] E. Cavalli, M. C. Zabeu; *Sistema Operacional para Processamento Paralelo*; I Simp. Bras. de Arquitetura de Computadores - Processamento Paralelo, Gramado; Maio 1987.

[COME84] D. Comer; *Operating Systems Design,the XINU Approach*; Prentice-Hall Software Series; 1984.

[CONT85] D. Conte, D. Del Corso; *Multi-Microprocessor Systems for Real-Time Applications*; D. Reidel Publishing Co.; 1985.

[CORS86] T. B. Corso, S. S. Toscani, C. M. da Costa; *A Implementação da Máquina Pascal Concorrente no Ambiente de Multiprocessamento da Máquina Micro-Bis*; VI Congr. Soc. Bras. de Computação (XIII SEMISH), Recife-Olinda; Julho 1986.

[DIJK68] E. W. Dijkstra; *The Structure of the"THE" Multiprogramming System*; CACM, 11(5); Maio 1968.

[DRUM87] R. Drummond, F. A. Vanini; *Sistema Operacional Sofia*; VII Congr. Soc. Bras. de Computação (XIV SEMISH), Salvador; Julho 1987.

[DUTR86] L. S. V. Dutra, A. J. Catto; *MIMD, Nível Triplo Aplicativo,...? Como Classificar uma Arquitetura*; VI Congr. Soc. Bras. de Computação (XIII SEMISH), Recife-Olinda; Julho 1986.

[ENSL78] P. H. Enslow; *What is a Distribuited Data Processing System*; IEEE Computer, Jan. 1978.

[FALL87] N. Faller, et al.; *Técnicas de Projeto Utilizadas na Construção do Supermicrocomputador Pégasus-32X e do Sistema Operacional Plurix*; I Simp. Bras. de Arquitetura de Computadores - Processamento Paralelo, Gramado; Maio 1987.

[FLYN72] M. J. Flynn; *Some Computer Organizations and Their Effectiveness*; IEEE Trans. on Computers; Set. 1972.

[GAJK85] D. D. Gajski, J. K. Pier; *Essential Issues in Multiprocessing Systems*; IEEE Computer, 18(6); Junho 1985.

[GURD85] J. R. Gurd, C. C. Kirkham, I. Watson; *The Manchester Prototype Dataflow Computer*; CACM 28(1); Jan. 1985.

[HABE76] A. N. Habermann; *Introduction to Operating System Design*; Science Research Associates Inc.; 1976.

[HANS70] P. Brinch-Hansen; *The Nucleus of a Multiprogramming System*; CACM, 13(4); Abril 1970.

[HEXS87] R. A. Hexsel, R. C. Machado; *Multiprocessador para Sistemas de Controle*; Documento Técnico do Instituto de Automação (DTIA), CTI; Junho de 1987.

[HEXS88] R. A. Hexsel, R. Drummond; *Sincronização e Comunicação no Multiprocessador MSC*; VIII Congr. Soc. Bras. de Computação (XV SEMISH), Rio de Janeiro; Julho 1988.

[HOAR74] C. A. R. Hoare; *Monitors: An Operating Systems Structuring Concept*; CACM, 17(10); Out. 1974.

[JONE80] A. K. Jones, P. Swarts; *Experiences Using Multiprocessor Systems - A Status Report*; ACM Computing Surveys; Junho 1980.

[KRAM83] J. Kramer, et al.; *CONIC: An Integrated Approach to Distribuited Computer Control Systems*; Proc. IEEE, 30(1), pt. E; Janeiro 1983.

[KERN84] B. W. Kernighan, R. Pike; *The UNIX Programming Environment*; Prentice-Hall Software Series; 1984.

[KERN86] B. W. Kernighan, D. M. Ritchie; *C – A Linguagem de Programação*; Ed. Campus, EDISA; 1986.

[KUNG88] H. T. Kung; *Parallel Computer Architectures*, curso ministrado na VI Escola de Computação, Campinas; Julho 1988.

[LAMP85] L. Lamport; *Solved Problems, Unsolved Problems and Nonproblems in Concurrency*; ACM Operating Systems Review, 19(4); Out. 1985.

[LPSN84] B. W. Lampson; *Hints for Computer System Design*; IEEE Software, 1(1); Jan. 1984.

[LOPE86] A. B. Lopes; *LPM e LCM: Linguagens para a Programação e Configuração de Aplicações de Tempo Real*; Dissertação de Mestrado em Sistemas e Computação, CCT UFPb; Ago. 1986.

[LOPE87] A. B. Lopes, J. M. Adán Coello, M. F. Magalhães; *Um Ambiente para o Desenvolvimento de Software de Tempo-Real e sua Implementação*; VII Congr. Soc. Bras. de Computação (XIV SEMISH), Salvador; Julho 1987.

[MASU80] R. Masuda, K. Hasegawa; *Mark Flow Graph and its Application to Complex Sequential Control System*; Proc. of the 13th Intern. Conf. on System Science, Hawaii; Jan. 1980.

[MEIR88] S. L. Meira; *Introdução à Programação Funcional*; VI Escola de Computação, Campinas; Julho 1988.

[NACA88] L. Nacamura Jr; *Projeto e Implementação de um Núcleo de Sistema Operacional Distribuído com Mecanismos para Tempo Real*; Dissertação de Mestrado em Eng. Elétrica, PGEE UFSC; Julho 1988.

[NETT86] J. C. Netto, et al.; *SUMUS – Projeto Bi-Microprocessador*; VI Congr. Soc. Bras. de Computação (XIII SEMISH), Recife-Olinda; Julho 1986.

[OSBa81] A. Osborne, G. Kane; *4 & 8-Bit Microprocessor Handbook*; Osborne/Mc Graw-Hill; 1981.

[OSBb81] A. Osborne, G. Kane; *16-Bit Microprocessor Handbook*; Osborne/Mc Graw-Hill; 1981.

[PAUL81] R. P. Paul; *Robot Manipulators: Mathematics, Programming and Control*; MIT Press; 1981.

[PETE83] J. Peterson, A. Silberschatz; *Operating Systems Concepts*; Addison-Wesley; 1983.

[RITC74] D. M. Ritchie, K. Thompson; *The UNIX Timesharing System*; CACM, 17(7); Julho 1974.

[SCHO84] J. D. Schoeffler; *Distribuited Computer Systems for Industrial Process Control*; IEEE Computer, 17(2); Fev. 1984.

[SEIT85] C. L. Seitz; *The Cosmic Cube*; CACM, 28(1); Jan. 1985.

[THAC87] C. P. Thacker, L. C. Stewart; *Firefly: a Multiprocessor Workstation*; II Intern. Conf. on Architectural Support for Progr. Lang. and Oper. Systems (ASPLOS II), Palo Alto; Out. 1987.

[TREL88] P. C. Treleaven; Palestras no VIII Congr. da Soc. Bras. de Computação, Rio de Janeiro; Julho 1988.

[VMEb82] VMEbus Manufacturers' Group; *VMEbus Specification Manual - Rev. B*; Ago. 1982.

[VMXb83] VMEbus Manufacturers' Group; *VMXbus Specification Manual - Rev. A*; Out. 1983.

[WILL87] T. J. Williams; *Recent Developments in the Application of Plant-Wide Computer Control*; Computers in Industry, num. 8, North-Holland; 1987.

[WILS83] P. Wilson; *Occam Architecture Eases System Design*; Computer Design, 22(13); Nov. 1983.

[WIRT77] N. Wirth; *Toward a Discipline of Real-Time Programming*; CACM, 20(8); Ago. 1977.

[WULF74] W. A. Wulf, et al; *HYDRA: the Kernel of a Multiprocessor Operating System*; CACM,17(6); Junho 1974.

Apêndice A

Camada Sete: Comunicação Intermódulos

O MSC pode ser encarado tanto como um sistema distribuído quanto como um multiprocessador. O grau de acoplamento entre os processadores pode ser determinado pela programação. O MSC pode ser usado como um multiprocessador e, neste caso, a interação entre os processos ocorre através de variáveis compartilhadas em memória global. No outro extremo, o MSC pode ser usado como um sistema fracamente acoplado onde os processos interagem apenas através de troca de mensagens. O NÚCLEO suporta as duas formas de comunicação interprocessos de uma maneira eficiente.

Além das aplicações em controle, o MSC pode ser usado como um laboratório para pesquisas em sistemas distribuídos. Este apêndice mostra como um ambiente de desenvolvimento de programas distribuídos para aplicações em controle pode ser suportado por uma camada adicional do NÚCLEO, que implementa os serviços de comunicação, nos quais tal ambiente é baseado. Para tanto, demonstramos como a "Camada Sete" do NÚCLEO, usando as primitivas das camadas inferiores, pode oferecer primitivas de comunicação com a mesma semântica daquelas do "Suporte de Tempo Real" (STR) [ADAN86]. O STR faz parte de um ambiente de programação (descrito abaixo) que está em desenvolvimento no Instituto de Automação do CTI e é baseado no projeto CONIC, concebido no Imperial College of Science and Technology (Londres) [KRAM83].

A finalidade deste apêndice não é mostrar a implementação de todas as primitivas do STR. A discussão é centrada na equivalência semântica entre a comunicação no STR e na Camada Sete do NÚCLEO. Uma discussão sobre os outros tópicos envolvidos na implementação completa do STR foge ao escopo deste trabalho.

Na próxima seção, o ambiente de programação do qual o STR faz parte é apresentado. A seguir, a semântica das primitivas de comunicação suportadas pelo STR é mostrada. Finalmente, discutimos a implementação de um subconjunto destas primitivas como parte da Camada Sete do NÚCLEO.

A.1 O Ambiente para Programação de Aplicações de Tempo-Real

O Ambiente para Programação de Aplicações em Tempo Real [LOPE87], a que chamaremos de o *Ambiente*, oferece ao projetista de sistemas de tempo-real um conjunto de ferramentas que facilitam a estruturação, desenvolvimento, testes e configuração de aplicativos.

O Ambiente é constituído por duas linguagens, a Linguagem de Programação de Módulos e a Linguagem de Configuração de Módulos [LOPE86] e pelo Suporte de Tempo

Real. O Ambiente está associado a uma metodologia de desenvolvimento de programas que retrata o projeto de um módulo de hardware (placa) onde os componentes (circuitos integrados) são estruturas modulares que encapsulam dados e funções e interagem com o mundo externo através de interfaces precisamente definidas.

A Linguagem de Programação de Módulos (LPM) é uma extensão do Pascal e fornece ao programador facilidades para expressar o conceito de módulo, suas interfaces e as interações entre os módulos. A Linguagem de Configuração de Módulos (LCM) permite a identificação dos tipos de módulos (escritos em LPM) e o estabelecimento de conexões entre os mesmos. Um programa de aplicação é dividido em módulos e cada módulo contém uma ou mais tarefas (ou processos). A geração de um aplicativo é dividida em duas fases. Na *fase de programação*, os módulos são codificados independentemente uns dos outros. Na *fase de configuração*, os módulos são agregados segundo o programa de configuração que especifica os componentes e as interações entre os componentes de um aplicativo.

A LPM oferece como paradigma de sincronização e comunicação a troca de mensagens através de portas. As primitivas de envio e recepção de mensagens não endereçam diretamente os módulos envolvidos na comunicação mas sim a portas de entrada e de saída locais a cada módulo. A criação dos canais lógicos de comunicação entre os módulos é efetivada pelo estabelecimento de conexões entre portas do mesmo tipo na fase de configuração.

A LCM permite que o programador descreva a configuração de um aplicativo, isto é, quais os módulos que o compõem e quais as interações entre eles. Um programa de configuração é formado pelas declarações dos tipos dos módulos que compõem o aplicativo, pela especificação das instâncias destes módulos e pela conexão das portas das instâncias dos módulos. Um *comando LCM de conexão* especifica a ligação entre a porta de saída e a porta de entrada que formam um canal de comunicação.

O STR fornece o suporte de tempo de execução aos comandos, funções pré-definidas e procedimentos da LPM que não fazem parte do Pascal padrão. O STR oferece serviços para a troca de mensagens entre tarefas, mudança de prioridade de um módulo, temporização e tratamento de interrupções. Atualmente, o STR está implementado para execução em computadores do tipo IBM-PC.

A.2 Semântica das Primitivas de Comunicação do STR

As mensagens fluem de portas de saída para portas de entrada. Tanto as mensagens quanto as portas envolvidas num canal lógico de comunicação devem ser do mesmo *tipo*. Uma porta pode ser síncrona ou assíncrona. Ligações do tipo multidestino são possíveis somente entre portas assíncronas; ligações do tipo "muitos para um" podem ser estabelecidas tanto entre portas assíncronas como entre portas síncronas. No caso de conexões multidestino, cada porta de entrada conectada recebe uma cópia da mensagem enviada através da porta de saída.

As mensagens enviadas através de portas de saída síncronas são enfileiradas na porta de entrada do destino; em cada instante, apenas uma mensagem enviada por uma certa porta síncrona pode estar enfileirada numa porta de entrada. As mensagens enviadas através de portas de saída assíncronas são armazenadas na porta de entrada do destino. Se o armazenador estiver cheio, as mensagens mais antigas são substituídas pelas mensagens recém-chegadas.

O STR mantém o estado de execução das tarefas nos *Blocos de Controle de Módulos* (BCM). Além de informações relativas ao estado de execução dos módulos, o BCM contém a lista de portas internas ao módulo. A cada porta de entrada é associado um *Descritor de*

Porta de Entrada (DPE). O DPE contém o armazenador (se for o caso) e a especificação do tipo (síncrono ou assíncrono) de cada porta de entrada, além de outras informações. O DPE das portas síncronas mantém um campo com o número da comunicação síncrona corrente. Este número é usado na validação das trocas de mensagens. Outro campo mantém um código de terminação da última operação efetuada (erro ou terminação normal).

Um canal lógico de comunicação é estabelecido pela ligação de uma porta de saída a uma porta de entrada. A tabela de portas do BCM contém elementos alocados a portas de entrada e a portas de saída. Os elementos alocados a portas de entrada contém apontadores aos DPEs das portas locais ao módulo. Os elementos alocados as portas de saída contém as conexões entre portas de entrada e de saída, isto é, contém apontadores aos DPEs das portas de entrada às quais estão conectadas. No caso de conexões multi-destino, ao invés de apontar um DPE, o elemento da tabela de portas aponta uma lista de DPEs.

Os próximos parágrafos apresentam os serviços de comunicação suportados pelo STR e os comandos LPM correspondentes.

Envio Assíncrono de uma Mensagem: envia mensagem pela porta assíncrona especificada.

```
SEND <mensagem> TO <porta de saída>
```

Envio Síncrono de uma Mensagem: envia mensagem pela porta síncrona especificada, esperando pela mensagem de resposta.

```
SEND <mensagem> TO <porta de saída> WAIT <resposta>
```

Envio Síncrono com Cláusula para Tratamento de Falha: envia mensagem pela porta síncrona especificada, esperando a resposta durante um intervalo definido. Se a resposta for recebida antes do esgotamento do intervalo, é executado o comando `Si`; senão, `Sj`.

```
SEND <mensagem> TO <porta de saida>
     WAIT <resposta>→ Si
     FAIL <tempo de espera>→ Sj
END
```

Recepção Bloqueante de uma Mensagem: retorna imediatamente se houver uma mensagem esperando na porta de entrada, senão, bloqueia a tarefa até que alguma mensagem seja recebida.

```
RECEIVE <mensagem> FROM <porta de entrada>
```

Envio de Mensagem de Resposta: deve seguir a execução de um RECEIVE; senão não tem efeito algum. Uma troca de mensagens síncrona é encerrada pela execução de um REPLY, pelo receptor da mensagem que iniciou a troca de mensagens. O REPLY pode ou não ser executado imediatamente após o RECEIVE.

```
RECEIVE <mensagem> FROM <porta de entrada>
REPLY <resposta> TO <porta de entrada>
```

Desvio de uma Comunicação Síncrona: a mensagem recebida através da porta de entrada é retransmitida, inalterada, para a porta de saída.

```
FORWARD <porta de entrada> TO <porta de saída>
```

Recepção Seletiva de uma Mensagem: este comando escolhe para execução o comando da cláusula não precedida por uma guarda (WHEN), ou precedida por uma guarda com resultado verdadeiro ou o comando precedido por uma cláusula de repetição (FOR) com a variável de controle dentro da faixa válida. O comando da cláusula escolhida é executado. A seleção dentre as cláusulas válidas pode ser aleatória ou por prioridade (a primeira cláusula da lista é a de maior prioridade). O comando pode ser um RECEIVE ou um TIMEOUT. No caso do TIMEOUT, se nenhuma mensagem for recebida antes que se esgote o período especificado, o comando `St` associado ao TIMEOUT é executado, terminando o PSELECT.

```
PSELECT
     WHEN <expr 1> RECEIVE <mens 1> FROM <porta 1>
         → S1
     ⋮
     OR WHEN<expr i> RECEIVE <mens i>FROM<porta i>
         → Si
     ⋮
     OR FOR <var.  de controle> := <x> TO <y>
         RECEIVE <mens j> FROM <porta j>
             → Sj
     ⋮
     OR TIMEOUT <tempo de espera>
         → St
END
```

Cancelamento de uma Comunicação Síncrona: ABORT pode ser usado no lugar de um REPLY para cancelar uma comunicação síncrona.

```
ABORT( <porta de entrada> , <motivo> )
```

Determinação da Razão de Falha de uma Comunicação Síncrona: REASON retorna a constante `etimeout` se o tempo de espera especificado num comando com tratamento de falha expirou; ou, `elink` se o módulo destino da mensagem respondeu com ABORT ao invés de REPLY. REASON é usada nos comandos que seguem a cláusula FAIL.

Teste de Conexão de uma Porta de Saída: LINKED retorna TRUE se a porta de saída estiver conectada; caso contrário, retorna FALSE.

```
LINKED <porta de saída>
```

Determinação da Quantidade de Mensagens Enfileiradas numa Porta de Entrada: QLEN retorna o número de mensagens disponíveis numa porta de entrada.

```
QLEN( <porta de entrada> )
```

A.3 Comunicação Intermódulos

Nesta seção, discutimos a implementação das primitivas de comunicação intermódulos, baseada nas primitivas de sincronização e comunicação do NÚCLEO. A discussão que segue é centrada no aspecto semântico da implementação; a equivalência funcional exata entre as primitivas de comunicação do STR e as da camada sete não é enfatizada. Algumas questões importantes são ignoradas, como por exemplo, o mapeamento dos BCM do STR nos descritores de processo do NÚCLEO. Na descrição das primitivas, empregamos um pseudocódigo que mistura comandos da linguagem C, as primitivas do NÚCLEO e "comandos" em Português. Os nome das primitivas resultantes são os mesmos que aqueles em [ADAN86], com o caracter "_" substituído por ".".

A.3.1 Portas

No MSC, a localização das portas, em memória local ou em memória global, é definida em função do programa de configuração gerado pela LCM. Este programa especifica quais módulos são carregados e executados em qual processador. A distribuição dos módulos estabelece a localização das portas de entrada e dos DPEs, ou seja, módulos residentes em processadores distintos comunicam-se por portas (e DPEs) em memória global. Os DPEs são agrupados num vetor de DPEs, dividido da mesma forma que o vetor de caixas postais (seção 6.7). Esta implementação simplifica a diferenciação entre os DPEs locais e os globais.

Um DPE é preenchido em função do comando LPM que define a porta de entrada a ele associada (veja os detalhes em [LOPE86, ADAN86]). O conteúdo de um DPE da camada sete é preenchido em função do tipo de porta que ele descreve. Se a porta for síncrona, a comunicação e sincronização entre tarefas é obtida através de uma caixa postal com capacidade para uma mensagem (o processo que envia uma mensagem sempre fica bloqueado à espera da recepção). O protocolo da troca de mensagens síncronas é chamado de *transação* porque envolve uma série de operações: um SEND, um RECEIVE e um REPLY ou ABORT.

Se a porta for assíncrona, o DPE possui um armazenador para as mensagens. Dois apontadores mantém uma fila circular de mensagens, possibilitando a substituição das mensagens antigas pelas novas. Um semáforo (`mutex`), protege a manipulação do armazenador, de tal forma que a inclusão ou retirada de mensagens seja uma operação atômica. Note que a semântica das operações sobre portas assíncronas não é a mesma das operações sobre "caixas postais assíncronas". Na porta assíncrona, se o armazenador estiver cheio, as mensagens mais antigas são descartadas. Na "caixa postal assíncrona", se a fila de mensagens estiver cheia, o processo que tentar depositar uma mensagem na caixa fica bloqueado.[1]

Um DPE da Camada Sete é mostrado abaixo. Os elementos de um DPE são usados em função do tipo de porta que ele descreve. Se a porta for síncrona, o semáforo `mutex` é usado para tornar atômicas as transações que envolvem o DPE. As caixas postais são criadas com capacidade para uma mensagem e seus identificadores são armazenados em `cpdest` e `cpresp`. Se a porta for assíncrona, `mutex` torna atômica a manipulação do armazenador do DPE. O semáforo `scont`, inicializado em `0`, mantém o número de mensagens enfileiradas na porta. As caixas postais ou o semáforo `scont` são criados apenas se o tipo de porta assim o exigir. Em ambos os casos, `mutex` é inicializado em `1`.

[1] Isso pode ser (parcialmente) resolvido pelo superdimensionamento da fila de mensagens.

```
struct DPE {              /* Descritor de Porta de Entrada */
     int modulo;          /* identificador do módulo "dono" da porta */
     int onde;            /* memória local ou global */
     int tipo;            /* porta síncrona ou assíncrona */
     int mutex;           /* semáforo que garante atomicidade das operações */
     /* campos usados em portas síncronas */
     int cpdest;          /* identificador da caixa postal associada ao DPE */
     int cpresp;          /* identif. da caixa postal para envio de respostas */
     int status;          /* código de terminação da última operação */
     /* campos usados em portas assíncronas */
     int scont;           /* semáforo que mantém número de mensagens pendentes */
     int arm[TAMMAX];     /* armazenador para mensagens */
     int nova;            /* aponta mensagem mais recente em arm[ ] */
     int velha;           /* aponta mensagem mais antiga em arm[ ] */
}
```

A.3.2 Primitivas da Camada Sete

A implementação das primitivas de comunicação intermódulos do STR com as primitivas do NÚCLEO é discutida a seguir. O código das primitivas que implementam os comandos ABORT e REASON não é mostrado porque não contém nenhuma aplicação das primitivas do NÚCLEO. Estas funções (determinação da razão de uma falha e teste de conexão de uma porta de saída) apenas examinam o estado das estruturas de dados (tabelas de portas do BCM e DPEs) e retornam valores em função deste exame.

Note que usamos a alocação de blocos de memória da(s) lista(s) livre do sistema. Isto permite a alocação de blocos de tamanho variável, mas, a um custo também variável. Poderíamos ter empregado conjuntos de armazenadores, com o tamanho dos armazenadores sendo definido pela maior mensagem passível de ser transferida.

Envio Assíncrono de uma Mensagem

A comunicação assíncrona permite conexões multidestino. `env.a()` copia a mensagem em todas as portas de entrada conectadas à porta através da qual a mensagem está sendo enviada. `scont` é sinalizado apenas se o armazenador não estiver cheio. Caso contrário, a mensagem mais antiga é substituída pela recém chegada; isto não altera o número de mensagens no armazenador, à espera de recepção.

```
env.a(prtsaida, tammen, endmen)
int   prtsaida;                    /* identificador da porta de saída */
int   tammen;                      /* tamanho da mensagem */
int   *endmen;                     /* aponta endereço da mensagem a enviar */
{
     int   pe;                     /* mantém identificador da porta destino */

     /* execução em exclusão mútua */
     while( não enviou para todas portas conectadas ) {
          pe = índice do DPE da próxima conexão;
          P(DPE[pe].mutex);        /* exclusão no acesso à arm[ ] */
               copia de endmen para DPE[pe].arm[ ];
               ajusta índices em arm[ ];
               if( esta mensagem não substituiu uma mais antiga )
                    V(DPE[pe].scont);
```

```
            V(DPE[pe].mutex);
        }
}
```

Se a semântica do envio assíncrono permitisse o bloqueio da tarefa que tenta enviar uma mensagem para uma porta "cheia", o código de `env.a()` seria o mostrado abaixo. Ao invés do armazenador `arm[ ]`, a caixa postal associada ao DPE seria criada com capacidade para várias mensagens. Note a simplicidade desta versão, quando comparada com aquela mostrada acima.

```
/* semântica diferente daquela do STR */
env.a(prtsaida, tammen, endmen)
int   prtsaida, tammen;
int   *endmen;
{
     int pe;
     int *carta;

     /* execução em exclusão mútua */
     while(não enviou para todas portas conectadas) {
          pe = indice do DPE da porta seguinte;
          carta = aloca(DPE[pe].onde, tammen);
          copia de endmen para carta;
          envia(DPE[pe].cpdest, carta);
     }
}
```

O envio de uma "carta" diferente para cada porta de entrada é necessário porque os DPEs destino podem estar em memória global e/ou local, dependendo da localização dos módulos a que pertencem. A devolução dos blocos de memória com as cartas deve ser feita na recepção das mensagens. Os semáforos `mutex` e `scont` são desnecessários neste caso porque `envia()` controla todo o sincronismo (exclusão mútua) envolvido no envio das cartas.

Envio Síncrono de uma Mensagem

`env.s()` aloca um bloco de memória da lista livre e copia a mensagem nele. Quando a mensagem de resposta é recebida, ela é copiada do armazenador temporário para o endereço especificado e o bloco de memória é devolvido à lista livre. Se a mensagem de resposta indicar a ocorrência de uma condição de erro, a cópia não é efetuada e a condição de erro é copiada no campo `status` do DPE.

```
env.s(prtsaida, tammen, endmen, endresp)
int   prtsaida;
int   tammen;
int   *endmen;
int   *endresp;            /* endereço de cópia da mensagem de resposta */
{
     int   pe;
     int   *carta;         /* aponta armazenador temporário da mensagem */

     /* execução em exclusão mútua */
     pe = índice do DPE conectado à prtsaida;
```

```
     carta = aloca(DPE[pe].onde, tammen);
     copia de endmen para carta;
     P(DPE[pe].mutex);    /* faz transação atômica */
          envia(DPE[pe].cpdest, carta);
          carta = recebe(DPE[pe].cpresp);
     V(DPE[pe].mutex);
     if( carta != ERRO )
          copia de carta para endresp;
     desaloca(DPE[pe].onde, carta, tammen);
     escreve resultado da transação em DPE[pe].status;
}
```

Note que o DPE da Camada Sete não possui um campo com o número da transação corrente. O uso do semáforo `mutex` faz a execução do `envia()` seguida do `recebe()`, uma operação atômica, desta forma garantindo a sincronização entre a tarefa que iniciou a transação e a que deve concluí-la. As duas caixas postais (`cpdest` e `cpresp`) têm capacidade para zero mensagens. A transação é iniciada pela execução do `envia()`) na caixa de destino das mensagens (`cpdest`); o emissor bloqueia-se então à espera da resposta (`recebe()` na `cpresp`, que deve estar vazia). As duas caixas postais são necessárias para evitar que o `envia()` e o `recebe()` processem a mesma mensagem.

Envio Síncrono com Cláusula para Tratamento de Falha

Esta primitiva tem praticamente o mesmo código que `env.s()`: no lugar do `recebe()` é executado `trecebe()` para garantir que a transação termina no período máximo estipulado. Se o período de espera pela resposta for excedido, esta condição é sinalizada no `status` do DPE. Abaixo, mostramos apenas a sintaxe desta primitiva já que o código é semelhante ao de `env.s()`.

```
env.s.temp(tempo, prtsaida, tammen, endmen, endresp)
int   tempo;              /* tempo máximo de espera */
int   prtsaida;
int   tammen;
int   *endmen;
int   *endresp;
```

Recepção Bloqueante de uma Mensagem

`rec()` decide se a recepção é de uma porta síncrona ou assíncrona. No caso da porta síncrona, se não houver nenhuma mensagem à espera na caixa postal, a tarefa fica bloqueada no `recebe()`; caso contrário, ela é copiada para o endereço indicado. No caso da porta assíncrona, o semáforo `scont` mantém o número de mensagens depositadas em `arm[ ]`. Se não houver nenhuma mensagem à espera de recepção, a tarefa fica bloqueada em `scont` até que alguma seja inserida em `arm[ ]`. Na comunicação síncrona, `rec()` retorna o endereço do armazenador temporário para que a mensagem de resposta seja retornada nele (o tamanho do armazenador deve ser compatível também com a mensagem de resposta).

```
rec(prtentr, endmen)
int   prtentr;                    /* identificador do DPE */
int   *endmen;                    /* endereço onde deve ser copiada a mensagem */
{
     int   *carta;                /* aponta mensagem recebida */
```

```
     /* execução em exclusão mútua */
     if( DPE[prtentr].tipo == SINCRONA ) {
          p = recebe(DPE[prtentr].cpdest)
          copia de carta para endmen;
          return(carta);
     }
     else {
          P(DPE[prtentr].scont);  /* bloqueia à espera de mensagens */
          P(DPE[prtentr].mutex);  /* exclusão no acesso à arm[ ] */
               copia de DPE[prtentr].arm[ ] para endmen;
               ajusta índices em arm[ ]
          V(DPE[prtentr].mutex);
     }
}
```

Se a semântica do envio assíncrono fosse aquela proposta no início desta seção, o código da primitiva de recepção seria aquele mostrado abaixo.

```
/* semântica diferente daquela do STR */
rec(prtentr, endmen)
int   prtentr;
int   *endmen;
{
     int   *carta;

     /* execução em exclusão mútua */
     carta = recebe(DPE[prtentr].cpdest);
     copia de carta para endmen;
     if( DPE[prtentr].tipo == SINCRONO )
          return(carta);
     else
          dealoca(DPE[prtentr].onde,carta,*carta);
          /* *carta contém o tamanho do armazenador temporário */
}
```

Envio de Mensagem de Resposta

O envio de uma mensagem de resposta completa uma transação iniciada por outra tarefa, que deve estar bloqueada à espera de uma resposta. Isto deve ser testado, bem como o estado da conexão entre as portas que formam o canal lógico. Se a outra tarefa não está bloqueada à espera ou o canal lógico foi interrompido, a execução de `resp()` não tem efeito algum.

```
resp(prtentr, tammen, endmen, carta)
int   prtentr;
int   tammen;
int   *endmen;
int   *carta;              /* aponta armazenador temporário da mensagem */
{
     /* execução em exclusão mútua */
     /* testa se a outra ponta da conexão está bloqueada à espera */
     if( transação em andamento ) {
          copia de endmen para carta;
```

```
            envia(DPE[prtentr].cpresp, carta);
        }
}
```

Desvio de uma Comunicação Síncrona

No STR, o comando FORWARD, implementado pela primitiva `desvia(prtentr, prtsaida)`, pode ser usado no lugar de REPLY para "terminar" uma transação. A seqüência de eventos numa transação com FORWARD é descrita a seguir. Suponha que o módulo `A` executou um envio síncrono para o módulo `B`. O módulo `B`, ao invés de terminar a transação com um REPLY, executa um FORWARD que reenvia a mensagem inalterada para o módulo `C`. Quando `C` executar o REPLY, a resposta será redirecionada, sendo entregue diretamente ao módulo `A`, sem intervenção de `B`. O redirecionamento é possível porque os DPEs do STR mantém as informações necessárias para isso sobre as transações correntes.

A semântica do `desvia()` é equivalente a uma execução de `env.s()` remetendo a mensagem pela porta de saída especificada, seguida de um `resp()`, enviando a resposta do `env.s()` ao módulo que iniciou a transação. O `desvia()` da camada sete é implementado desta maneira. As informações necessárias para o desvio da mensagem e a retransmissão da resposta são passadas a `desvia()` como parâmetros.

```
desvia(prtentr, prtsaida, tammen, endmen, tamresp, endresp)
int   prtentr;            /* porta de onde a mensagem original é recebida */
int   prtsaida;           /* porta para onde a mensagem original é desviada */
int   tammen;
int   *endmen;            /* aponta mensagem original */
int   tamresp;
int   *endresp;           /* aponta mensagem de resposta à original */
{
      int   *carta;

      carta = env.s(prtsaida, tammen, endmen, endresp);
      resp(prtentr, tamresp, endresp, carta);
}
```

A diferença desta implementação para aquela do STR é que, nesta, o módulo que contém o FORWARD toma uma parte ativa na retransmissão da resposta ao módulo que iniciou a transação (no exemplo acima, `B` faz a retransmissão da resposta de `C` para `A`).

Recepção Seletiva de uma Mensagem

Se o comando PSELECT contiver uma cláusula TIMEOUT, todas as cláusulas com RECEIVE são implementadas da mesma forma que `rec()`, exceto que, ao invés do `recebe()` na "parte síncrona" de `rec()`, é executado `trecebe()` com o intervalo de espera da cláusula TIMEOUT. Na "parte assíncrona" de `rec()`, é executado um `tP()` no lugar do primeiro `P()` e a coleta da mensagem só é efetuada se o semáforo `scont` for sinalizado, isto é, se não ocorrer o decurso de prazo. Se o comando não contiver a cláusula TIMEOUT, os comandos RECEIVE são implementados com `rec()`. Se o `trec()` escolhido para execução terminar por decurso de prazo, os comandos que seguem a cláusula TIMEOUT são executados e o PSELECT termina. O código que faz a seleção entre as cláusulas válidas é gerado pelo compilador LPM.

Cancelamento de uma Comunicação Síncrona

Da mesma forma que no envio de uma mensagem de resposta, a tarefa que iniciou a transação deve estar à espera de uma resposta e o canal lógico deve estar ativo. `resp.falha()` envia um código de erro, ao invés de um apontador para um bloco de memória.

```
resp.falha(prtentr, motivo)
int   prtentr;
int   motivo;        /* motivo do cancelamento */
{
     if( transação em andamento )
          envia(DPE[prtentr].cpresp, motivo);
}
```

Determinação da Razão de uma Falha

Esta função consite apenas do exame do valor de `status` do DPE através do qual ocorreu uma transação.

Teste de Conexão de uma Porta de Saída

Esta função testa o estado de um canal lógico, pelo exame da tabela de portas do BCM e retorna TRUE ou FALSE em função do resultado do teste.

Determinação da Quantidade de Mensagens Enfileiradas numa Porta de Entrada

O valor retornado por `mem.prt.entr()` indica o número de mensagens à espera de recepção numa porta.

```
mem.prt.entr(prtentr)
int   prtentr;
{
     if( DPE[prtentr].tipo == SINCRONA )
          return( examina(DPE[prtentr].cpdest) );
     else
          return( valorsem(DPE[prtentr].scont) );
}
```

Se a semântica do envio assíncrono fosse aquela do início desta seção, o código de `mem.prt.entr()` seria aquele mostrado abaixo.

```
/* semântica diferente daquela do STR */
mem.prt.entr(prtentr)
int   prtentr;
{
     return( examina(DPE[prtentr].cpdest) );
}
```

A.4 Coda

O que tentamos mostrar neste apêndice não foi apenas a implementação de um conjunto de primitivas de comunicação tendo por base semáforos e caixas postais. Nossa intenção é mostrar o grau de dificuldade a ser enfrentado na distribuição de um programa em vários processadores. A Transparência de Multiprocessamento simplifica substancialmente esta tarefa. Uma vez programada a configuração de um aplicativo, o multiprocessador "desaparece" da cena.

Apêndice B

Avaliação de Desempenho

Neste Apêndice, mostramos as medidas de tempo de execução das primitivas do NÚCLEO. Estas medidas foram efetuadas com o auxílio do *Trigger Trace Analyzer* (TTA), executando num sistema de desenvolvimento da Tektronix, composto pelos 8561–Multi-User Software Development Unit e 8540–Integration Unit (emulador). O 8561 suporta o desenvolvimento de programas com as ferramentas do UNIX (TNIX, no caso) e a depuração destes com o TTA.

Os processadores executam acessos ao código sem *wait states* (400 ns @ 10 MHz), a dados em memória local com um *wait state* (500 ns) e a dados em memória global, via VME, com quatro *wait states* (800 ns). As medições de tempo de execução foram levantadas com uma resolução de 200 ns; os valores resultantes foram arredondados para unidades de microssegundos.

Na maioria das primitivas que fazem acessos à memória global, as medições foram efetuadas sem tráfego no VME. Dessa forma, os valores obtidos referem-se somente à execução do código, sem a influência de eventuais conflitos de acesso. Nos casos em que a interferência de outro processador é necessária, sua utilização do barramento foi mantida no mínimo indispensável. Os valores medidos para a execução de primitivas que envolvem processos externos ao processador foram obtidos com latência zero no VME, isto é, a interrupção gerada pelo processador que executa a primitiva é atendida pelo processador onde reside o processo "alvo" num intervalo não maior que cinco microssegundos. Nas tabelas de valores, a coluna *VME* indica o intervalo máximo em que o acesso ao barramento global é bloqueado pelo processador que está executando a primitiva em questão (acessos em exclusão mútua à memória global).

Todas as medidas foram efetuadas com o escalonador funcionando em modo preemptivo e a relação de prioridades entre os processos é tal que sempre ocorre uma troca de contexto (medidas de pior caso). Quando a execução de uma primitiva provoca uma troca de contexto, o valor medido indica o tempo decorrido desde a primeira instrução do código da primitiva até a última instrução da função que faz a troca de contexto. Dessa forma, a próxima instrução a ser executada já pertence ao contexto do processo que recebeu a posse do processador. Estes casos são indicados pela presença de um "†" no comentário que acompanha a medida. Em modo não preemptivo, o tempo de execução das primitivas pode ser obtido a partir dos valores apresentados, subtraindo-se o tempo necessário para a troca de contexto.

Em algumas das medidas, o tempo de execução é mostrado como a soma de dois valores. A primeira parcela refere-se à execução do código que precede o bloqueio do processo e à troca de contexto associada. Quando o processo retomar a posse do processador, o

restante do código da primitiva é executado durante o intervalo constante da segunda parcela. Estes casos são indicados pelo "#" no comentário que acompanha a medida. Nas primitivas executadas remotamente, o tempo de execução é a soma de dois valores. A primeira parcela representa o tempo de execução local da primitiva; a segunda parcela mostra o tempo de execução da primitiva no outro processador. O tempo de execução no outro processador também é o valor de pior caso. Esta situação é indicada pelo "¶" no comentário.

B.1 Gerenciamento de Memória

O valor medido é o da requisição e devolução de um bloco de memória. O tamanho do bloco requisitado é menor ou igual ao do primeiro bloco na lista livre (este é o melhor caso).

primitiva	comentário	μs	VME
`aloca()`	memória local	118	
	memória global	143	52
`dealoca()`	memória local	149	
	memória global	176	72

B.2 Filas de Processos

As funções desta camada são de uso exclusivo das primitivas do NÚCLEO. O tempo de execução destas funções é uma fração do tempo de execução das primitivas.

B.3 Processos

As medidas relativas às funções que implementam o escalonador são apresentadas a seguir. A função `ctxsw()` faz a troca de contexto entre dois processos, isto é, salva os registradores do processo corrente na sua própria pilha e carrega os valores dos registradores do novo processo, a partir da pilha do novo processo. A função `reescal()` escolhe o processo que vai receber a posse do processador; a primitiva `troca()` força uma troca de contexto.

primitiva	comentário	μs
`ctxsw()`		44
`reescal()`	processo corrente continua executando	57
	proc. corr. inserido na fila pronto	244
	proc. corr. muda de estado	150
`troca()`	tempos de `reescal()` + 24 μs	

As medidas das primitivas de manipulação de processos são mostradas abaixo. Os inúmeros valores relativos a `mata()` decorrem da possibilidade de um processo ser morto em qualquer estado em que se encontre.

primitiva	comentário	μs	VME
`cria()`	criação lc. de processo	$497 + 1.5$/param.	
	criação remota de proc.¶	$1244 + 595 + 26$/param.	
`mata()`	processo corrente†	422	
	pr. bloqueado em sem. lc.	350	
	pr. bloqueado em sem. glb.	368	34
	pr. dormindo	348	
	pr. temporizado em sem. lc.	400	
	pr. temporizado em sem. glb.	421	34
	pr. na fila de pronto	352	
	pr. externo¶	162 + `mata()`	
`ativa()`	com reescalonamento†	382	
	sem reescalonamento	210	
	proc. externo¶	162 + `ativa()`	
`suspende()`	processo corrente†	202	
	outros procs. (fila de pronto)	114	
	proc. externo¶	162 + `suspende()`	
`queprio()`		54	
`mudaprio()`	provoca reescalonamento†	301	
	não provoca reescalonamento	128	
	proc. remoto¶	162 + `mudaprio()`	
`meuid()`		43	
`publica()`	insere na tabela vazia*	657	160
	tabela cheia* – retorna erro	1714	25

* Tabela com capacidade para 11 elementos.

B.4 Sincronização Interprocessos

As medidas referentes às primitivas das camadas 4, 5 e 6 mostram os valores resultantes das possíveis combinações de estados dos objetos envolvidos: estado dos processos, tipo de semáforo, quantidade de mensagens enfileiradas numa caixa postal, etc...

O `tratador()` é o procedimento que trata a interrupção provocada por outro processador indicando a liberação de um processo interno ao processador que atende à interrupção. O comportamento do `tratador()` é similar à execução de `V(s)`: o processo sinalizado é inserido na fila de pronto e o escalonador é invocado.

primitiva	comentário	μs	VME
`P()`	s. lc., proc. fica bloqueado†	276	
	s. lc., proc. não bloqueia	86	
	s. glb., proc. fica bloquedo†	358	97
	s. glb., proc. não bloqueia	100	35
`V()`	s. lc., libera processo†	508	
	s. lc., não libera processo	88	
	s. glb., libera processo†	618	129
	s. glb., não libera processo	102	36
	s. glb., libera proc. externo¶	273 + 471	64
	s. lc., proc. temporizado†	592	
	s. glb., pr. interno temporizado†	700	129
	s. glb., pr. externo temporizado¶	684 + 471	64
`tP()`	s. lc., proc. bloqueia†	389	
	s. lc., não bloqueia	88	
	s. glb., proc. bloqueia†	478	93
	s. glb., proc. nao bloqueia	103	36
`criasem()`	semáforo em mem. lc.	88	
	semáforo em mem. glb.	102	45
`matasem()`	s. lc., valor positivo	94	
	s. lc., um proc. bloqueado†	586	
	s. glb., valor positivo	139	58
	s. glb., um pr. interno bloq.†	739	147
	s. glb., um pr. externo bloq.¶	308 + 471	147
`semreset()`	s. lc., valor positivo	106	
	s. lc., um pr. bloqueado†	598	
	s. glb., valor positivo	151	56
	s. glb., um pr. interno bloq.†	751	146
	s. glb., um pr. externo bloq.¶	310 + 471	146
`valorsem()`	semáforo local	75	
	semáforo global	88	29
`multiV()`	s. local, valor positivo	141	
	s. lc., um proc. bloqueado†	591	
	s. global, valor positivo	175	31
	s. glb., um pr. interno bloq.†	717	115
	s. glb., um pr. externo bloq.¶	232 + 471	162
`tratador()`	†	471	72
`dorme()`	faz pr. corr. "hibernar"†	309	
`acorda()`	processo dormindo†	537	
	pr. temporizado, sem. lc.†	537	
	pr. temporizado, sem. glb.†	565	64

B.5 Armazenadores

primitiva	comentário	μs	VME
`criaarm()`	cria armazenador em mem. lc.	359	
	cria armazenador em mem. glb.	413	147
`reqarm()`	m. lc., pr. não fica bloqueado	190	
	m. lc., pr. fica bloqueado#†	305 + 80	
	m. glb., pr. não fica bloqueado	204	36
	m. glb., pr. bloqueado#†	385 + 82	97
`libarm()`	m. lc., não libera pr. bloqueado	188	
	m. lc., libera proc.†	595 + 94	
	m. glb., não libera pr. bloqueado	204	25
	m. glb., libera proc interno †	706	98
	m. glb., libera proc. externo¶	508 + 568	64

B.6 Comunicação Interprocessos

primitiva	comentário	μs	VME
`criacp()`	cria caixa postal em m. lc.	306	
	cria caixa postal em m. glb.	351	52
`destroicp()`	m. lc., nenhum pr. bloq.	327	
	m. lc., um pr. bloqueado#†	760 + 33	
	m. glb., nenhum pr. bloqueado	430	58
	m. glb., um pr. interno bloq.#†	967 + 33	147
	m. glb.,um pr. externo bloq.#¶	636 + 471 + 33	147
`resetcp()`	m. lc., nenhum pr. bloqueado	380	
	m. lc., um pr. bloqueado#†	808 + 33	
	m. glb., nenhum pr. bloq.	488	26
	m. glb., um pr. interno bloq.#†	1023 + 33	146
	m. glb., um pr. externo bloq.#¶	670 + 471 + 33	146
`examcp()`	caixa postal em memória lc.	109	
	caixa postal em memória glb.	112	29
`envia()`	m. lc., pr. não fica bloqueado	340	
	m. lc., pr. fica bloqueado#†	355 + 176	
	m. lc., libera pr. bloqueado†	734	
	m. glb., pr. não bloqueia	370	57
	m. glb., pr. fica bloqueado#†	431 + 192	97
	m. glb., libera pr. interno†	858	129
	m. glb., libera pr. externo¶	533 + 471	64
`recebe()`	m. lc., proc. não fica bloqueado	338	
	m. lc., pr. fica bloqueado#†	358 + 176	
	m. glb., pr. não fica bloqueado	372	55
	m. glb., pr. fica bloqueado#†	432 + 200	97

B.7 Tempo de Execução do Configurador do Capítulo 5

Nesta seção, apresentamos a medida do tempo de execução do Configurador mostrado no Capítulo 5. O código que foi "medido" é mostrado abaixo. O programa `prog2()` foi previamente publicado pelo Configurador do processador onde seu código reside; na criação do processo `proc2`, apenas o nome de `prog2()` (a cadeia `"prog2"`) é explicitado, e não a sua localização.

```
configurador()
{
     int   s1, s2;
     int   pid1, pid2, pid3;
     int   prog1(), prog2(), prog3();

     s1 = criasem(LOCAL,0);
     s2 = criasem(GLOBAL,2);
     pid1 = cria(LOCAL,NULL,prog1,TPILHA,PRIO20,"proc1",2,s1,s2);
     pid2 = cria(GLOBAL,"prog2",NULL,TPILHA,PRIO20,"proc2",1,s2);
     pid3 = cria(LOCAL,NULL,prog3,TPILHA,PRIO20,"proc3",1,s1);
     mudaprio(meuid(),PRIO50);      /* evita reescalonamento */
     ativa(pid1);
     ativa(pid2);
     ativa(pid3);
     mudaprio(meuid(),PRIO10);      /* provoca reescalonamento */
}
```

O tempo de execução deste Configurador, desde a execução da sua primeira instrução até a última instrução de `ctxsw()` (invocada pelo segundo `mudaprio()`) é de 2,26 milissegundos.

B.8 Sintonia Fina

Estes valores para os tempos de execução foram obtidos após uma pequena revisão no código de algumas das funções internas ao NÚCLEO. A função que faz a troca de contexto (`ctxsw()`) foi reescrita: seu tempo de execução caiu de 77 para 44 microssegundos. O código gerado pelo compilador para as funções `enfileira()`, `retira()`, `insere()` foi otimizado manualmente. Os ganhos neste caso ficaram entre 40% (`insere()`) e 30% (`enfileira()` e `retira()`). O reflexo disso foi uma melhora da ordem de 8% nos valores medidos das demais primitivas.

Ganhos significativos[1] podem ainda ser obtidos pela otimização do código objeto de outras funções da camada dois (`primeiro()` e `ultimo()`) e das primitivas `P()` e `V()`. Esta revisão possui uma boa relação/custo benefício devido à freqüência com que estas funções e primitivas são usadas tanto interna quanto externamente ao NÚCLEO. Uma pequena dificuldade envolvida neste caso, é a importação dos valores definidos nos arquivos *header* (por `#define`s) pelos módulos codificados em *assembly* (em que a definição de constantes é por `equates`). Isto pode ser facilmente resolvido pela inclusão nos *Makefile*s de *shell scripts* [KERN84] para controlar a conversão dos `#define`s em `equ`s.

[1] De 20 a 40% em alguns casos.