UM AMBIENTE PARA PROGRAMAÇÃO CONCORRENTE EM TURBO PASCAL

Carlos Fernando R. de Avila Goulart

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS DE COMPUTAÇÃO.

Aprovada por:

Prof. Valmir Carneiro Barbosa, Ph.D.
(Presidente)

Prof. Edil S. Tavares Fernandes, Ph.D.

Prof. Adriano Joaquim de O. Cruz, Ph.D.

RIO DE JANEIRO, RJ - BRASIL
OUTUBRO DE 1991

GOULART, CARLOS FERNANDO RODRIGUES DE AVILA

Um Ambiente Para Programação Concorrente em Turbo Pascal [Rio de Janeiro] 1991

XI, 160 pp. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas de Computação, 1991)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Programação Concorrente I. COPPE/UFRJ II. Título (série)

"Buscai, em primeiro lugar, o Reino de Deus e a sua justiça, e todas as coisas vos serão acrescentadas"

Mt 6,33

Ao Senhor, fonte inesgotável da Sabedoria.

À minha esposa Carla e ao meu filho Pedro, meus amores.

AGRADECIMENTOS

Ao Senhor, por me dar a graça de terminar este trabalho.

Ao meu orientador, Valmir, pelo grande apoio e pela amizade que cultivamos durante este período.

Ao Laboratório de Matemática Aplicada, no Instituto de Matemática da UFRJ, pelas horas de equipamento que consumi durante a Tese.

Ao CNPq e à FAPERJ, pelo apoio financeiro dado.

À Denise, Cláudia e Ana Paula, secretárias acadêmicas do Programa de Sistemas, pela amabilidade e soluções dos meus problemas.

Aos meus pais, por terem me dado condições e educação o suficiente para chegar até aqui, e pela cultura que recebi, herança maior (e única...) que posso ter.

Finalmente, à minha esposa Carla e ao meu filho Pedro, razões primeiras de se procurar um futuro melhor, pelas alegrias que me dão a cada dia, enchendo a minha vida de amor.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para obtenção do grau de Mestre em Ciências (M.Sc.).

UM AMBIENTE PARA PROGRAMAÇÃO CONCORRENTE EM TURBO PASCAL

Carlos Fernando R. de Avila Goulart
Outubro, 1991

Orientador: Valmir Carneiro Barbosa
Programa : Engenharia de Sistemas de Computação

O presente trabalho visa fornecer à comunidade acadêmica, em forma de software, um ambiente propício para a prática de Programação Concorrente. O ambiente é constituído de quatro módulos: Kernel (o núcleo de gerência de processos), ShareMem (o módulo de memória compartilhada), MsgPass (o módulo de troca de mensagens) e Debugger (o depurador para programas concorrentes). O equipamento alvo escolhido foram os microcomputadores IBM-PC compatíveis, pela sua enorme difusão e relativo baixo custo. A linguagem de programação é o Turbo Pascal, pela sua grande popularidade, clareza própria do Pascal, velocidade de compilação e execução e suas facilidades de depuração. O princípio determinante destas escolhas foi a de popularizar a prática da Programação Concorrente, descartando portanto equipamentos e linguagens especializados, de uso difícil ou de acesso restrito.

A proposta da Tese é fornecer ao usuário a implementação dos principais mecanismos e estruturas para Programação Concorrente encontrados na literatura. Desta forma, em um só ambiente podemos encontrar características diversas e por vezes antagônicas. A finalidade desta abordagem é abrir um leque razoável de opções, de maneira que mais aplicações possam ser programadas, não restringindo demasiadamente o usuário a modelos fechados.

Abstract of Thesis presented to COPPE/UFRJ as partial fulfillment of the requirements for the degree of Master of Science (M.Sc.).

AN ENVIRONMENT FOR CONCURRENT PROGRAMMING IN TURBO PASCAL

Carlos Fernando R. de Avila Goulart

October, 1991

Thesis Supervisor: Valmir Carneiro Barbosa

Department : Computing Systems Engineering

The present work intends to bring to the academic community, in a software package, a propitious environment for practicing Concurrent Programming. The environment is made up of four modules: Kernel (the process manager nucleus), ShareMem (the shared memory module), MsgPass (the message passing module), and Debugger (the debugger for concurrent programs). The target machines chosen are IBM-PC compatible microcomputers, for their widespread use and relative low cost. The programming language is Turbo Pascal, for its enormous popularity, Pascal's clarity, compilation and execution speed and its debugging facilities. The fundamental motivation for this choice was to make popular the practice of Concurrent Programming, therefore discarding machines and languages too specialized, difficult to use or of restricted access.

The thesis' proposal is to give the user an implementation of the main mechanisms and structures for Concurrent Programming found in the literature. So in a single environment we can find characteristics diverse from one another and sometimes opposite as well. The purpose of this approach is to open up a good number of options, allowing more applications to be programmed, not restricting the user to closed models.

I N D I C E

CAPITULO V: O MÓDULO DE TROCA DE MENSAGENS

CAPITULO VI: O MÓDULO DEBUGGER

CAPITULO I: INTRODUÇÃO

O presente trabalho visa fornecer à comunidade acadêmica, em forma de software, um ambiente propício para a prática de Programação Concorrente. Uma das motivações do projeto é o fato de que, ainda hoje, mesmo após três décadas do advento deste paradigma de programação, há uma carência muito grande de ferramentas para a construção de programas concorrentes. É notória a quantidade de trabalhos publicados sobre o aspecto teórico do assunto, mas a falta de software adequado ou, por outro lado, a confecção de software específico para máquinas de aquisição restrita impedem uma maior difusão do modelo. Aliam-se a estes fatores a idéia mal-concebida da Programação Concorrente ser útil apenas na construção de sistemas operacionais e do assunto ser tratado com descaso nos cursos de graduação em computação no país.

Com o crescente interesse do mundo tecnológico pelos supercomputadores, a Programação Concorrente, na sua forma de paralelismo dito real, por envolver mais de um processador em computações simultâneas, tem ganho mais espaço em pesquisas e produtos. Além disso, com a queda do custo do hardware e o não acompanhamento da evolução por parte da área de software, faz-se necessário começar cada vez mais a administrar melhor os recursos disponíveis em um equipamento, principalmente o tempo do processador.

Mesmo em máquinas uniprocessadas, onde não temos paralelismo real, os conceitos da Programação Concorrente podem ser vastamente aplicados. O exemplo mais óbvio é a confecção de um sistema operacional, mas existem outros como a programação de sistemas de tempo real, engenharia de software, a melhoria da eficiência de um algoritmo (BEN-ARI, 1982) e a abordagem bem mais simples que a versão seqüencial de um mesmo problema, como em (HORTON, 1986).

Para escrever programas concorrentes é necessário, naturalmente, uma linguagem com recursos que permitam que mais de um trecho de programa (melhor dizendo, processo) coexista e execute simultaneamente. A noção de simultaneidade aqui é flexível, não importando se os processos existentes rodam em processadores diferentes (paralelismo real) ou partilham o tempo de um mesmo processador (pseudo-paralelismo). A maioria esmagadora das linguagens hoje existentes é dita seqüencial e não dispõe destes recursos. As linguagens ou ambientes que permitem concorrência são basicamente de três tipos:

1) Linguagens projetadas realmente visando o suporte de concorrência;

2) Linguagens redefinidas por alteração sintática de uma linguagem seqüencial, seja mudando o compilador ou fazendo-a passar por um pré-processador;

3) Linguagens estendidas através de bibliotecas de rotinas para este fim.

No primeiro caso encontram-se linguagens como *Edison* (BRINCH HANSEN, 1981; DUBNICKI, 1988), *Joyce* (BRINCH HANSEN, 1987a, 1987b), *CSP* (HOARE, 1978; HULL, 1986; CERRADA, 1988; WRENCH, 1988), *ADA* (U.S. Dept. of Defense, 1981; LEDGARD, 1983), *Occam* (INMOS, 1984), *Modula* (WIRTH, 1977), *SR* (ANDREWS, 1982a) e *BSP* (GEHANI, 1984), entre outras. Todas estas linguagens foram projetadas com suporte para concorrência fazendo parte de sua sintaxe, realizada de diversas formas. Como compêndio de diversas linguagens e seus mecanismos, recomenda-se ANDREWS (1983).

As maiores vantagens das linguagens desse primeiro tipo são a elegância das construções, a possibilidade de uma checagem semântica a tempo de compilação e uma melhor otimização do código gerado. Suas maiores desvantagens são justamente a falta de

flexibilidade de se incluir novas construções e a pouca difusão obtida, por serem demasiado específicas e não permitirem aplicações gerais.

No segundo caso estão linguagens como *Concurrent Pascal* (BRINCH HANSEN, 1975), *Pascal-Plus* (WELSH, 1979), *Path Pascal* (KOLSTAD, 1980), *Concurrent Pascal-S* (BEN-ARI, 1982), *Concurrent C* (TSUJINO, 1984; GEHANI, 1986, 1989) e *Concurrent C++* (GEHANI, 1988). Estas linguagens foram adaptadas a partir das versões originais do Pascal (WIRTH, 1977) e C (KERNIGHAN, 1978) para incorporarem recursos de concorrência na sua sintaxe estendida. A maior vantagem dessa abordagem é justamente aproveitar a enorme popularidade das linguagens de onde elas foram adaptadas, tornando desnecessário o aprendizado de uma sintaxe completamente nova e principalmente aproveitar todo o material já existente escrito nas linguagens-base. De qualquer forma, ainda sofre da limitação de suas construções não poderem ser mais estendidas e da necessidade de um novo compilador, a não ser que ela gere um código intermediário (seria um pré-processador) para um compilador da linguagem-base, o que introduz mais um passo na compilação e com isso perda de tempo. Neste último caso temos como exemplo *Parallel Pascal* (BEAUMONT, 1978).

O terceiro tipo de abordagem na verdade não cria mais uma linguagem concorrente ou estende uma sintaxe, mas sim adiciona-se às rotinas oferecidas pela mesma, através de uma biblioteca de rotinas. Além de ser a metodologia mais fácil e rápida, é considerada por vários autores a mais moderna. Podemos citar KRIZ (1980), BERRY (1986), HORTON (1986), BERGMANN (1986), KRISHNAMOORTHY (1987), LINDLEY (1987), CORMACK (1988), VASCONCELOS (1988, 1989), DUGLOSZ (1988, 1989) e GREEN (1989). Todos esses trabalhos acrescentaram núcleos de suporte à concorrência em linguagens populares como Pascal, C e C++, à semelhança do segundo tipo já descrito. Com isso conseguem uma maior difusão, o aproveitamento total dos compiladores já exis-

tentes e também uma enorme flexibilidade, uma vez que os procedimentos das bibliotecas podem ser alterados e novas estruturas incluídas e/ou excluídas sem nenhuma alteração no compilador. Desta forma a "nova" linguagem não fica atrelada a nenhum modelo específico.

Sua maior desvantagem é, em muitos casos (mas nem todos), a deselegância de alguns comandos, que por não fazerem parte da linguagem têm que ser adaptados, causando prejuízo para a compreensão do programa ou não podendo ser checados semanticamente pelo compilador em tempo de compilação.

Outra diferenciação entre as diversas linguagens existentes para Programação Concorrente é quanto ao modelo de comunicação adotado. São dois os principais modelos:

1) Modelo de Memória Compartilhada;
2) Modelo de Troca de Mensagens.

O primeiro modelo permite que os processos de um programa concorrente troquem informações utilizando variáveis comuns, ou seja, fazendo acesso às mesmas posições de memória. Este modelo é especialmente utilizado em sistemas em que todos os processadores (se mais de um) podem acessar toda a memória, dita, neste caso, memória centralizada (ANDREWS, 1983).

Já o segundo modelo só permite aos processos trocarem informações através de estruturas especiais, denominadas em geral canais, portas ou "mailboxes". Não é o próprio processo que acessa a região de memória onde ele quer armazenar um dado, mas sim a estrutura em questão. Este modelo é geralmente mais utilizado em sistemas em que os processadores não têm acesso a toda a memória, como no caso de sistemas distribuídos (BAL, 1989). Mesmo assim, seu uso tem crescido bastante até mesmo em sistemas que possuem memória centralizada, pois muitos autores acreditam ser o

melhor em todos os casos.

A proposta desta Tese é fornecer um ambiente, desenvolvido na forma de bibliotecas de rotinas, para ser utilizado junto com o software Turbo Pascal, por si só uma extensão do Pascal padrão de Wirth (BORLAND 1989a, 1989b). A escolha da abordagem, ou seja, extensão da linguagem através de bibliotecas, segue a tendência moderna e se respalda nas vantagens expostas anteriormente. A escolha da linguagem, que basicamente se dividiu entre C e Pascal, recaiu sobre a última pela sua maior clareza, maior utilização nos cursos de graduação e pela versão do software utilizado ter recursos bastante desejáveis, como módulo de rotinas pré-compiladas, checagem estrita de tipos de dados, programação orientada a objeto, inserção direta de código de máquina, etc. O uso de linguagem Assembly foi descartado de imediato, por consumir mais tempo de codificação e depuração, além do Turbo Pascal poder suprir todas as necessidades desejadas. A argumentação para o uso de linguagens de alto nível na construção de um núcleo para Programação Concorrente pode ser encontrada também em HOPPE (1980).

Outra escolha deste trabalho foi o equipamento alvo, no caso microcomputadores PC-compatíveis rodando o sistema operacional DOS. Isto se deve ao grande parque já instalado destas máquinas tanto no Brasil como no resto do mundo, o que permite ao ambiente rodar e ser útil em diversos lugares. A contrapartida a esta escolha seria utilizar estações de trabalho rodando o sistema operacional Unix ou até mesmo máquinas paralelas, mas sua utilização seria muito mais limitada. Tanto uma quanto a outra são de custo proibitivo e portanto de aquisição restrita, enquanto micros PC-compatíveis rodando Unix também são em bem menor número do que os que rodam DOS.

O ambiente foi desenvolvido para rodar em um micro PC com a configuração mais usual do país: XT

(microprocessadores Intel 8086 ou 8088) com 640k Bytes de memória RAM, um drive e qualquer placa de vídeo (monocromática ou colorida). É sabido que os processadores mais novos como o 80286, 80386 e 80486 suportam recursos que auxiliam na construção de ambientes deste gênero, mas se os utilizássemos estaríamos recaindo nos mesmo problema de máquinas de alcance restrito.

Mas, será válido criar um ambiente para Programação Concorrente em uma máquina uniprocessada? Ora, os conceitos envolvidos neste paradigma não estão completamente atrelados ao número de processadores envolvidos, e portanto podemos validar a maior parte dos problemas executando em um sistema uniprocessado com escalonamento de tempo entre os processos antes de executá-lo num sistema multiprocessado (VASCONCELOS, 1989). De qualquer forma, o conceito de processo transcende o de paralelismo real e podemos fazer programas para máquinas uniprocessadas perfeitamente bem.

Ambientes deste gênero já existem há algum tempo, mesmo para micros PC-compatíveis. Podemos citar os trabalhos de LINDLEY (1987), VASCONCELOS (1988, 1989), DUGLOSZ (1988, 1989), BOWLING (1989) e GREEN (1989), entre outros. O que diferencia o aqui apresentado dos demais é a diversificação de recursos: o usuário tem várias formas de criar e destruir processos; suspensão e continuação da execução de um processo; filas de prioridade; mudança de "time-slice"; co-rotinas; conceito de processador virtual, para simular sistemas paralelos ou distribuídos, inclusive com migração de processos; suporte dos dois modelos de memória (compartilhada e troca de mensagens) com as primitivas mais usadas e mais modernas sobre eles; um depurador on-line específico para os processos concorrentes, aliado ao já existente no Turbo Pascal; "timeouts", "delays" e escalonamento em determinada hora; comando de execução não-determinística; mensagens de erro amigáveis; facilidade de uso das construções; liberação do

usuário de várias tarefas, como garantir exclusão mútua no uso do DOS e em operações de Entrada e Saída; salvamento do contexto do co-processador aritmético em programas matemáticos; etc. Todos os recursos foram implementados de maneira eficiente e de modo a serem relativamente fáceis de se alterar.

Esta filosofia do ambiente, aliada à já discutida facilidade de acesso aos micros PCs, dão ao ambiente um caráter bastante genérico. Não é o ambiente que determina ao usuário que construções ou modelo de sistema e de memória utilizar, mas sim ele próprio, escolhendo dentre as que são mais adequadas ao seu caso. Poder-se-ia argumentar contra a idéia com o tamanho dos programas gerados, mas o "linker" do Turbo Pascal remove todo o código não-utilizado. Outra argumentação seria a dificuldade de aprendizado do software, dadas tantas características. Os trabalhos vistos durante a pesquisa sempre optaram por um ou outro modelo, o que os torna mais rápidos de serem completamente dominados, mas também com um potencial bem mais reduzido. Contudo, neste ambiente nem há tantas novas construções a ponto de ser uma nova linguagem, nem tampouco isso inibe o usuário de fazer bom e rápido uso dele, pois, como em qualquer software, basta saber o essencial e o mínimo desejado para atender a um dado objetivo.

O ambiente é composto de quatro módulos, a saber:

1) O Kernel, que é o núcleo do ambiente e realiza toda a gerência dos processos, além de outras funções;

2) O Módulo de Memória Compartilhada, onde são definidas primitivas para comunicação e sincronização entre processos através de variáveis comuns;

3) O Módulo de Troca de Mensagens, onde são definidas primitivas para comunicação e sincronização entre

processos através de trocas de mensagens;

4) O Depurador, que ajuda na visualização das filas de processos e seus blocos de controle.

O restante deste texto está dividido da seguinte forma: o Capítulo II apresenta uma introdução à Programação Concorrente, de maneira que o ambiente possa ser usado com um razoável entendimento da teoria; o Capítulo III descreve o Módulo Kernel; O Capítulo IV trata do Módulo de Memória Compartilhada; o Capítulo V trata do Módulo de Troca de Mensagens; o Capítulo VI mostra como usar o Depurador e os problemas inerentes à depuração de programas concorrentes.

Em anexo seguem a ficha técnica do software (Anexo I), uma cópia da interface (cabeçalhos das rotinas e declaração de constantes, tipos e variáveis exportadas) dos quatro módulos (Anexos II, III, IV e V), explicações das mensagens de erro do módulo Kernel (Anexo VI) e do módulo MsgPass (Anexo VII), um exemplo de utilização do Kernel (Anexo VIII), medidas de performance dos mecanismos de exclusão mútua de ShareMem (Anexo IX) e telas do módulo Debugger (Anexos X e XI). Por pretender uma maior divulgação deste trabalho e também de toda a nomenclatura da ciência da computação ser mais comum na língua inglesa, todas as construções e comentários nos programas-fonte são em inglês.

Uma última observação é quanto ao nome do ambiente. Como seu principal módulo é o chamado "Kernel", vamos nos referir a partir de agora ao ambiente simplesmente como Kernel. Onde isto causar ambigüidade com o módulo em si uma ressalva será feita.

CAPITULO II: INTRODUÇÃO À PROGRAMAÇÃO CONCORRENTE

Neste capítulo serão discutidos os tópicos mais relevantes para a compreensão e confecção de programas concorrentes. Boas referências sobre o aspecto teórico são PETERSON (1985) e BEN-ARI (1982). Quanto à implementação de núcleos para concorrência, ver COMER (1984), TANEMBAUM (1987) e MILENKOVIC (1987).

II.1 O que é Programação Concorrente

Um programa concorrente se caracteriza pela existência de mais de um processo rodando ao mesmo tempo em um ou mais processadores, onde processo é um trecho de programa executado de forma seqüencial. Para ilustrar essa definição, podemos imaginar um programa que copia o conteúdo de um arquivo-texto para um outro arquivo, lendo e escrevendo linha por linha de texto. Se considerarmos dois procedimentos "Le_Linha" e "Escreve_Linha", onde o primeiro opera sobre o arquivo-fonte e o segundo sobre o arquivo-destino, o programa seria da seguinte maneira:

```
While not Fim_de_Arquivo do begin
   Le_Linha;
   Escreve_Linha;
End;
```

Até aqui temos um programa seqüencial. Se pudéssemos fazer com que Le_Linha e Escreve_Linha fossem executados ao mesmo tempo, teríamos o programa original dividido em dois trechos de código, sendo as instruções dos dois procedimentos executadas de forma entrelaçada. Utilizando uma sintaxe estendida podemos escrever:

```
While not Fim_de_Arquivo do
   Cobegin
      Le_Linha;        Also
      Escreve_Linha;
   Coend;
```

Esta sintaxe se baseia nos trabalhos de DIJKSTRA (1965), no que se refere à estrutura Cobegin/Coend (originalmente Parbegin/Parend) e de BRINCH HANSEN (1981), no tocante ao Also. A estrutura Cobegin/Coend atua tal qual o Begin/End da sintaxe usual do Pascal, delimitando blocos de comandos. A diferença básica é que num Cobegin/Coend os comandos podem ser executados de forma concorrente, isto é, como processos. A palavra Also serve como separador dos processos, quer dizer, em cada Also termina a especificação de um processo dentro de um Cobegin/Coend. Não é necessário um Also no último processo do bloco, uma vez que o próprio Coend faz esse papel.

Desta forma, temos o programa original dividido em dois processos que rodam simultaneamente. Isto significa que, ao mesmo tempo em que uma linha está sendo lida do arquivo-fonte, uma linha pode estar sendo escrita no arquivo-destino. Esta noção de simultaneidade já foi discutida na introdução deste trabalho, sendo considerada em ambiente uniprocessados o entrelaçamento de instruções de máquina: durante um curto perído de tempo o processador executa instruções do primeiro procedimento, sendo rapidamente chaveado para o segundo; este executa e faz o mesmo, voltando ao primeiro; isto se repete até os procedimentos acabarem, e dá a ilusão de os dois estarem sendo executados "ao mesmo tempo".

Para gerenciar a existência de processos é necessário um núcleo de gerência de processos. Ele é o módulo básico que manipula todas as estruturas relativas à implementação da concorrência, deixando o usuário com uma visão de alto nível em relação a um programa concorrente.

II.2 Visão Interna de um Processo

Todo processo possui um "thread", isto é, um endereço de programa corrente e um fluxo de execução que são independentes dos demais processos existentes. Para

realizar tal empreendimento, ele deve possuir valores de registradores e uma pilha próprios, de maneira que, ao ser interrompido para ceder o processador a outro processo, ele possa salvar os registradores nesta pilha para uma posterior retomada da execução. Esta pilha, além disso, serve para a usual alocação de variáveis locais e armazenamento de valores temporários, tal qual a pilha de um programa seqüencial. Desta forma, para todos os efeitos, é como se um processo rodasse continuamente sem interrupção. Ao ser suspenso, todo o chamado *contexto* é salvo; ao voltar a executar, ele é completamente restaurado.

É importante ressaltar, portanto, que variáveis locais alocadas quando da entrada em um procedimento são exclusivas de um único processo, uma vez que cada um deles tem sua própria pilha. Conseqüentemente, é possível executar um mesmo procedimento simultaneamente por mais de um processo, desde que aquele seja *reentrante*. Um procedimento reentrante é aquele que não utiliza variáveis globais, podendo ser executado por mais de um processo ao mesmo tempo e exibindo a todos eles as mesmas condições.

Para armazenar informações pertinentes a cada processo, existe uma estrutura manipulada exclusivamente pelo núcleo de gerência de processos, comumente chamada de *PCB (Process Control Block - Bloco de Controle do Processo)*. Em um PCB estão guardadas informações diversas, como por exemplo o estado de um processo, sua prioridade, suas relações hierárquicas, seu número de filhos, seu encadeamento em alguma lista do núcleo e quaisquer outras que sejam individuais a ele.

Além disso, todo processo deve possuir uma identificação perante o núcleo de gerência de processos, de forma que seu PCB possa ser localizado em determinadas operações. Ela é conhecida normalmente como *PID (Process IDentification - Identificador de Processo)*.

II.3 Relações Hierárquicas

Todos os processos em um programa se relacionam de alguma maneira. A partir do programa principal, processos podem ser criados, esses podem criar outros e assim por diante. Desta forma temos uma família de processos, representada por um estrutura de árvore enraizada, onde a raiz é o programa principal (por si o primeiro processo). Um processo P é dito pai de Q quando P criou Q, sendo Q por sua vez filho de P. Se P criou Q e R, nesta ordem, então estes são irmãos, sendo Q o primogênito de P e R o caçula de P.

II.4 Estados de um Processo

Dependendo da implementação de cada núcleo, um processo pode assumir vários estados durante a sua existência. Os mais comuns são Running (processo usando o processador), Ready (processo pronto para rodar), Blocked (processo bloqueado), Suspended (processo suspenso), Halted (processo parado), Timing (processo em temporização) e Deadlocked (processo envolvido em deadlock).

II.5 Operações com Processos

Relacionamos a seguir as operações com processos mais comuns encontradas na literatura.

a) Criação: um processo pode criar outro a qualquer momento de sua execução, estabelecendo as relações hierárquicas abordadas na seção II.3. A criação de um processo envolve o estabelecimento de uma nova pilha, um novo PCB e um novo PID, para seu gerenciamento.

b) Destruição: um processo pode destruir outro ou a si mesmo, retirando-o do núcleo e levando consigo (ou não) todos os outros processos diretamente dependentes dele, ou seja, seus filhos e as gerações destes. A destruição

de um processo envolve a desalocação de sua pilha e de seu PCB, além do cancelamento de seu PID.

c) Suspensão e Continuação: um processo pode suspender outro e depois fazê-lo continuar a rodar. Estas operações envolvem a alteração de seu estado de Ready para Suspended e vice-versa, retirando-o e recolocando-o na fila de prontos para execução.

d) Espera: um processo pode esperar por uma condição qualquer, saindo da fila de prontos e entrando em uma fila de espera específica de alguma estrutura.

e) Temporização: um processo pode pedir uma temporização, isto é, esperar um certo período de tempo para voltar a rodar ou uma determinada hora ser alcançada. Esta é uma operação típica de sistemas de Tempo Real (ver seção II.14).

II.6 Escalonamento de Processos

O escalonamento de um processo é justamente a operação de se transferir o controle do processador de um processo para outro. Esta tarefa envolve o salvamento de todo o contexto do processo interrompido, como registradores da CPU e do co-processador aritmético e o apontador de pilha, e a restauração do contexto do próximo processo a executar. O procedimento que realiza esta operação se chama escalonador ou despachante. O escalonamento em si, a chamada *troca de contexto*, é feito pelos núcleos sem muitas variações, mas a política de escalonamento pode ser definida de diversas formas. Alguns núcleos podem implementar políticas de prioridade e vários esquemas para a fila de prontos, tais como o chamado Round-Robin, filas de múltiplos níveis, FCFS, etc (PETERSON, 1985). Algumas linguagens, como Modula-2, permitem até que o próprio usuário altere essa política.

A política de prioridades envolve atribuir um valor a cada processo, permitindo ao núcleo diferenciá-los por importância e determinar que os mais prioritários rodam por mais tempo ou até acabarem, por exemplo.

Quanto ao esquema da fila de prontos, uma classificação básica reside no fato dele ser preemptivo ou não-preemptivo. O esquema preemptivo faz com que a troca de contexto seja acionada automática e periodicamente pelo núcleo, baseada em um relógio de hardware. Cada período de tempo deste é chamado de *quantum* ou *time-slice*. O esquema não-preemptivo deixa a responsabilidade do escalonamento para o programador, que deve explicitamente pedir a troca de contexto. Enquanto o primeiro é mais cômodo e permite uma maior concorrência, o segundo é de implementação mais fácil e gera programas mais simples de serem depurados. A próxima seção trata especificamente de matéria sobre um esquema não-preemptivo.

II.7 Co-rotinas

Co-rotinas são procedimentos comuns mas que podem ter sua execução suspensa e continuada mais tarde, a partir de onde foram interrompidas. A diferença básica entre uma co-rotina e um procedimento usual é que a chamada deste último sempre faz com que sua primeira instrução seja executada, ao passo que uma co-rotina ao ser chamada (na verdade, "continuada") executa a próxima instrução a partir do ponto onde foi interrompida. A primeira chamada de uma co-rotina, no entanto, é similar a de um procedimento usual.

As operações básicas sobre uma co-rotina são Transfer e Detach (outros termos com o mesmo significado podem ser encontrados na literatura). O Transfer passa a execução de um programa de uma co-rotina para outra, em geral tendo como parâmetro o endereço de entrada da co-rotina destino ou algo similiar a um PID, de forma que ela

possa ser identificada. O Detach passa o controle de execução de volta para o processo que a criou, ou seja, é como se fosse um Transfer onde o destino é implicitamente o pai da co-rotina.

Apenas a introdução do conceito de co-rotina em uma linguagem de alto nível já aumenta a capacidade desta linguagem. Elas podem ser tratadas como um tópico isolado da Programação Concorrente, por não demandarem muitos esforços de implementação e serem bastante úteis na confecção de alguns programas. Vários autores, como HORTON (1986), KRIZ (1980), SEWRY (1988), PAULI (1980), DE RIDDER (1986) e MOODY (1980) possuem trabalhos com co-rotinas, onde foi feita uma extensão de alguma linguagem de alto nível para o seu suporte.

Em HORTON (1986), a melhor referência, pode ser encontrada uma classificação para co-rotinas, que depende das operações suportadas. Um modelo de co-rotinas é dito Semi-simétrico se co-rotinas irmãs (filhas do mesmo pai) não podem utilizar o Transfer para transferir o controle de uma para outra, tendo que necessariamente fazer um Detach para voltar ao pai. O funcionamento deste modelo é semelhante a um esquema mestre-escravo, onde o pai (mestre) pode chamar qualquer uma das co-rotinas filhas (escravas), mas estas devem retornar ao pai para transferir o fluxo do programa. Já no modelo Simétrico a liberdade de transferência de controle é total, estando disponíveis tanto o Transfer como o Detach.

II.8 O Processo Idle

O Idle (inativo) é um processo interno de um núcleo que só roda quando não há mais nenhum outro processo pronto para executar. Sua função é facilitar o gerenciamento da fila de prontos e rodar procedimentos periódicos, como por exemplo testes de "deadlock" (ver seção II.13). Em núcleos com política de prioridade ele tem

a menor delas, geralmente utilizada apenas para o Idle. Autores como SEARS (1985), LEBLANC (1984), BEMMERL (1987), NIEUWENHUIS (1983), HOLT (1987) e COMER (1984) adotam a inclusão do processo Idle em seus núcleos.

II.9 Comandos Alternativos

Esta é uma construção comum em programas concorrentes e é baseada nos comandos guardados de DIJKSTRA (1975). Trata-se de um estrutura de controle não-determinística, isto é, na qual não se pode determinar a priori que fluxo o programa vai seguir. Funciona analogamente a uma estrutura CASE do Pascal, mas não tendo valores associados às opções, sendo estas selecionadas aleatoriamente. Podemos exemplificá-la tomando como molde a construção SELECT da linguagem ADA, onde os colchetes significam itens opcionais e as chaves uma repetição de zero ou mais vezes:

```
SELECT
   [WHEN <expressão booleana>]
      [<comandos>]
   {OR [WHEN <expressão booleana>]
      [<comandos>}
   [ELSE <comandos>]
END SELECT;
```

Os vários comandos precedidos ou não de WHEN são selecionados de forma randômica, até que um deles possa ser executado. Um comando que não esteja precedido de WHEN pode sempre ser executado, uma vez que nenhuma condição lhe é imposta. Já os precedidos de WHEN só podem executar se a expressão booleana for verdadeira, senão um outro comando será escolhido. No caso de nenhum comando poder ser executado, a cláusula ELSE, se existente, será executada. Se não houver ELSE, a semântica pode ser a emissão de um erro, como em ADA, uma tentativa contínua até que um dos comandos seja executado ou simplesmente ignorar o comando SELECT.

II.10 Problemas de Exclusão Mútua

Um dos problemas mais comuns em Programação Concorrente é o da exclusão mútua no acesso a regiões críticas. Uma região crítica é um segmento do código em que um processo pode estar lendo ou atualizando variáveis comuns, escrevendo um arquivo, etc, e somente um pode estar fazendo isso a cada momento. Por isso se diz que os processos devem executar as regiões críticas mutuamente exclusivos no tempo.

Para acessar uma região crítica, portanto, todos os processos devem seguir um determinado protocolo. Primeiro deve-se pedir permissão para entrar na região; quando autorizado, executar a região; por fim, informar que está deixando a região. Para que qualquer protocolo implementado funcione, três requisitos devem ser satisfeitos:

a) Exclusão mútua: somente um processo pode estar executando sua região crítica em dado instante.

b) Progresso: se não há nenhum processo em uma região crítica e existem processos querendo executá-la, somente aqueles que não estão deixando suas regiões (fase final do protocolo) podem participar da decisão de quem será o próximo a entrar em sua região, e esta seleção não pode ser adiada indefinidamente. Mais ainda, um processo parado fora de sua região crítica não pode impedir a entrada de um outro processo em sua região crítica.

c) Espera Finita: deve existir um limite no número de vezes que outros processos são permitidos entrar em suas regiões críticas, após um processo ter solicitado sua entrada e antes dela ser atendida.

É admitido que todos os processos executam numa velocidade maior que zero, mas nada se pode afirmar sobre

suas velocidades relativas.

Um exemplo simples de problema de exclusão mútua é a atualização de uma variável compartilhada por vários processos, como mostra o trecho de programa a seguir.

```
Cobegin
   x:=x+2;  Also
   x:=x+3;
Coend;
```

Temos neste exemplo dois processos, cada um executando uma atualização da variável "x", e duas regiões críticas, justamente as atribuições a esta variável. O problema da exclusão mútua surge devido a essas atribuições não serem executadas de forma indivisível. Se as duas expressões, por exemplo, pegarem o mesmo valor de "x" no lado direito, ao atribuir o resultado a esta variável uma das somas será perdida. O correto seria uma delas executar integralmente primeiro; aí, então, a segunda obteria um valor de "x" já atualizado.

Existem diversos algoritmos de software e algumas instruções de hardware em certas máquinas para a garantia de exclusão mútua. Porém, tanto utilizar as instruções de máquina como os algoritmos diretamente nos programas tornam estes bastante complexos. A melhor solução é utilizar mecanismos próprios que encubram sua implementação. O Capítulo IV discute melhor esses mecanismos.

II.11 Problemas de Sincronização

Um problema de sincronização entre processos é quando se tem mais de um processo cooperando na mesma tarefa, e para realizá-la cada um deles deve respeitar certos tempos e condições impostas pelos outros. Como exemplo, podemos citar o clássico problema do *Produtor-*

Consumidor com número de buffers limitado. Quando o produtor produz um item, ele deposita num buffer vazio do consumidor, que posteriormente o retira e o processa. O problema da sincronização acontece quando o produtor produz mais rápido que o consumidor consome, lotando os buffers deste. Desta forma, o produtor deve esperar para sincronizar com o consumidor. Por outro lado, se o consumidor for mais rápido, este deverá esperar pelo produtor, pois seus buffers estarão vazios e não haverá nada para processar.

Este exemplo se encontra no Anexo IV, implementado utilizando um mecanismo discutido no Capítulo IV.

Outros problemas clássicos de sincronização, com os quais novos algoritmos são sempre testados, são o dos *Leitores e Escritores* (COURTOIS, 1971) e os *Dining Philosophers* (DIJKSTRA, 1965).

II.12 Comunicação entre Processos

Os problemas de sincronização, discutidos na seção anterior, são na verdade exemplos simples de um problema mais amplo, que é o da comunicação entre processos que desejam cooperar. Para resolver tal problema, existem dois esquemas básicos e complementares: memória compartilhada e trocas de mensagens. Os dois esquemas já tiveram uma breve discussão na Introdução desta Tese, e uma discussão mais ampla de ambos será deixada para seus respectivos capítulos (IV e V).

II.13 Deadlocks

Um deadlock é uma situação de impasse onde um ou mais processos, que detém algum recurso, necessitam de um outro que está em poder destes mesmos processos, que também esperam por uma liberação de recursos. Mais precisamente,

um conjunto de processos está em estado de deadlock quando todos os processos deste conjunto estão esperando por um evento que só pode ser causado por um outro processo deste mesmo conjunto. Um deadlock, obviamente, é uma situação indesejável, uma vez que os processos neste estado não progridem.

Um deadlock pode ser caracterizado por quatro condições necessárias:

1) Exclusão mútua: pelo menos um dos recursos disputados deve ser conseguido com exclusão mútua, ou seja, apenas um processo pode estar utilizando-o em um dado momento.

2) "Hold and Wait": deve existir ao menos um processo que detenha algum recurso e esteja esperando por recursos adicionais, que estão em poder de outros processos.

3) Entrega voluntária: os recursos não podem ser tomados por outros processos. O processo que detém um recurso deve entregá-lo voluntariamente.

4) Espera circular: deve existir um conjunto de processos bloqueados tal que o primeiro espera por um recurso em poder do segundo, o segundo espera por um recurso em poder do terceiro e assim por diante, fechando um círculo de espera.

Uma boa discussão sobre deadlocks pode ser encontrada em PETERSON (1985).

II.14 Sistemas de Tempo Real

Sistemas de Tempo Real são aqueles em que o processamento deve ser realizado dentro de certos limites. Estão, em geral, relacionados com dispositivos externos de aplicação dedicada, como equipamentos médicos, industriais e científicos. A Programação Concorrente está intuitiva-

de processo se encaixa perfeitamente bem para modelar um sistema de Tempo Real.

Núcleos de suporte para Tempo Real podem ser encontrados em DOWNES (1982), MAGALHÃES (1986), BEMMERL (1987), BOWLING (1989), CHERITON (1979), CRAMER (1988), GARETTI (1982), GEHANI (1989), LABRECHE (1990), MANIECKI (1984), NIEUWENHUIS (1983), RAJA (1990) e SEARS (1985).

CAPITULO III: O MODULO KERNEL

Neste capítulo apresentaremos o principal componente do ambiente *Kernel*, que é o módulo de mesmo nome. Ele é o responsável pelo gerencimento de processos e implementa primitivas e estruturas próprias da Programação Concorrente.

III.1 Introdução ao Kernel

Este módulo, como os outros três a serem apresentados nos próximos capítulos, é uma "unit" do Turbo Pascal, ou seja, um módulo pré-compilado onde constantes, variáveis, tipos, procedimentos e funções são exportados para uso em algum programa. Para programar com o Kernel são necessários apenas um micro IBM-PC compatível (XT, AT, 386, 486...) com 640k de memória e o software Turbo Pascal (a partir da versão 5.0). Outros detalhes podem ser encontrados na Ficha Técnica, no Anexo I.

A construção do Kernel foi baseada nos melhores mecanismos de diversos trabalhos sobre núcleos para suporte de Programação Concorrente. Nenhum dos trabalhos pesquisados reunia tantas características como as implementadas aqui, o que o torna uma ferramenta bastante versátil e diferente. Mesmo as linguagens com suporte próprio de concorrência são muito limitadas, como, por exemplo, Modula-2. Esta linguagem, na versão de WIRTH (1977), só implementa processos através de co-rotinas, sem nenhuma forma de preempção. Ainda mais, as co-rotinas criadas não podem ter parâmetros. Uma versão de Modula-2 para micros IBM-PC denominada TopSpeed Modula-2 introduziu o escalonamento preemptivo, mas ainda sofre do problema dos parâmetros nas co-rotinas.

Uma das aplicações mais imediatas do Kernel é justamente em cursos sobre Programação Concorrente e Sistemas Operacionais. A parte prática nestes cursos, ge-

ralmente relegados apenas a conceitos teóricos, é defendida e tem sido aplicada por vários autores. Por exemplo, BRINCH HANSEN (1983) utilizou sua linguagem Edison como ferramenta de programação; KAUBISCH (1976) utilizou SIMONE; BEN-ARI (1982) utilizou um kit adaptado de Wirth denominado Concurrent Pascal-S e na COPPE/UFRJ VASCONCELOS (1989) utilizou um núcleo denominado NPCP.

A interface do Kernel, ou seja, nomes e parâmetros das primitivas, se encontra no Anexo II. Um exemplo de utilização pode ser achado no Anexo VIII.

III.2 O Modelo das CPUs Virtuais

Uma das diferenças mais marcantes do Kernel em relação aos núcleos de concorrência tradicionais é o modelo das CPUs virtuais. Concebido de forma independente e original, tem similar com o trabalho de BUHR (1990). A idéia é que o Kernel, além de implementar o conceito de processo, introduzindo concorrência em uma linguagem seqüencial, também simula a existência de mais de um processador. Com isso, em uma máquina uniprocessada que é um micro IBM-PC comum, podemos executar programas como se fossem rodar em máquinas paralelas ou sistemas distribuídos.

A idéia é bem simples: o programa principal, ao iniciar, deve chamar o procedimento SetProcessors (ver seção III.15.6). Este faz com que "outros" processadores sejam alocados, que nada mais é do que a criação de outros processos semelhantes ao programa principal. Estes processos, chamados MAIN-X, onde X é o número de sua CPU (processador) virtual, começam a executar exatamente a partir do primeiro "begin" do programa. Sendo processos, têm contextos próprios, e o tamanho de suas pilhas é determinado pelo Kernel. O programa principal original (MAIN-0) roda na CPU virtual default, que é a zero. Sua pilha é determinada pela diretiva de compilação M+ do Turbo Pascal (BORLAND, 1989b). A única diferença entre o MAIN-0 e

os outros "MAIN", para efeito de execução, é a semântica da primitiva SetProcessors: esta só tem efeito quando é chamada pela CPU virtual 0.

Para simular a idéia de vários processadores se usa a mesma filosofia básica de programas concorrentes: alterna-se, a cada período de tempo determinado por um relógio de hardware, que CPU virtual está realmente executando. Esta é uma extensão de um escalonador de processos baseado em um esquema Round-Robin. A cada time-slice se troca não só o processo, mas também o processador. Ilustrando, vamos imaginar três CPUs virtuais, 0, 1 e 2. A CPU 0 possui os processos A e B prontos para rodar; a CPU 1 só possui o processo C; a CPU 2 possui os processos D, E e F. No primeiro time-slice A rodará, no segundo rodará C, no terceiro D, no quarto B, no quinto C, no sexto E, no sétimo A e assim por diante. Desta forma, há o Round-Robin dos processos, feito em cada CPU, e o Round-Robin das próprias CPUs.

Além de todo este esquema, o Kernel implementa política de prioridades para os processos. Na versão atual existem três tipos de prioridades, 0, 1 e 2, onde a prioridade 0 é a mais baixa e a prioridade 2 a mais alta. A prioridade default é 1. Em razão disto, existem também três filas de processos prontos em cada CPU virtual, uma para cada prioridade. Esta separação por filas torna a implementação mais simples e eficiente. Em cada uma destas filas, há um processo denominado *Head* (líder), que tem por finalidade facilitar as operações de inclusão e exclusão nas filas. E, para permitir o chaveamento entre as CPUs virtuais, há um processo denominado *Current* (corrente), que é o próximo processo a rodar em cada CPU.

A apresentação das CPUs virtuais e o funcionamento do escalonador serão retomados na seção III.10.

III.3 O Modelo das Famílias de Processos

Os processos no Kernel são criados primeiramente a partir do programa principal. Estes podem criar outros e assim por diante, formando processos hierárquicos como discutido na seção II.3. Ao contrário de algumas implementações, como o famoso XINU de COMER (1984), o Kernel não limita o número máximo de processos em razão de uma alocação prévia de um tabela de PIDs. Ao invés disso, os PIDs vão sendo gerados seqüencialmente, até o limite do tipo de dados com o qual foi definido. Nesta versão, foi usado o tipo Word, permitindo até 65535 processos. Se este valor não for considerado razoável, o Kernel pode ter seu programa fonte facilmente alterado para utilizar um tipo Longint como PID, permitindo até 4.294.967.295 processos (!).

A ausência de uma tabela de PIDs, utilizada para localizar o PCB de um determinado processo, pode ser explicada pelo conceito de família adotado no Kernel. Uma Família de Processos compreende apenas os descendentes diretos, ou seja, os filhos de um processo. No PCB de cada processo temos apontadores para seu pai, seu filho mais novo, seu irmão mais velho e seu irmão mais novo. Com essas relações, podemos percorrer seqüencialmente uma família em busca de um determinado PID. E é justamente essa a implementação do Kernel: para localizar o PCB de um processo a partir de seu PID, começa-se a procurá-lo seqüencialmente entre os filhos de seu pai. Isto é válido porque todas as primitivas que envolvem essa operação são feitas somente pelos pais em relação a seus filhos. Não é permitido, portanto, executar uma primitiva sobre um processo fora de uma família. Tomando como usual famílias não muito grandes, esta busca seqüencial pode ser considerada com melhor custo/benefício do que o uso da tabela de PIDs.

Com este modelo de Família de Processos, poder-se-ia até mesmo fazer a geração de PIDs local à cada família. Isto permitiria um número muito maior de processos gerados com o mesmo tipo de dados, uma vez que cada processo poderia ter até 65535 filhos. Esta idéia não foi adotada visando evitar complicações futuras com alguma primitiva que mudasse um processo de família, mas é potencialmente viável.

III.4 O PCB (Process Control Block)

O PCB utilizado no Kernel é um record bastante extenso. A seguir descreveremos cada campo e sua função.

* *PID*. A identificação do processo.
* *Name*. O nome do processo, para efeitos de depuração.
* *State*. O estado em que o processo se encontra.
* *Prio*. A prioridade do processo.
* *CPUid*. A CPU virtual onde o processo está alocado.
* *Stack*. O apontador de pilha do processo.
* *StackBase*. Um ponteiro para a base da pilha do processo, para quando o processo for destruído e a pilha for desalocada.
* *StkSize*. O tamanho da pilha do processo, para quando a pilha for desalocada.
* *Next*. O próximo processo na fila onde ele se encontra.
* *Last*. O processo anterior na fila onde ele se encontra.
* *Parent*. O pai do processo (quem o criou).
* *OlderBro*. O irmão mais velho do processo.
* *YoungerBro*. O irmão mais novo do processo.
* *Youngest*. O filho mais novo do processo (caçula). É ele que encadeia os outros filhos do processo.
* *Children*. O número de filhos do processo (meramente informativo).
* *Dependents*. O número de filhos dependentes, isto é, criados por um Cobegin (ver seção III.15.2). São contados também no campo Children.

* *JustBorn*. Uma "flag" indicando um processo recém-criado por um Fork ou LongFork (ver seção III.15.2).
* *Cobeginner*. Uma "flag" indicando que o processo foi criado por um Cobegin (ver seção III.15.2).
* *Use80x87*. Uma "flag" indicando se o processo quer salvar, em algum momento, o contexto do co-processador aritmético.
* *Save80x87*. Uma "flag" indicando se o processo precisa salvar, quando interrompido, o contexto do co-processador aritmético.
* *_80x87State*. Um ponteiro para uma área de 94 bytes, quando alocada, que contém o contexto do co-processador aritmético salvo.
* *WaitingPrio*. A prioridade do processo em alguma fila de espera. É um valor inteiro, entre -32768 e 32767. Quanto maior o valor, maior a prioridade.
* *TimeoutFlag*. Uma "flag" indicando se o processo teve um timeout, ou seja, saiu de uma fila de espera devido à expiração de um tempo máximo contido no campo Timeout.
* *Timeout*. Tempo máximo que um processo pode permanecer em uma fila de espera. É contabilizado em "ticks" (aproximadamente 55ms), e tem como teto aproximadamente 32776 horas (mais de três anos!).
* *Rest*. Tempo que falta para o timeout do processo. É contabilizado em "ticks".
* *NextDelayed*. O próximo na fila de temporização, se o processo se encontrar nela.
* *WaitList*. Endereço da fila de espera onde o processo se encontra, se for o caso. É utilizada no caso de timeout, para retirada do processo desta fila.
* *MsgAddr*. É um ponteiro para um endereço qualquer, que na implementação atual é utilizado para auxiliar um rendezvous em uma troca de mensagens (ver Capítulo V).
* *SelectIP*. Offset do endereço onde o procedimento Select (ver seção III.15.10) está sendo chamado.
* *NumAlt*. Número total de opções em um procedimento Select.

* *NextAlt*. Próxima opção selecionada em um procedimento Select.
* *AltTests*. Número de opções já testadas e rejeitadas em um procedimento Select.

III.5 Estados dos Processos

São os seguintes os possíveis estados de processos no Kernel:

- *Running*: estado do processo que está executando. Este estado é implícito no Kernel, pois na verdade os processos quando rodam não saem da filam de prontos.
- *Ready*: estado dos processos que se encontram na fila de prontos para executar.
- *Waiting*: estado dos processos que se encontram em alguma fila de espera.
- *Suspended*: estado dos processos que foram explicitamente suspensos.
- *Timing*: estado dos processos que se encontram exclusivamente em temporização.
- *Halted*: estado exclusivo dos processos MAIN-X, quando estes já acabaram.
- *Unknown*: estado desconhecido. Na verdade, só é utilizado como retorno na primitiva GetState (ver seção III.15.7), quando o processo referenciado não foi encontrado ou não existe mais.

Um processo no estado Waiting, que possua no campo Timeout do PCB um valor maior que zero, é o único caso no Kernel onde um processo pode se encontrar em mais de uma fila simultaneamente. Isto porque sua saída da fila de espera pode ser feita pelas vias normais ou em vista de um fim de temporização. A condição que for atendida primeiro faz o processo voltar à fila de prontos e cancelar sua presença nas duas filas.

III.6 A Criação de Processos

A criação de um processo no Kernel envolve a alocação de um novo PCB, a geração de um novo PID e a iniciação de alguns campos do PCB. O PCB é alocado dinamicamente da heap e o PID, como dito anteriormente, é o próximo valor da seqüência no Kernel. A iniciação de alguns campos do PCB consiste no encadeamento do processo na família de seu pai e a criação de sua própria família, sua imediata inclusão na lista de prontos, a atribuição de valores iniciais aos vários campos existentes, a alocação de uma área para salvamento do contexto do co-processador aritmético (se for o caso) e por fim a alocação de uma nova pilha. Esta pilha também é uma variável dinâmica, e contém inicialmente valores que serão atribuídos aos registradores quando o processo for escalonado pela primeira vez. O endereço alocado como base da pilha é sempre ajustado de forma a coincidir com uma posição par, devido a esse fato implicar em um melhor desempenho nos microprocessadores utilizados na linha IBM-PC (Intel 80x86).

Deve-se atentar principalmente para o tamanho da pilha alocada ao novo processo, de forma que este não seja subdimensionado. Os problemas oriundos deste tipo de erro são os mesmos que em um programa seqüencial, ou seja, funcionamento incorreto ou pane total. É aconselhável em fases de teste o uso da diretiva S+, que realiza checagem de tamanho de pilha suficiente quando da entrada em procedimentos. A utilização de recursividade em processos pode ser feita normalmente, bem como na criação destes.

A criação de um processo pode ser feita através de mais de uma forma, utilizando as primitivas descritas na seção III.15.2. Cada uma delas se adequa melhor em determinados casos, e fica a gosto do usuário a escolha entre elas. Inicialmente, o Kernel cria dois processos antes mesmo de um programa começar a rodar, que serão discutidos individualmente a seguir. São eles o processo Idle e o MAIN-0.

III.6.1 O Processo Idle

O conceito de processo Idle foi introduzido na seção II.8. Ele é o primeiro processo criado pelo Kernel, tendo por isso o PID igual a zero e seu nome para efeito de depuração é "IDLE". Não é contado como processo ativo, e só executa quando não há mais nenhum outro processo pronto para executar. Sua função, além de facilitar a gerência da lista de processos prontos, é fazer um teste trivial de deadlock e executar uma rotina de idle do usuário.

O teste de deadlock consiste simplesmente em verificar se não há mais nenhum processo ativo e se existem processos em estado de espera sem que nenhum deles esteja em temporização. Se algum processo está esperando e nenhum outro pode tirá-lo desse estado, o Kernel acusa um erro de deadlock. A necessidade de testar que o número de processos ativo é zero, aparentemente sem sentido, advém do fato de algum processo que estava temporizando voltar à lista de prontos antes do referido teste ser feito (isto é possível pois a temporização é uma operação assíncrona). Além disso, como o processo Idle é um loop que só termina quando o número de processos vivos é igual a zero (terminação normal), a cada iteração ele sempre libera a exclusão mútua de uso interno do Kernel e espera 55ms. Este é o tempo necessário para um processo que acabou uma temporização conseguir acusar o evento e entrar na lista de prontos.

A rotina de idle do usuário é chamada a cada iteração do loop do processo Idle, *antes* do teste de deadlock. Isto permite ao usuário inserir um tratamento mais refinado a um possível deadlock ou simplesmente executar algum procedimento de espera, como exibir algo no vídeo, imprimir um texto, etc. Para incluir uma rotina própria, o usuário deve atribuir um procedimento à variável de tipo procedural *UsrIdleProc*. O valor default desta variável é um procedimento nulo do Kernel chamado *DoNothing*, que também pode ser utilizado pelo usuário.

III.6.2 O Processo MAIN-0

O processo MAIN-0 é, na verdade, o programa seqüencial do usuário que inicia no primeiro "Begin", ou seja, o bloco principal. Sua transformação de programa em processo é a base da construção da família de processos que será gerada a partir dele. É ele o pai do primeiro processo criado pelo usuário. Seu PID é igual a 1 e seu nome para depuração é MAIN-0. Sua pilha é a já existente para o programa seqüencial, alocada pelo compilador. Ele é criado permitindo o salvamento de contexto do co-processador aritmético, mas desabilita-o antes mesmo de rodar. Se o usuário desejar esta característica, deve fazê-lo explicitamente com a primitiva Enable80x87Save (ver seção III.15.7). O processo MAIN-0 termina no próprio fim do programa seqüencial usual.

III.7 A Destruição de Processos

A destruição de um processo envolve a sua exclusão da família de seu pai, a destruição de sua própria família, sua retirada da lista de prontos, a desalocação de sua pilha e a da área de salvamento de contexto do co-processador aritmético (se for o caso) e a desalocação de seu PCB, se não for um processo MAIN-X. Um processo MAIN-X não pode ter seu PCB desalocado, pois ele é o processo responsável pela sua CPU virtual.

A exclusão da família de seu pai significa reencadear seus irmãos e deixar para o pai um novo caçula, se ele o era. A destruição de sua família significa destruir, em cadeia, seus filhos, os filhos destes e assim por diante. Os PIDs nunca são reutilizados, devido ao método de alocação seqüencial utilizado na criação. Sua retirada da lista de prontos pode deixar a sua fila de prioridades vazia, fazendo com que o novo processo corrente seja um de prioridade mais baixa. Se nenhum existir, a CPU virtual fica inativa, podendo alterar a execução do

despachante ou fazer o programa terminar (ver seções III.10 e III.14).

Um processo só pode ser efetivamente destruído quando ele estiver na lista de prontos. Se ele mesmo solicita sua destruição, o efeito é imediato. Se seu pai é quem solicita, o processo deve primeiro voltar à lista de prontos, se não estiver lá ainda. Caso não esteja, a sua pilha é alterada de forma que ao ser escalonado da próxima vez ele caia diretamente no procedimento que o destrói. Este esquema evita possíveis deadlocks e também foi utilizado nos núcleos de GARETTI (1982) e DUGLOSZ (1988).

A destruição de um processo pode ser implícita ou explícita. Processos criados pelas primitivas Cobegin/Also/Coend terminam automaticamente. Processos criados por um Fork ou LongFork só terminam se alcançarem o "End" final de um programa ou se forem explicitamente destruídos. Processos criados por Spawn ou LongSpawn terminam automaticamente no fim do procedimento onde começaram a executar.

III.8 Exclusão Mútua no Kernel

Todas as primitivas do Kernel são executadas com exclusão mútua, ou seja, há a garantia de não-preempção enquanto manipulando estruturas de dados globais. No interior do Kernel são utilizados os procedimentos Lock e Unlock, que desabilitam e habilitam, respectivamente, o escalonador de processos. Seu uso foi estendido também ao usuário, por serem primitivas simples e em muitos casos dispensarem o uso de mecanismos mais sofisticados. Contudo, há exceções de uso que devem ser respeitadas, para não violar a exclusão mútua no Kernel. Ao usar um Lock no seu programa, o usuário não deve chamar nenhuma primitiva do Kernel ou dos outros módulos deste ambiente, que podem também utilizar este mesmo mecanismo. O problema surge porque ao acabar uma destas primitivas é chamada o Unlock,

e ao voltar ao programa do usuário não há mais exclusão mútua.

Comentários e maiores detalhes sobre Lock e Unlock podem ser encontrados na próxima seção, na seção III.15.4 e no Capítulo IV.

III.9 A Interrupção do Relógio de Hardware

Esta seção é um pouco mais técnica, pois se refere à manipulação do relógio de hardware existente no micro IBM-PC. Todos estes micros possuem um "chip" com funções de temporização, comumente chamado de 8253. No caso específico de IBM-PCs, ele é conectado à interrupção de hardware de número 8, programado pela BIOS para gerar um pulso ("tick") a cada 55ms, aproximadamente. Na verdade, são precisamente 18,20648193 ticks por segundo. Existe também uma outra interrupção, a 28 ($1C, em hexa), que é chamada a partir da de número 8 e pode ser utilizada para aplicações do usuário. Para se implementar um escalonador preemptivo, basta revetorizar uma das duas interrupções para uma rotina própria, que além de cuidar da troca de contexto deve manter o relógio da BIOS atualizado.

A princípio, utilizar a interrupção $1C seria preferível, pois é destinada realmente ao usuário e dispensaria uma atualização deste relógio da BIOS, já realizada pela própria interrupção 8. Contudo, não é possível implementar um escalonador com a $1C, pois quando ela é invocada o registrador de segmento de pilha (SS) é alterado pela BIOS. O problema conseqüente disto é que a troca de contexto salvaria um apontador de pilha falso. Por esta razão, o despachante do Kernel foi implementado revetorizando-se diretamente a interrupção de hardware 8.

Outro motivo para manipular diretamente a interrupção de hardware deve-se ao fato do Kernel permitir a alteração do time-slice original. O procedimento

SetTimeSlice, a ser discutido na seção III.15.6, pode modificar o time-slice para alguns valores padrões. Como dito antes, o chip 8253 é programado pela BIOS para interromper a CPU a cada 55ms, mas este tempo é o valor máximo permitido por ele. Se diminuirmos essè tempo, teremos interrupções mais freqüentes e mais overhead de atendimento, mas a ilusão de concorrência é bastante melhorada. A alteração deste tempo pode ser feita de modo simples e rápido reprogramando-se o chip 8253. O problema mais imediato desta alteração é a atualização do relógio da BIOS, que deve continuar sendo feito com a antiga precisão de 55ms. Para resolver este novo problema, basta que só sejam permitidos novos períodos divisores do original. Com isto, a rotina que trata da interrupção 8 pode ter um contador que só chame a rotina de atualização do relógio da BIOS cada vez que o contador atinja o valor do divisor. Mais ainda, como o valor original é uma potência de 2, os divisores também devem sê-lo, de forma a não gerar restos e com isso imprecisão. Uma boa referência sobre este tópico é BOWLING (1989).

A rotina de atendimento da interrupção do relógio fica presente permanentemente, a menos que haja apenas um processo ativo. Esta medida visa evitar perda de tempo, pois neste caso não há necessidade de preempção e a presença da rotina traria um overhead sem nenhum ganho. Como em seu interior e também através do uso das primitivas Lock e Unlock pelo usuário o Kernel inibe a ação do despachante, a rotina de atendimento foi implementada eficientemente e levando esse aspecto em consideração. Através de uma única "word" (2 bytes) utilizada em um "jump" indireto, a rotina decide se vai executar o despachante (trocar de contexto) ou não. O efeito do Lock e Unlock é simplesmente alterar essa "word" convenientemente.

III.10 O Funcionamento do Despachante e a Ready List

O despachante do Kernel é a rotina responsável pela troca de contexto de um processo para outro. Seu funcionamento, numa visão mais geral, é bastante simples. Declarado como um procedimento "Interrupt" do Turbo Pascal, logo ao iniciar ele empilha automaticamente todos os registradores existentes no microprocessador 8086. Em seguida, se o processo corrente, denominado *Running* (um ponteiro para um PCB), precisa salvar o contexto do co-processador aritmético, esta operação é realizada. O apontador de pilha do Running é salvo no campo Stack do seu PCB e um novo processo, denominado *RunNext*, passa ao seu lugar.

Antes da troca de contexto propriamente dita, uma rotina chamada *ScheduleNext* faz rodar a próxima CPU virtual, como elaborado na seção III.2. Se somente uma única CPU virtual existir, esta rotina, na verdade uma chamada indireta para um procedimento adequado, aponta para uma instrução "RET" (retorno de chamada de procedimento). Esta forma de implementação conserva todas as vantagens do modelo das CPUs virtuais sem perda de performance nos casos em que essa característica não é utilizada. O escalonamento de uma CPU virtual significa salvar o processo Running recém-escalonado (pois este pertence à CPU virtual correntemente executando) no seu campo Current e determinar a próxima, em seqüência numérica circular, dentre as que estiverem ativas. O processo Current da próxima CPU assume o papel do Running.

O passo seguinte do despachante é determinar um novo RunNext, ou seja, qual o processo que rodará no próximo escalonamento. Isto é feito atribuíndo-se a ele o campo Next do processo Running, que é o seu próximo na *Ready List*. A Ready List, na nomenclatura do Kernel, é a lista de processos prontos para execução, independente de sua prioridade ou CPU virtual. Apesar dessa abrangência, os

processos só são encadeados numa mesma fila de prioridades e em cada CPU virtual, de forma circular.

Continuando a seqüência, o despachante faz a troca da pilha do antigo Running (a que está em uso) para a pilha do processo atual. O contexto do co-processador aritmético é restabelecido, se for o caso, e a exclusão mútua do Kernel é liberada. Esta operação é necessária porque muitas primitivas do Kernel, executadas com exclusão mútua, terminam com uma chamada explícita do despachante, sem antes tê-la liberado. Por fim, o próprio código de saída de um procedimento Interrupt desempilha todos os registradores salvos "anteriormente", mas que na verdade agora pertencem a outro processo.

III.11 O Controle de Temporizações e a Time List

Processos que requisitam temporizações, explícita ou implicitamente, são inseridos numa lista especial denominada *Time List*. A Time List é uma lista encadeada simples, ordenada por tempos relativos. Este método de implementação pode ser encontrado em MILENKOVIĆ (1987), e funciona da seguinte maneira: o primeiro processo da lista contém o tempo que resta para acabar sua temporização; o segundo processo, apenas o tempo que resta após acabar a temporização do primeiro; o terceiro, o tempo que resta após a temporização do segundo e assim por diante. Desta forma, os tempos inseridos na lista são todos relativos.

Para ilustrar melhor o funcionamento da Time List, vamos tomar como exemplo a inclusão de quatro processos com temporizações de 5, 8, 8 e 12 segundos. O tempo relativo do primeiro processo, o menor de todos, ficará mesmo em 5 segundos. O segundo processo tem 3 segundos a mais que o primeiro, e portanto é este o seu tempo relativo. O terceiro tem o mesmo tempo absoluto que o segundo, e portanto seu tempo relativo é zero. Por fim, o tempo relativo do quarto processo é de 4 segundos, que é a

diferença entre o seu tempo absoluto e o do terceiro processo.

Algoritmicamente falando, a inclusão de um processo na Time List é realizada comparando-se o seu tempo relativo, inicialmente igual ao absoluto, com o do primeiro processo na lista. Enquanto aquele valor for maior, subtrai-se dele o tempo relativo do processo sendo comparado. Quando for menor, subtrai-se do processo de maior valor o tempo relativo final do processo sendo incluído. Inclusões no início e fim da lista requerem o acréscimo de pequenos testes. Fundamentalmente, o tempo absoluto de um processo na lista é a soma de todos os tempos relativos dos processos que o antecedem, inclusive o próprio.

Por ser essencialmente uma rotina de tempo real e não estar ligada à presença ou não do despachante, o procedimento que controla as temporizações é uma rotina de atendimento de interrupção (ISR - Interrupt Service Routine), conectada à interrupção $1C (vide seção III.9). A cada atualização do relógio da BIOS, ou seja, 55ms, o tempo relativo do *primeiro processo apenas* é decrementado. Se este chegar a zero ou ficar negativo, o processo é retirado da Time List. É possível um tempo ficar negativo quando dois ou mais processos têm uma diferença igual a zero entre si. Na implementação do Kernel, ao contrário da de MILENKOVIĆ (1987), apenas um processo por vez pode ser retirado da Time List. Isto é feito de modo a não causar perda de interrupções do relógio devido a demoras na rotina de atendimento, uma vez que dentro dela as interrupções estão desabilitadas. Com isto, um processo que tinha tempo relativo igual a zero e já podia ser retirado da lista aguarda a próxima interrupção, ficando seu valor negativo.

A entrada de um processo na Time List pode ser explícita ou implícita, como dito anteriormente. As primitivas Delay e Schedule (ver seção III.15.8) inserem o

processo na Time List especialmente para aguardar certo tempo ou uma determinada hora chegar. Primitivas que levem o processo a um estado de espera (Waiting), tendo sido especificado um timeout para a operação, colocam-no na Time List implicitamente. A especificação de um timeout é realizada com a primitiva SetTimeout (ver seção III.15.8).

A saída de um processo da Time List pode ser feita normal ou forçosamente, sempre fazendo-o retornar à Ready List. A saída normal deve-se ao fim da temporização solicitada; a forçosa deve-se ao cancelamento da temporização, nos casos em que a espera do processo por uma condição terminou antes. Na saída normal, se o processo estava temporizando implicitamente, o campo do PCB TimeoutFlag é colocado em True, indicando que houve um timeout da operação solicitada. A função booleana Timedout pode ser utilizada após tal uso da primitiva, para tratar adequadamente o timeout.

III.12 Exclusão Mútua em Operações de Entrada e Saída

As operações de entrada e saída (E/S) do Turbo Pascal, como consta em seu manual, não são reentrantes. Isto acontece porque elas utilizam o DOS, que por sua vez utiliza as rotinas da BIOS. As interrupções da BIOS, consideradas individualmente, não são reentrantes, na sua maioria. Além disso, em relação ao Turbo Pascal, existe o problema da manipulação do buffer de E/S de um determinado arquivo lógico, isto é, da variável que representa um arquivo físico qualquer. Nos casos gerais, tanto em operações de E/S de arquivos, escritas em tela, leituras de teclado, impressões e tudo o mais, a exclusão mútua deve ser garantida pelo próprio usuário, com algum dos mecanismos disponíveis. O próprio Lock/Unlock do Kernel é bastante útil e eficiente nestes casos. Estas observações também são válidas quanto ao uso direto de interrupções da BIOS pelo usuário. Quanto ao uso de funções do DOS, não há problema algum, apesar delas também não serem reentrantes.

É que neste caso o Kernel só escalona um novo processo quando o atual não está utilizando o DOS.

Contudo, existem algumas facilidades providas pelo Kernel que diminuem bastante o trabalho de codificação desta exclusão mútua. No caso de leituras do teclado, por exemplo, ela não é necessária. O Kernel revetoriza a antiga rotina para uma que chama o despachante enquanto o teclado está ocioso, garantindo automaticamente a exclusão mútua e aumentando a concorrência do programa. Em operações de E/S com arquivos do tipo Text, incluídos aqui escritas na tela e impressões, o trabalho é bastante simplificado com o uso da primitiva LockFile (ver seção III.15.4) sobre o arquivo em questão. A primitiva faz com que todas as operações de E/S do arquivo sejam realizadas com exclusão mútua. Ainda assim, cada processo deve ter a sua própria variável de arquivo Text, devido a problema da manipulação do buffer de E/S citado anteriormente. Por exemplo, para vários processos escreverem na tela ao mesmo tempo, de forma segura, cada um deles deve ter um arquivo Text associado ao dispositivo de saída de tela e protegê-lo com LockFile. Esta associação pode ser feita comumente de duas maneiras: se o programa utiliza a unit Crt, chamando o procedimento AssignCrt sobre o arquivo; se utiliza a saída padrão, abrindo o arquivo com o nome "CON".

Uma última observação refere-se à concorrência entre um processo que faz leitura de teclado e outro que escreve na tela. Neste caso, apesar de cada um aparentemente utilizar um recurso diferente, a rotina de leitura de teclado usualmente faz ecoar a tecla pressionada. Com isso, há disputa de uso da tela, o que leva o processo que escreve nela a utilizar algum mecanismo de exclusão mútua, como o LockFile sobre o arquivo Output.

III.13 Exclusão Mútua no Manuseio da Heap

A heap é a área da memória utilizada pelo Turbo Pascal para alocação de variáveis dinâmicas. Seu manuseio, conforme informação constante no manual da linguagem, não é reentrante. Isto é facilmente compreensível devido à manutenção da chamada "Free List", que é uma lista encadeada com os blocos livres da heap. Portanto, processos que utilizam variáveis dinâmicas teriam que garantir a exclusão mútua da heap, mas isto não é necessário. O Kernel estabelece exclusão mútua automaticamente quando são utilizadas as primitivas New, Dispose, GetMem, FreeMem, Mark e Release, inclusive nas chamadas implícitas contidas em "Constructors" e "Destructors" de variáveis dinâmicas do tipo Object (BORLAND, 1989a).

O programador, em casos em que apenas um processo faça uso de variáveis dinâmicas, pode desabilitar e reabilitar esta característica, através das primitivas ResetHeapMutex e SetHeapMutex (ver seção III.15.4). Com a desabilitação o programa pode ganhar mais velocidade, mas ela deve ser utilizada com cuidado. Quanto à questão do próprio Kernel também utilizar variáveis dinâmicas na criação de um processo, não há com o que se preocupar. Mesmo o programador utilizando a primitiva ResetHeapMutex o Kernel ainda protege o manuseio da heap.

Sobre a implementação desta exclusão mútua da heap, consultar a seção III.16.

III.14 A Terminação de um Programa

A terminação de um programa pode acontecer de modo normal ou anormal. Devido ao modelo das CPUs virtuais, o conceito de terminação no Kernel é um pouco mais elaborado. Independentemente do modo, ao terminar o Kernel restaura todos os vetores de interrupções alterados, como o do relógio de hardware, do tick e do teclado. O usuário

pode continuar utilizando procedimentos de saída normalmente, através da modificação da variável ExitProc do Turbo Pascal. Mas, devido ao modelo das CPUs virtuais, o término de uma CPU nem sempre significa o término do programa como um todo. Por isso, se utilizar um procedimento de saída encadeada em um programa com mais de uma CPU virtual, após as devidas modificações o programador deve chamar a primitiva InstallKernelExit (ver seção III.15.9).

III.14.1 Terminação Normal

Um programa para terminar normalmente deve ter todas as suas CPUs virtuais inativas (quando nenhuma possui processos ativos, exceto o que está rodando) e estar com a Time List vazia. O término dos processos depende da forma como foram criados ou do uso de primitivas de destruição no programa. A chamada do procedimento Halt do Turbo Pascal tem efeito local sobre a CPU virtual que o executou, destruindo todos os seus processos, fazendo-a ficar inativa e colocando seu MAIN no estado Halted. A primitiva HaltAll do Kernel (ver seção III.15.9) é mais poderosa do que o Halt: faz terminar todas as CPUs virtuais de uma só vez. Uma CPU virtual também pode ficar inativa, quando, por exemplo, o seu MAIN chega ao fim do programa e todos os outros processos eram filhos dele ou de alguma geração destes. Uma CPU pode voltar à atividade se algum processo for migrado para ela.

III.14.2 Terminação Anormal

O término anormal de um programa acontece devido a um erro de execução próprio do Turbo Pascal, uso incorreto das primitivas do Kernel ou pela interferência direta do usuário com um Ctrl-Break. Em todos os casos a execução é abortada, ou emitindo uma mensagem de erro do Turbo Pascal ou do Kernel. O Kernel foi construído visando, além de eficiência, ser amigável. Por isso, a maior parte de suas primitivas tem a sua utilização checada, facili-

tando muito a codificação correta de um programa. A decisão de se incluir tal verificação de erro, em detrimento de um ganho mínimo de velocidade, pode ser argumentada com o fato da Programação Concorrente por si só já ser de uso difícil. Por essa razão, é válido retirar do programador a responsabilidade completa de uso das primitivas.

As mensagens de erro emitidas pelo Turbo Pascal são como de costume. As mensagens emitidas pelo Kernel têm o seguinte formato:

***** Kernel Error <n> (<mensagem de erro>)**
***** on Process <p> (<nome do processo>), Virtual CPU <c>**
***** at <endereço do erro>**

Como exemplo, o uso incorreto da primitiva Cobegin poderia gerar a seguinte mensagem:

```
*** Kernel Error 10 (Illegal Cobegin Use)
*** on Process 1 (MAIN-0), Virtual CPU 0
*** at 0000:01AB
```

O endereço do erro exibido na mensagem em geral pode ser utilizado no comando "Find Error" do ambiente Turbo Pascal, localizando automaticamente a linha do programa fonte onde ele aconteceu. Algumas vezes a mensagem "Target Address not Found" pode vir como resposta, indicando que a linha do programa não foi encontrada. Isto pode acontecer se o erro foi ocasionado dentro do DOS, da BIOS ou da biblioteca do Turbo Pascal, por exemplo. Em casos de erro de deadlock, uma situação mais especial, apenas a primeira linha de uma mensagem normal é exibida. Os endereços emitidos só são válidos no ambiente integrado, não fazendo sentido se o programa for "desassemblado" em um depurador como o Turbo Debugger ou similares.

Para evitar que o endereço de um erro seja dado como dentro de uma primitiva chamada, o que não esclare-

ceria sua real posição no programa do usuário, o Kernel percorre todas as chamadas de procedimentos contidas na pilha do processo corrente até encontrar uma que não seja dos módulos do ambiente Kernel. Na verdade, como cada unit tem seu próprio segmento de código, basta que todas as chamadas com esses segmentos sejam descartadas. Isto é feito através da primitiva AddCSeg (ver seção III.15.9), estendida também ao usuário. Com isso ele pode evitar a geração de endereços de erro em módulos que não acrescentem nenhuma informação útil para a solução do problema.

O Anexo VI contém todas as mensagens de erro exibidas pelo módulo Kernel e sua devida explicação. A implementação da localização de um endereço de erro será discutida na seção III.16.

III.15 As Primitivas

Nas sub-seções seguintes descreveremos a sintaxe e a semântica de cada uma das primitivas do Kernel. Por facilidade, elas foram agrupadas levando em consideração a sua finalidade.

III.15.1 Constantes, Tipos e Variáveis Predefinidas

As constantes predefinidas do Kernel só podem ser alteradas via programa fonte. Sua finalidade, em geral, é limitar certos tipos de parâmetros encontrados em algumas primitivas. São elas:

* *Version = '1.00'*. Meramente informativa, retorna a versão atual do Kernel.
* *MaxCPUs = 8*. Número máximo de CPUs virtuais que podem ser alocadas pela primitiva SetProcessors.
* *MaxPrio = 2*. Maior prioridade de um processo. A mínima é sempre zero.
* *ProcessNameSize = 8*. Tamanho máximo do nome de um processo quando do uso da primitiva SetName.

* *MinStackSize = 256*. Tamanho mínimo da pilha alocada a um processo. Este é o valor mínimo recomendável, caso o programa fonte seja alterado. Segundo orientações encontradas no manual técnico do DOS (MICROSOFT, 1986), interrupções da BIOS podem utilizar até 128 bytes na pilha, que somados aos utilizados no momento podem chegar até 256 bytes.
* *StackLowerBound = 192*. Valor mínimo que deve ter a pilha de um processo ao chamar um procedimento que tente alocar mais variáveis locais. Abaixo desse valor não é seguro rodar um programa, e portanto deve ser mantido. Este número foi tirado da implementação de stack-checking da biblioteca do Turbo Pascal.

Os seguintes tipos predefinidos são utilizados nos parâmetros das primitivas e em declaração de variáveis:

* *PidType = Word*. Definição do PID do Kernel.
* *StateType = (Ready, Waiting, Suspended, Timing, Halted, Unkown)*. Estados possíveis de um processo.
* *ProcFar = Procedure*. Tipo procedural para uso nas primitivas Spawn e LongSpawn.
* *NameStr = String[ProcessNameSize]*. Tipo de string passada como parâmetro em SetName e retornada em GetName.

A única variável predefinida é UsrIdleProc, do tipo ProcFar. É utilizada para definir uma rotina especial de "idle". O procedimento vazio *DoNothing*, que pode ser enxergado como uma variável de tipo procedural, é o valor default de UsrIdleProc.

III.15.2 Primitivas de Criação de Processos

Existem três formas distintas de se criar um processo com o Kernel, e cada uma delas será abordada em separado. De qualquer forma, algumas observações são comuns a todas, baseadas no uso dos seguintes procedimentos, funções e variáveis do Kernel:

Function GetStackDefault : Word;
Procedure SetStackDefault(StackSize : Word);
Function GetPrioDefault : Word;
Procedure SetPrioDefault(Prio : Word);
Const Save80x87 : Boolean = False;
Const CopyParentStack : Boolean = True;

Nas primitivas em que a pilha e a prioridade do processo sendo criado não forem explícitas, seus valores default serão utilizados. Estes valores podem ser retornados com as primitivas *GetStackDefault* e *GetPrioDefault* e alterados com *SetStackDefault* e *SetPrioDefault*. Nas alterações, uma checagem é realizada para garantir que o novo valor está dentro da faixa permitida. O valor default de stack é 1024 e o de prioridade é 1.

Existem duas constantes booleanas que influem no procedimento de criação de processos: *CopyParentStack* e *Save80x87*. CopyParentStack, se True (o default), faz uma cópia da pilha do processo pai para a do processo filho. Com isso, os valores das variáveis locais do pai quando da criação do processo filho são copiados para a pilha deste. Esta característica permite, por exemplo, utilizar a variável de controle de um loop de criação de processos como parâmetro em uma rotina do processo filho. O valor deste parâmetro será o da variável no momento da criação do processo filho. Já a constante Save80x87, se True (o default é False), aloca uma área para salvamento do contexto do co-processador aritmético quando da criação do processo. Durante a execução deste este salvamento pode ser habilitado ou desabilitado, com as primitivas Enable80x87Save e Disable80x87Save (ver seção III.15.7).

*** Cobegin/Also/Coend**
Procedure Cobegin;
Procedure Coend;
Procedure Also;

Esta é a forma estruturada proposta por DIJKSTRA (1965), já discutida anteriormente. A idéia é fazer com que os novos processos sejam envoltos pelas primitivas *Cobegin* e *Coend,* sendo separados entre si pela primitiva *Also.* A ilusão obtida faz com que a chamada desses procedimentos se pareça com um Begin e um End do próprio Pascal.

Exemplo III.1:

```
Cobegin;
  p1;  Also;  { Primeiro processo }
  p2;  Also;  { Segundo processo }
  for i:=1 to 10 do   { Terceiro processo }
    t3; Also;
  p4;  { Quarto processo. Não é necessário Also após este }
Coend;
```

É importante notar que os processos criados não são apenas chamadas de procedimento, mas mesmo quaisquer trechos de código, como no terceiro processo do exemplo acima. O tamanho de pilha e a prioridade dos novos processos são sempre tomados como os default. Um Cobegin/Also/Coend gera erro se apenas um processo for declarado em seu interior. Apesar da estrutura não ser checada a tempo de compilação, alguns testes são realizados durante a execução para conferir se o modo de uso está correto.

Uma característica bastante interessante do Kernel é que estruturas Cobegin/Also/Coend podem ser aninhadas estática e dinamicamente. Com isso podemos criar várias gerações de uma família tendo execuções independentes. No aninhamento estático (Exemplo III.2), as estruturas são codificadas uma dentro da outra, na seqüência de leitura do programa. No aninhamento dinâmico (Exemplo III.3), o fluxo de um processo pertinente a uma estrutura Cobegin/Also/Coend acaba por encontrar uma outra estrutura deste tipo, suspendendo sua execução e criando processos

filhos dele.

Exemplo III.2:

```
Cobegin;
   p1; Also;  { Primeiro processo do Cobegin mais externo }
   Cobegin;   { Segundo processo do Cobegin mais externo }
     p2_1; Also; { Primeiro proc. do Cobegin mais interno }
     p2_2;       { Segundo proc. do Cobegin mais interno }
   Coend; Also;
   p3;         { Terceiro processo do Cobegin mais externo }
Coend;
```

O exemplo anterior mostra um aninhamento estático. Sua execução redunda na criação de cinco processos: três relativos à estrutura mais externa e dois relativos à mais interna. A estrutura mais externa só termina quando p1, p3 e a estrutura mais interna acabarem. Esta por sua vez só termina quando p2_1 e p2_2 acabarem.

Exemplo III.3:

```
Procedure p1;
begin
Cobegin;
   p1_1; Also;
   p1_2;
Coend;
end;
:

Cobegin
   p1; Also;
   p2;
Coend;
:
```

Neste exemplo, o Cobegin/Also/Coend mais externo criará dois processos, um chamando o procedimento p1 e o outro o p2. O procedimento p1 por sua vez executa um outro Cobegin/Also/Coend, que se suspende dando lugar à execução de dois processos, que chamam os procedimentos p1_1 e p1_2. A estrutura mais externa só termina quando todas as mais interna acabarem.

O nível máximo de aninhamento, tanto no caso estático como no dinâmico, é determinado tão somente pela lógica do programa do usuário. O Kernel não impõe nenhuma restrição a essa profundidade.

*** Fork/LongFork**

```
Procedure Fork(Var Pid : PidType);
Procedure LongFork(Prio,StackSize : Word;
                   Var Pid : PidType);
Function  ChildProcess : Boolean;
```

Este tipo de mecanismo foi proposto por CONWAY (1963), sendo uma forma não muito elegante de criar processos, porém a mais poderosa. A idéia de um *Fork* é derivar um outro fluxo de programa após a sua chamada, rodando a partir dela dois processos: o processo já existente, que fez a chamada, e o seu filho. Para se diferenciar entre um e outro temos a função booleana *ChildProcess*, que em geral deve ser utilizada logo após o Fork (ou *LongFork*). Utilizando esta função em um comando condicional pode-se decidir que fluxos tomarão os processos pai e filho.

Exemplo III.4:

```
Fork(x);
if ChildProcess then
  t1
else
  t2;
```

O exemplo acima mostra a criação de um novo processo com a primitiva Fork. Após a sua execução, tanto o processo que a chamou quanto o processo recém-criado executarão o "if" da próxima linha. O novo processo achará verdadeira a condição do teste, executando portanto o procedimento t1. Já o processo pai cairá no "else" da condição, executando portanto o procedimento t2.

Todo este esquema de funcionamento também é válido para o LongFork. A diferença entre este e o Fork reside apenas na não aceitação dos valores default de prioridade e tamanho de pilha, criando o novo processo com outros valores. Quanto à primitiva ChildProcess, esta só pode ser utilizada uma única vez para cada processo criado, e somente por um Fork ou LongFork (com outros métodos ela não faz sentido).

Processos criados com Fork não devem sair do escopo atual para trás, ou seja, retornar do procedimento de onde foram criados, seja com o "end" do procedimento ou por um "Exit". O problema é que estes processos não possuem uma história de chamadas na pilha, o que ocasiona uma pane no programa se isto acontecer. Tanto Fork como LongFork retornam o PID do novo processo no parâmetro de saída Pid.

*** Spawn/LongSpawn**

```
Function Spawn(Task : ProcFar) : Word;
Function LongSpawn(Task : ProcFar;
                   Prio,StackSize : Word) : Word;
```

Esta forma de criação de processos é semelhante à encontrada em Modula-2. O processo a ser criado deve necessariamente ser um procedimento, passado como parâmetro para *Spawn* ou *LongSpawn* (a diferença entre os dois é similar à discutida em Fork/LongFork). Mais ainda, este procedimento deve ser compilado no modo "Far" do Turbo Pascal (diretiva F+), de forma a ser compatível com o tipo

predefinido ProcFar. O processo que executa um Spawn continua normalmente após sua chamada, enquanto o processo recém-criado (seu filho) iniciará sua execução no início do procedimento determinado. A vantagem desta primitiva é que este novo processo se destrói automaticamente ao alcançar o fim do procedimento (ou com a chamada de "Exit", que faz justamente isto). O valor retornado pelas duas primitivas é o PID do novo processo.

Exemplo III.5:

```
i:=Spawn(@p1);
```

Esta única linha de código cria um novo processo iniciando no procedimento p1, enquanto seu pai prossegue normalmente. Uma observação neste tipo de criação é sobre a constante CopyParentStack: independentemente de seu valor, a pilha do processo pai nunca é copiada, por não fazer sentido neste caso.

III.15.3 Primitivas de Destruição de Processos

Procedure Kill(Pid : PidType);
Procedure KillMyself;

Processos podem ser explicitamente destruídos utilizando-se estas primitivas, conforme descrito na seção III.7. A primitiva *Kill* destrói simplesmente um processo filho de quem a executa (junto com os descendentes deste), passando como parâmetro seu PID. A primitiva *KillMyself* destrói o próprio processo que a chamou, junto com seus descendentes.

III.15.4 Primitivas de Exclusão Mútua

Procedure Lock;
Procedure Unlock;
Procedure LockFile(Var F : Text);

Procedure SetHeapMutex;
Procedure ResetHeapMutex;

As primitivas *Lock* e *Unlock* simplesmente desabilitam e habilitam, respectivamente, a atuação do despachante do Kernel (ver seção III.8). A primitiva *LockFile* garante a exclusão mútua em operações de entrada e saída em um arquivo do tipo Text (ver seção III.8). *SetHeapMutex* e *ResetHeapMutex* habilitam e desabilitam a exclusão mútua no manuseio da heap (ver seção III.8).

A única observação aqui é quanto à primitiva LockFile. Esta deve ser utilizada *somente* em arquivos fechados, e após o devido "Assign", que pode ser o do Turbo Pascal ou um feito pelo programador (como "AssignCrt" - ver manual da linguagem). Seu uso seria como no exemplo a seguir:

Exemplo III.6:

```
Assign(f,'TESTE');
LockFile(f);
Rewrite(f);
:
```

III.15.5 Primitivas de Sincronização

Procedure Join(n : Word);
Procedure Barrier(n : Word);

A primitiva *Join* foi baseada na descrição de PETERSON (1985). Basicamente, todos os processos que chamam um mesmo Join morrem, à exceção do último. Ou seja, se uma linha de programa contém um Join(5), por exemplo, os quatro primeiros processos que o executarem morrerão, continuando apenas o quinto e último que passar por ele. Desta maneira juntamos os fluxos de vários processos em apenas um, obtendo um tipo de sincronização.

Vale apenas ressaltar que, uma vez que a linha que contém o Join é a mesma para todos os processos, o seu parâmetro naturalmente deve ser igual para todos. Se for passada uma variável como parâmetro, esta não deve ter seu valor alterado por nenhum dos processos, sob pena de não funcionar a primitiva. Além disso, após uma execução completa de um Join (quando o último processo prossegue) ele é reiniciado para uso idêntico, apesar desse uso ser bastante raro. A semântica do Join faz uma chamada automática da primitiva OrphanChildren (ver seção III.15.7).

O *Barrier* é uma primitiva de sincronização mais encontrada em programas paralelos de fato. Serve para parar a execução de um determinado número de processos (seu parâmetro) até que todos eles tenham atingido esta primitiva, sincronizando-se neste ponto. Em seguida todos eles continuam normalmente.

III.15.6 Primitivas de Gerência de Processador

```
Procedure Yield;
Procedure Transfer(Pid : PidType);
Procedure Detach;
Procedure SetTimeSlice(NewFrameType : Word);
Procedure SetProcessors(n : Word);
Function  CPUid : Word;
```

A primitiva *Yield* passa simplesmente o uso processador para o próximo processo na lista de prontos. *Transfer* e *Detach* são mais específicas: a primeira transfere o processador para um de seus processos filhos, enquanto a última transfere o controle de volta para o seu pai. Yield, Transfer e Detach podem ser aplicadas em co-rotinas, como descrito na seção II.7.

SetTimeSlice controla o time-slice utilizado pelo Kernel. Valores entre 0 e 15 são aceitos, sendo zero o

default. Quanto maior o parâmetro de SetTimeSlice (seu "frame"), mais troca de contexto haverá, e portanto mais concorrência e também overhead. Apesar de 15 ser o valor máximo, o equipamento hóspede o limita mais ainda: em um micro AT-386 com clock de 20MHz, um "frame" de 9 é o maior valor aceito, sem que o programa "congele". Esta pane acontece quando a freqüência das interrupções do relógio de hardware é tão grande que a sua rotina de atendimento não dá conta, provocando um "overrun".

Para habilitar e desabilitar a preempção no Kernel, existem duas constantes predefinidas que podem ser utilizadas como parâmetro: *On* e *Off* (o default é On). Com elas pode-se implementar um sistema não-preemptivo baseado em co-rotinas, por exemplo, chamando SetTimeSlice(Off). Ou ainda, utilizá-las na criação de novos processos, quando se deseja que os processos filhos só comecem a rodar quando o último deles for criado (em um programa que crie em um loop 100 processos, por exemplo, é bem possível que haja uma preempção do processo criador e um dos processos filhos comece a rodar).

O uso de SetTimeSlice merece duas considerações. Sua implementação no Kernel não garante exclusão mútua, uma vez que não faz sentido mais de um SetTimeSlice simultâneo. A outra diz respeito a uma situação anômala, mas que deve ser registrada. Se SetTimeSlice for utilizada em um loop com parâmetros distintos a cada iteração, o relógio da BIOS pode atrasar ou até quase parar. Isto acontece porque o chip responsável pela interrupção do relógio de hardware (ver seção III.9) é reprogramado a cada "frame" novo, o que não chega a ser um "bug", apenas mau uso da primitiva.

As primitivas *SetProcessors* e *CPUid* são utilizadas em programas com mais de uma CPU virtual. SetProcessors recebe como parâmetro o número de CPUs virtuais que irão executar no programa (contando com a CPU 0, já executando). A partir de sua chamada, fluxos

independentes correspondendo cada um a um novo programa principal começam a rodar. O modelo das CPUs virtuais foi extensamente discutido na seção III.2.

SetProcessors é a peça-chave da implementação do Kernel em um ambiente real, com mais processadores. A forma sugerida seria cada processador físico receber de um processador mestre, através do sub-sistema de comunicação, quantas e quais CPUs virtuais rodar. A idéia das CPUs virtuais, portanto, continuaria válida: mesmo em um sistema com 3 processadores físicos, por exemplo, poderíamos ter 10 CPUs virtuais; cada um dos três processadores físicos poderia simular um determinado número de CPUs lógicas.

A primitiva CPUid é um apoio na confecção de programas com CPUs virtuais. Sua chamada retorna o número da CPU virtual correntemente executando, o que serve como base na tomada de decisões e desvios de fluxo. Com isso, o mesmo trecho de código pode ser executado por diversas CPUs virtuais, derivando quando necessário para operações específicas de cada CPU.

III.15.7 Primitivas de Gerência de Processos

```
Procedure Suspend(Pid : PidType);
Procedure Resume(Pid : PidType);
Procedure Migrate(Pid : PidType; NewCPU : Word);
Procedure SetPrio(Prio : Word);
Procedure SetChildPrio(Pid : PidType; Prio : Word);
Function  GetPid : PidType;
Function  GetParentPid : PidType;
Function  GetState(Pid : PidType) : StateType;
Procedure SetName(Name : String);
Function  GetName : NameStr;
Procedure OrphanChild(Pid : PidType);
Procedure OrphanChildren;
Procedure Enable80x87Save;
Procedure Disable80x87Save;
```

As primitivas *Suspend* e *Resume* trabalham em conjunto. Sua ação é bastante simples: Suspend suspende a execução de um processo filho do processo Running, enquanto Resume o faz continuar. Processos suspensos não ficam em nenhuma lista do Kernel, por ser desnecessário. Outro detalhe importante é que, se o processo suspenso não estava na Ready List, ele apenas será marcado como tal, e sua pilha alterada de forma a executar num escalonamento futuro ele próprio a sua suspensão. Este é o mesmo esquema utilizado na destruição de processos, descrito na seção III.7.

A primitiva *Migrate* só pode ser utilizada em programas com mais de uma CPU virtual. Sua função é transferir a execução de um processo filho do Running de uma CPU virtual para outra. Isto é bastante útil para sistemas que pesquisam efeitos de balanceamento de carga. Entretanto, sua implementação em um ambiente real, com mais processadores físicos, está descartada. Esta seria uma tarefa bastante árdua, senão impraticável, com a arquitetura de um micro IBM-PC. O problema está nas referências a posições de memória que um processo faz em seu código e nas contidas em sua pilha. Mudando para um outro processador físico, estas referências, com enorme probabilidade, serão diferentes, e não há como remediar esta situação.

SetPrio e *SetChildPrio* são primitivas para alteração da prioridade de um processo. SetPrio altera a prioridade do processo Running, enquanto SetChildPrio altera a de um de seus filhos. Mudanças de prioridade envolvem trocas de posição na Ready List, afetando diretamente o conjunto de processos escalonáveis em um dado momento. Um uso interessante de SetPrio é SetPrio(MaxPrio), quando nenhum outro processo possui também a prioridade máxima: ele garante exclusão mútua até sua prioridade ficar mais baixa novamente.

As primitivas *GetPid* e *GetParentPid* retornam o PID do processo Running e o de seu pai, respectivamente. Devem ser utilizados em programas mais avançados, geralmente na decisão de fluxo de um processo ou em outras ocasiões que necessitem do PID.

GetState retorna o estado atual de um processo filho do Running, dentre os definidos no tipo StateType (naturalmente este estado não pode ser o "Running"). A aplicação desta primitiva deve ser feita com exclusão mútua se o programa tiver preempção, pois de outra forma o estado retornado pode não ser mais válido quando esta informação for utilizada. Esta função também existe na linguagem Concurrent C (GEHANI, 1986).

As primitivas *SetName* e *GetName* têm maior finalidade em conjunto com o módulo Debugger (Capítulo VI). SetName estabelece um nome para o processo Running, enquanto GetName o retorna.

OrphanChild e *OrphanChildren* fazem alterações na família do processo Running. OrphanChild exclui um processo filho, em especial, de sua família. Essa exclusão é similar à realizada quando da destruição de um processo, mas neste caso os processos continuam vivos. Quanto ao processo filho excluído, este passa a se considerar "sem pai", isto é, com um ponteiro para o pai (Parent) igual a Nil. OrphanChildren procede de modo idêntico, mas realizando uma operação coletiva: todos os filhos do Running são excluídos. A finalidade dessas primitivas é libertar processos da dependência de seu pai. Como descrito na seção III.7, ao morrer um processo leva consigo sua família, a família destes e assim por diante. Como isso nem sempre é desejável, OrphanChild e OrphanChildren podem ser usadas para dar "emancipação" aos processos.

Enable80x87Save e *Disable80x87Save* estabelecem se o processo Running, quando sofrer uma preempção e também

quando for reescalonado, salvará e restaurará o contexto do co-processador aritmético. Esta operação só é necessária se tal co-processador realmente existir (não for emulado) e se mais de um processo o utilizar simultaneamente. Além disso, é necessário que processos que o façam tenham sido criados com a constante predefinida Save80x87 tendo o valor True, *sob pena de pane do programa*. O processo MAIN-0, como descrito na seção III.6.2, é criado desta forma, mas desabilita automaticamente o salvamento do contexto do co-processador. Esta medida visa poupar tempo, pois na maioria dos casos seria uma operação desnecessária.

III.15.8 Primitivas de Tempo Real

```
Procedure SetTimeout(ms : Longint);
Function  Timedout : Boolean;
Procedure Delay(ms : Longint);
Procedure Schedule(Hour,Min,Sec : Word);
```

SetTimeout é utilizada para estabelecer um limite de tempo (timeout) quando o processo entrar em alguma fila de espera. Ela é uma operação bastante poderosa e genérica, podendo ser aplicada em quaisquer estruturas deste tipo, como as encontradas nos módulo ShareMem (Capítulo IV) e MsgPass (Capítulo V) ou em futuras implementações. Uma vez estabelecido um timeout, todas as operações que ocasionem uma espera estarão restritas a este valor, até que este seja alterado ou zerado. O valor zero significa não haver mais limite de tempo.

A primitiva *Timedout*, uma função booleana, deve ser utilizada após operações que possam ocasionar uma espera, se o processo tem um valor de timeout maior que zero. Isto por vezes é necessário para distinguir se a operação foi realizada com sucesso ou se retornou devido ao timeout imposto.

A primitiva *Delay*, similar à do módulo CRT do Turbo Pascal, se diferencia daquela por não gastar inutilmente o tempo solicitado. Ao invés disso, este tempo é utilizado para a execução de outros processos, o que permite um ganho no programa. Contudo, por ser uma redefinição de um procedimento já existente em outro módulo, é importante observar a ordem dos módulos Kernel e CRT na cláusula "Uses". Se a palavra Kernel vier depois de CRT nesta cláusula, a primitiva do Kernel será a utilizada; caso contrário, será a da CRT. A desvantagem de usar a da CRT é que o processo que a chamou não sai da Ready List, fazendo um busy-waiting.

A primitiva *Schedule* estabelece um determinado horário, com hora, minuto e segundo, para um processo voltar a rodar. Ele é transferido da Ready List para a Time List até que aquele seja atingido. Poderá haver algum atraso na hora de acordar dependendo do número de processos existentes na Ready List.

III.15.9 Primitivas de Terminação de Programa

Procedure HaltAll;
Procedure InstallKernelExit;
Procedure AddCSeg(CS : Word);

Essas três primitivas já foram apresentadas nas seções III.14.1 e III.14.2. O efeito de *HaltAll* é terminar a execução de *todas* as CPus virtuais do programa. *InstallKernelExit* deve ser utilizada em programas com mais de uma CPU virtual, em que o programador altera o procedimento de saída usual ("ExitProc"). *AddCSeg* exclui um determinado módulo do usuário na localização de um erro, evitando mensagens com endereços que não acrescentem nenhuma informação útil. Esta primitiva é utilizada pelos quatro módulos que compõem o ambiente Kernel. Desta forma, se por exemplo uma primitiva do Kernel resulta em erro, o endereço fornecido na mensagem não se encontra dentro dela,

mas no programa do usuário.

III.15.10 Primitivas de Controle de Execução Não-determinística

Function Select(n : Word) : Word;
Function PriSelect(n : Word) : Word;
Procedure When(Guard : Boolean);

O funcionamento geral de uma execução não-determinística já foi descrito na seção II.9, sob o título de "comandos alternativos". No Kernel, tais comandos foram implementados pelas primitivas Select (ou PriSelect) e When. Um *Select* é a evolução de uma estrutura "Case" do Pascal, onde as opções são testadas de forma aleatória, e que só termina quando uma das opções escolhidas tenha sua condição satisfeita ou todas forem testadas sem êxito. Neste último caso, o próprio "else" pertencente à estrutura pode ser utilizado para a tomada de alguma decisão. O parâmetro de um Select é justamente o número total de opções que devem ser testadas no Case. As opções necessariamente devem iniciar a partir do número 1.

O *PriSelect* é idêntico ao Select, mas a ordem em que as opções são testadas é a da posição no texto do programa, como em um Case normal. O *When* deve ser utilizado em substituição a um possível "if" de teste, para saber se as condições para uma opção ser aceita são favoráveis. Quando este não é utilizado, é considerado que a opção é válida. O exemplo seguinte esclarecerá um pouco mais essa estrutura:

Exemplo III.7:

```
Repeat
  Selected:=True;
  Case Select(2) of
    1: Begin
         When(j = 100);
         writeln(1);
       End;
    2: Begin
         When(j = 100);
         writeln(2);
       End;
    Else
       Selected:=False;
  End;
  Inc(j);
  Until Selected;
```

Esse trecho de programa realiza um loop que espera a variável j atingir o valor 100. Enquanto j for menor que 100, o Case seleciona as opções 1 e 2, em ordem aleatória, a cada passo. Como o When existente nas duas opções impedem que estas sejam selecionadas antes de j atingir 100, o Else do Case será executado nas primeiras 99 iterações. Na centésima, então, uma das duas opções já terá sua condição satisfeita, e escreverá na tela o número 1 ou o 2, indicando a escolhida. Se um PriSelect fosse utilizado neste exemplo, a opção 1 sempre seria a escolhida, visto que as condições no When de ambas são iguais. Aliás, esta codificação não passa mesmo de um exemplo, uma vez que os dois When poderiam ser removidos e um teste "if j = 100" colocado antes do Case ser executado.

III.16 Detalhes de Implementação

Nesta seção abordaremos alguns tópicos de implementação. Como por vezes não há nenhuma relação entre

eles, faremos uma divisão por itens.

* Utilização de interrupções de software. O Kernel utiliza as interrupções de número $86 a $8F (valores hexadecimais), originalmente alocadas para o Basic residente que os primeiros IBM-PCs possuíam. O único cuidado que se deve ter é alguma incompatibilidade com outros programas que as usem, se estes forem TSRs (Terminate and Stay Resident). Ao acabar, o Kernel não restaura os vetores originais.

* Tamanho máximo de uma pilha. Como uma pilha é alocada dinamicamente, sofre as mesmas restrições impostas pelo Turbo Pascal a variáveis deste tipo: o tamanho máximo de 65520 bytes. Na verdade, no caso de pilhas essa restrição não chega a ser um problema, visto que o valor é bastante alto.

* Código máximo de um processo numa estrutura Cobegin/Also/Coend. Processos criados por essa estrutura têm seu fim delimitados por um Also ou pelo Coend, no caso do último. Como para criar um processo desta forma é necessário varrer diretamente o código gerado à procura de uma chamada de Also ou Coend, a sua extensão é limitada a 1024 bytes. Esta limitação visa facilitar a checagem de utilização dessa estrutura, sendo o valor 1024 suficiente para a quase totalidade das aplicações. Entende-se por "código gerado" nessa estrutura apenas os bytes realmente dentro dela. Desta forma, se um processo contém uma chamada de procedimento, apenas o código da chamada está sendo contado, não os bytes do procedimento em si.

* Stack-Checking. Como cada processo tem a sua própria pilha, não faria sentido deixar o stack-checking normalmente feito pelo Turbo Pascal através da diretiva S+. Isto, inclusive, poderia fazer com que um programa parasse erroneamente por um estouro de pilha. A solução

foi alterar o stack-checking para um particular ao Kernel, que testa se a pilha do processo corrente tem condições de alocar os bytes requisitados. Como a chamada deste teste de pilha é feita automaticamente pelo Turbo Pascal, não é possível desviá-la diretamente, sendo necessária uma técnica mais forte: os primeiros bytes da rotina de stack-checking, dentro da biblioteca do Turbo Pascal, são alterados para um "jump" intersegmentos direto, caindo em uma rotina dentro do Kernel. O método para se localizar o endereço inicial na biblioteca é ler diretamente o código gerado pela inclusão de uma das chamadas de stack-checking.

Todas as rotinas dentro do Kernel executam sem nenhum teste de pilha, visando maior velocidade. Isto não chega a criar incômodos para o usuário, uma vez que uma das maiores atenções na escrita de um programa concorrente com o Kernel é quanto ao tamanho de pilha.

* Exclusão mútua da heap. Esta característica bastante útil do Kernel é realizada desviando todas as rotinas de manuseio da heap (inclusive por tipos Object) para um procedimento de exclusão mútua dentro do Kernel. Seu funcionamento é similar ao de um depurador: os primeiros bytes da rotina desviam para o procedimento do Kernel, que os possui guardados e são recolocados em seus lugares; os bytes no endereço de retorno da rotina são alterados de forma idêntica, de forma que, ao terminar a sua execução, os primeiros bytes voltam a ser alterados e os do endereço de retorno reestabelecidos. A função básica do procedimento do Kernel é fazer um Lock ao entrar na rotina e um Unlock ao sair dela. Os endereços das rotinas de manuseio da heap, que não são retornados pela função Addr, são descobertos varrendo-se diretamente um código gerado para esse propósito.

* Exclusão mútua do DOS. As formas de garantir a exclusão mútua de um processo utilizando o DOS são duas:

revetorizar a interrupção $21 (chamada do DOS), fazendo um Lock ao entrar e um Unlock ao sair dela, ou testar uma flag interna do DOS denominada DOSBusyFlag dentro do despachante. As duas maneiras foram testadas e provaram ser de velocidades similares, sendo preferida a última por facilidade.

* Localização do Endereço de Erro. Para gerar corretamente o endereço aonde ocorreu um erro do Kernel, devemos percorrer o "display" dos registros de ativação dos procedimentos chamados (AHO, 1986), até encontrarmos um segmento de código não pertencente ao ambiente Kernel. O processo é bastante simples, apesar de conceitualmente complicado. Restaura-se primeiramente a pilha do último procedimento chamado. Depois faz-se um loop que acessa o seu registro de ativação e restaura a pilha do último procedimento chamado. Enquanto o segmento do endereço de retorno estiver dentro do ambiente Kernel ou em módulos determinados pelo usuário com a primitiva AddCSeg, o loop prossegue. Ao terminar, os registradores CS e IP contêm o endereço do erro.

Uma vez que esse loop é uma aceleração dos retornos normais de todos os procedimentos chamados até o momento, é óbvio que após ele o programa não pode continuar. Para isso seria necessário salvar todo o encadeamento desfeito, mas, como o Kernel só gera erros fatais, isto foi descartado.

III.17 Considerações Finais

Alguns acréscimos podem ser feitos ao Kernel, em uma versão futura, de modo a aumentar o seu potencial de uso. As idéias em mente incluem o aproveitamento das chamadas páginas de vídeo do IBM-PC, onde poderíamos simular a presença de mais monitores em um programa distribuído; um sistema para facilitar a escrita simultânea na tela por mais de um processo, quando estes utilizam

atributos de vídeo (cores, tipo do cursor, etc) diferentes e trabalham com janelas; mostrar automaticamente a linha do programa fonte onde aconteceu um erro, inserindo caracteres no buffer de teclado de modo a chamar o comando "Find Error"; criar uma interface para o Kernel ser utilizado a partir de qualquer linguagem.

CAPITULO IV: O MODULO DE MEMORIA COMPARTILHADA

Neste capítulo serão apresentadas as estruturas que compõem o Módulo de Memória Compartilhada, doravante chamado de *ShareMem*. Este módulo fornece tipos de dados e operações sobre eles para a solução de problemas de exclusão mútua e de sincronização, descritos no Capítulo II.

IV.1 As Estruturas

Seguindo a filosofia do Kernel de prover diversas maneiras diferentes de realizar a mesma função, para ser utilizado como instrumento de laboratório ou ter abragência de uso, o módulo ShareMem implementa vários mecanismos que por vezes são equivalentes. Esta política também foi utilizada no trabalho de KNOP (1990), e se justifica pelas diferenças de uso, tipo e tamanho das regiões críticas encontradas comumente nos programas. Mais uma vez, vale argumentar que a quantidade de estruturas fornecida não aumenta muito o aprendizado do usuário para este iniciar um projeto, pois elas são independentes e podem ser escolhidas para estudo e uso em separado. Desta forma, seria o mesmo que prover um ambiente com um ou dois mecanismos apenas.

As seguintes estruturas estão disponíveis: **Semáforos, Semáforos Binários, EventCounts, Sequencers, Semáforos de Conjunto de Recursos, Variáveis Compartilhadas (Shared), e Signalling Regions.** Nas próximas seções descreveremos cada uma em separado, enquanto o Anexo III contém a interface completa do módulo ShareMem. Quanto à eficiência de cada uma para a exclusão mútua de regiões críticas, desprezando-se a elegância, a clareza, a facilidade de uso e a segurança contra erros das construções, consultar o Anexo IX.

Cabe aqui uma observação muito importante na utilização de qualquer uma dessas estruturas, seja para

exclusão mútua ou sincronização de processos. A variável declarada como um dos tipos exportados por ShareMem *deve ser global ao programa*. A razão disto é que, sendo local, ela seria alocada na pilha de um dos processos, só sendo acessível e alterável por este.

IV.2 Semáforos

Essa é uma das primitivas de memória compartilhada mais antigas, criada por DIJKSTRA (1965), e tem como operações básicas os procedimentos P e V. Serve tanto para garantia de exclusão mútua como para sincronização entre processos. A forma de espera implementada em ShareMem é a da fila, ou seja, o processo fica bloqueado esperando o devido sinal de que pode continuar.

Tipo em ShareMem: *Semaphore*
Operações:

```
Procedure InitSemaphore(Var S : Semaphore;
                        Value : Integer);
Procedure P(Var S : Semaphore);
Procedure V(Var S : Semaphore);
Procedure PPrio(Var S : Semaphore; Prio : Integer);
Function  SemaphoreCount(Var S : Semaphore) : Integer;
Function  Awaited(Var S : Semaphore) : Boolean;
```

O procedimento *InitSemaphore* é o único que pode ser utilizado para iniciar o semáforo, mas não há nenhuma checagem para o seu uso por mais de uma vez. Ele recebe o semáforo e o seu valor inicial como parâmetros, que em geral determina o número de recursos que serão compartilhados.

Os procedimentos *P* e *V* fazem as operações usuais sobre um semáforo. A forma de implementação é a mesma sugerida em MADNICK (1974) e PETERSON (1985), ou seja, um P

sempre decrementa e um V sempre incrementa o valor do semáforo. Desta forma um valor positivo corresponde aos recursos disponíveis e um negativo, em termos absolutos, ao processos bloqueados esperando a liberação de um recurso. A função SemaphoreCount (PERIHELION, 1989) retorna justamente este contador do semáforo, mas deve ser usada com cuidado. A chamada desta função e a decisão do que fazer com este valor devem ser feitas com exclusão mútua, pois se houver uma preempção entre essas duas operações o valor do semáforo poderá estar incorreto.

O procedimento *PPrio* executa um "P" com uma dada prioridade (-32768 a 32767), válida se o processo ficar bloqueado na fila de espera do semáforo. Após a liberação de um recurso através de um "V", o processo de maior prioridade tem preferência para prosseguir. O procedimento P opera com uma prioridade igual a zero, o que permite a combinação com chamadas de PPrio com prioridades maiores ou menores. A idéia de se incluir prioridades em semáforos é difundida entre vários autores, como por exemplo LEBLANC (1984).

A função *Awaited* também é uma extensão das primitivas originais dos semáforos (LEBLANC, 1984). Ela retorna os valores True ou False dependendo de existir ou não algum processo esperando em um dado semáforo. Esta fila de espera de um semáforo pode ser observada durante uma depuração com o procedimento SpySemaphore, do módulo Debugger, descrito no Capítulo VI.

IV.3 Semáforos Binários

Este tipo de semáforo é, na verdade, um caso especial de um semáforo dito geral, descrito na seção anterior. Ele controla o uso de apenas um recurso, daí ser binário, e é usado fundamentalmente para exclusões mútuas em regiões críticas. Mas, para não ser redundante e prover ao usuário uma nova forma de espera por um recurso, este

tipo de semáforo a realiza através de busy-waiting, que é um loop que testa o valor do semáforo e abandona imediatamente o processador se este for igual a zero (o recurso ainda não está disponível) (PETERSON, 1985).

A vantagem do Semáforo Binário sobre o Semáforo, em particular nesta implementação, é uma estrutura de dados e operações mais simples e a rapidez, dependendo do caso, no controle de regiões críticas. Isto porque diversas vezes é mais demorado tirar e colocar um processo de volta à Ready List do que simplesmente esperar o tempo passar testando uma condição se tornar verdadeira.

Tipo em Sharemem: *BinSemaphore*
Operações:

```
Procedure InitBinSemaphore(Var B : BinSemaphore);
Procedure PBin(Var B : BinSemaphore);
Procedure VBin(Var B : BinSemaphore);
```

As operações *PBin* e *VBin* do BinSemaphore são basicamente as mesmas P e V do tipo Semaphore, a nível de utilização (a implementação é bem diferente). O procedimento de iniciação InitBinSemaphore, como se nota, só recebe uma variável do tipo BinSemaphore como parâmetro, uma vez que o número de recursos inicialmente é sempre igual a um.

IV.4 EventCounts e Sequencers

Estas duas estruturas, utilizadas em geral de forma complementar, foram criadas por REED (1979). Apesar de não tanto difundidas quanto os semáforos, elas são bastante simples e ao mesmo tempo bastante completas no tocante à sua equivalência com aqueles. Os EventCounts e os Sequencers permitem aos processos controlarem diretamente a

ordem dos eventos, ao invés de utilizarem exclusão mútua para proteger manipulações de variáveis que controlam tais ordens. Foram incluídos em ShareMem mais por serem uma ferramenta de construção de outras estruturas e portanto como mecanismo de pesquisa.

Tanto os EventCounts quanto os Sequencers são variáveis inteiras não-decrescentes iniciadas com o valor zero. Os EventCounts mantêm a contagem do número de eventos ocorridos em uma classe particular, enquanto os Sequencers permitem ordenar arbitrariamente esses eventos.

Tipo em ShareMem: *EventCount*
Operações:

```
Procedure InitEventCount(Var E : EventCount);
Procedure Advance(Var E : EventCount);
Function  ReadE(Var E : EventCount) : EventCount;
Procedure Await(Var E : EventCount; v : Longint);
```

O procedimento *InitEventCount* é apenas a iniciação de um EventCount com o valor zero. O uso de um procedimento para uma operação tão simples é preferível por questões de clareza e utilização correta de um tipo de dado (AHO, 1983).

A primitiva *Advance* sinaliza a ocorrência de um evento da classe associada ao EventCount passado como parâmetro. Seu efeito é o de incrementar o EventCount de uma unidade. Para obter o valor de um EventCount, temos duas primitivas: *ReadE* e *Await*. ReadE retorna simplesmente o valor de um Eventcount, enquanto Await bloqueia o processo que a chamou até que o EventCount passado como primeiro parâmetro atinja o valor "v" passado como segundo parâmetro. Em ambas, o valor do EventCount ao final da operação é garantido ser um "lower-bound" do seu valor corrente, pois outras operações similares podem ter acontecido durante sua execução.

Apesar da sugestão de Reed de se implementar Await evitando busy-waiting, em ShareMem foi escolhida esta última forma, fazendo um "loop" abandonando o processador enquanto o EventCount não atingir o valor desejado. Por esta razão, os EventCounts devem ser utilizados com cuidado, porque o Kernel não pode detectar situações de deadlock quando existem processos ativos em busy-waiting.

Tipo em ShareMem: *Sequencer*
Operações:

Procedure InitSequencer(Var Seq : Sequencer);
Function Ticket(Var Seq : Sequencer) : Longint;

Um Sequencer só possui duas primitivas: a *InitSequencer,* que o inicia com zero, semelhante à InitEventCount; e *Ticket,* que retorna sempre um valor crescente e diferente para cada chamada. A idéia por trás desta primitiva é a de uma máquina automática que é utilizada para controlar a ordem de um serviço em determinadas lojas. A máquina fornece um ticket numerado para cada um que solicita um serviço, e comparando esses números (todos diferentes) podemos saber quem chegou primeiro.

Na implementação de ShareMem tanto EventCount quanto Sequencer estão limitados ao número máximo de 2.147.483.648, o que parece ser suficiente para a maioria das aplicações.

Para exemplificar o uso de EventCounts e Sequencers, podemos construir um semáforo utilizando-os. Seja o tipo NewSemaphore, definido por um record com um campo E (um EventCount), um campo T (um Sequencer) e ainda um campo I (seu valor inicial). As operações P e V podem ser definidas da seguinte maneira:

```
Procedure P(Var S : NewSemaphore);
begin
  Await(S.E,Ticket(S.T)-S.I+1);
end;

Procedure V(Var S : NewSemaphore);
begin
  Advance(S.E);
end;
```

A operação Ticket atribui um número a cada processo que chama P. O processo espera então a sua vez ser anunciada, que ocorrerá quando um número correspondente de eventos V for assinalado, descontado o valor inicial do semáforo.

Outros exemplos, como a construção de um P simultâneo em dois semáforos, podem ser encontrados em REED (1979).

IV.5 Semáforos de Conjunto de Recursos (Resource Set Semaphores)

Este mecanismo, devido a Keedy, Ramamohanarao e Rosenberg (KEEDY, 1979), é um complemento à estrutura elegante de um semáforo. Em muitas aplicações, é necessária a introdução de um inconveniente semáforo de exclusão mútua para se descobrir qual recurso de um conjunto foi alocado por uma operação P. A proposta de um Resource Set Semaphore, chamado em ShareMem de RSemaphore, é que a primitiva P retorne também um identificador indicando qual recurso do conjunto foi alocado (ao invés de simplesmente *que* um recurso foi alocado) e a primitiva V receba como parâmetro a identidade do recurso liberado.

Tipo em ShareMem: *RSemaphore*
Operações:

```
Procedure InitRSemaphore(Var RS : RSemaphore;
                         Resources : Word);
Procedure RP(Var RS : RSemaphore;
             Var ResourceAlloc : Byte);
Procedure RV(Var RS : RSemaphore;
             Var ResourceFreed : Byte);
```

O exemplo de utilização dado por Keedy et alii é o da alocação de impressoras, como mostrado a seguir. Considerando um semáforo geral PRINTER e um semáforo binário MUTEX, temos:

```
P(PRINTER);
P(MUTEX);
{ Determina qual impressora foi alocada }
V(MUTEX);
{ Usa a impressora... }
P(MUTEX);
{ Avisa que impressora foi liberada }
V(PRINTER);
V(MUTEX);
```

O exemplo mostra a dificuldade do uso de semáforos usuais para resolver o problema. Após conseguirmos uma impressora com uma operação P ainda temos que descobrir quem é ela, o que involve uma exclusão mútua desnecessária e deselegante; o mesmo acontece ao liberá-la. Utilizando um RSemaphore, escreveríamos:

```
RP(PRINTER,n);
{ Usa a impressora n }
RV(PRINTER,n);
```

PRINTER agora é um RSemaphore. A solução é bem mais simples, elegante e à prova de erros, além de aumentar

a eficiência do sistema com a eliminação do semáforo MUTEX.

A implementação de RSemaphore em ShareMem é feita através de um tipo Set do Pascal, de um tipo Semaphore e de dois campos de controle do conjunto. Como um Set em Turbo Pascal é limitado a no máximo 256 elementos, temos a mesma restrição quanto ao número máximo de elementos manipulados por um RSemaphore. Os elementos do conjunto (os recursos) são numerados de 0 a 255, o que coincide com o tipo Byte do Turbo Pascal. Por esta razão, os parâmetros que recebem e retornam a identidade de um recurso em RP e RV são deste tipo. Na iniciação de um RSemaphore com InitRSemaphore não podemos utilizar o tipo Byte, pois o valor 256 (que é o máximo) está fora de sua faixa. Desta maneira o número inicial de recursos é do tipo Word.

Os dois campos de controle mencionados anteriormente servem para guardar o número total de recursos definidos na iniciação e para alternar de maneira circular o próximo recurso a ser alocado. Keedy et alii argumentam que se alocássemos sempre o primeiro recurso disponível no conjunto, no caso de um periférico como uma impressora teríamos um desgaste mecânico muito mais rápido nos primeiros recursos do que nos últimos. Por esta razão, este campo de controle faz com que sucessivas chamadas de RP retornem os recursos 0, 1, 2 e assim por diante, mesmo que um deles tenha sido liberado através de um RV.

IV.6 Variáveis Compartilhadas (Shared)

O conceito de variável compartilhada foi introduzido por BRINCH HANSEN (1972), em que a uma declaração usual de variável se adicionava o atributo "Shared" e esta só poderia ser utilizada dentro de um comando estruturado "Region" (similar a um "With" do Pascal). Como implementá-la desta maneira só pode ser feita com um modificação do compilador, a solução em ShareMem foi adaptá-la para o mecanismo de atualização de um dado

compartilhado sugerido em STONE (1987): no início da região crítica executa-se uma operação "Lock" sobre a variável compartilhada, e ao final uma operação "Unlock".

Para implementar esta estrutura de forma elegante, foi escolhido o uso de Programação Orientada a Objeto (BORLAND, 1989a), disponível a partir da versão 5.5 do Turbo Pascal (com isto este mecanismo não está definido se for usado o Turbo Pascal 5.0, apesar do Kernel rodar nesta versão). Resumidamente, um objeto é um record do Pascal que contém, além dos campos de dados, os procedimentos que operam sobre ele. O uso de um procedimento do objeto ("método", no jargão de OOP - Object Oriented Programming) já implica a variável sobre qual ele opera, não sendo necessário passá-la como parâmetro. As definições e o exemplo a seguir esclarecerão mais o uso dessa estrutura.

Tipo em ShareMem: Shared
Operações:

Procedure Init;
Procedure Lock;
Procedure Unlock;

Podemos usar esta estrutura de duas formas distintas para a proteção de regiões críticas. Na primeira delas, podemos declarar diretamente uma variável deste tipo, iniciá-la uma única vez no programa com um *Init* e garantir uma exclusão mútua através de um *Lock* e um *Unlock* nesta variável. O seguinte trecho de programa exemplifica isto:

```
Var Mutex : Shared;

   :
  Mutex.Init;
   :
```

```
Mutex.Lock;

{ Acessa a região crítica }
Mutex.Unlock;
:
```

Repare na elegância de uso de um objeto: seu procedimento (método) é chamado como se fosse um componente de um record, o que ele é mesmo, na verdade. Neste estilo de uso, uma variável Shared se assemelha muito a um semáforo binário, o que não mostra nenhuma vantagem do emprego de um objeto na sua implementação.

A segunda forma de utilização reflete mais a definição original de Brinch Hansen, e que só pode ser conseguida através de um objeto. A uma variável de qualquer tipo, seja simples ou estruturada, adicionamos uma do tipo Shared, mas *implicitamente* na sua declaração (adicioná-la diretamente como uma das componentes de um record seria uma solução sem o uso de objetos, mas deselegante). Isto é possível pela característica dos objetos poderem ser herdados, isto é, ao serem declarados poderem assumir implicitamente os campos e métodos de seu "pai" como também seus. O trecho seguinte ilustra este método:

```
Var RegCritica : Object(Shared)
                    Dado : Integer;
                 end;
  :
  RegCritica.Init;
  :
  RegCritica.Lock;
  RegCritica.Dado:=RegCritica.Dado + 2;
  RegCritica.Unlock;
  :
```

Neste exemplo temos a declaração da variável RegCritica, que seria apenas uma variável inteira chamada Dado. Como a manipulação desta variável precisava ser feita com exclusão mútua, RegCritica foi declarada como um objeto que herda o tipo Shared, e Dado passa a ser um campo deste novo objeto. A herança do tipo Shared permite à RegCritica utilizar normalmente os métodos Init, Lock e Unlock, como mostrado. Além disso, pela natureza de um objeto ser parecida com a de um tipo record, o comando With do Pascal poderia ser utilizado no exemplo acima, fazendo "With RegCritica do begin..." e chamando Init, Lock, Unlock e manipulando Dado sem mencionar seu nome antes.

A implementação dos métodos Lock e Unlock é feita em ShareMem por um semáforo binário do tipo BinSemaphore.

IV.7 Signalling Regions

Esta nova e recente estrutura, criada por REYNOLDS (1990), vem a resolver definitivamente os maiores problemas de uma das estruturas mais famosas e discutidas em sistemas de memória compartilhada: os monitores. Criados por BRINCH HANSEN (1973) e HOARE (1974), os monitores foram idealizados para acabar com a desestruturação e a desorganização promovida pelo uso de semáforos em algumas situações, pois operações P e V não emparelhadas ou a inversão de seus usos causam enormes problemas de programação.

Um monitor é tal qual um objeto (ver a seção anterior) onde dados e procedimentos são encapsulados, mas com um acréscimo: todos os procedimentos em um monitor possuem exclusão mútua automática, isto é, como se usássemos operações P e V na entrada e na saída de cada um deles. Além disso, tal qual uma região crítica evoluiu para uma região crítica condicional (BRINCH HANSEN, 1972), os monitores também incluem o mecanismo das variáveis de condição. Através dessas variáveis os processos podem

entrar no monitor, ser suspensos até que uma condição seja satisfeita e retornar a executar dentro dele com a garantia de exclusão mútua. Maiores detalhes sobre monitores podem ser encontrados em BOWEN (1980), SEGRE (1981a), ANDREWS (1983) e TERRY (1986).

Uma variável do tipo "Condition" tem duas operações associadas, "Wait" e "Signal". A primeira suspende o processo que a invocou numa fila própria desta condição, e a segunda faz continuar um e no máximo um processo suspenso nesta fila. Caso não haja nenhum suspenso, o Signal é simplesmente ignorado. É importante notar a diferença dessas operações em relação a P e V: Wait sempre bloqueia o processo que a chamou, P nem sempre; Signal pode não ter nenhum efeito, V sempre tem.

Entretanto, a idéia de se prover automaticamente exclusão mútua dentro de um monitor, inicialmente considerada uma grande vantagem, foi descoberta sofrer alguns sérios problemas. Podemos citar o discutido problema das chamadas aninhadas de monitores (ANDREWS, 1983), em que um processo P que chama um monitor M1 e dentro dele chama o monitor M2 pode vir a provocar um "deadlock", se P entrar em uma fila de condição em M2 (pois o monitor M1 fica "fechado" por P, que não abandona a exclusão mútua em M1 e a condição esperada em M2 só pode ser satisfeita por um outro processo que precisa entrar em M1).

Outros problemas seriam a exclusão mútua colocada desnecessariamente em alguns procedimentos do monitor, causando ineficiência, e a política de ordem de escalonamento após um Signal. Esta última tinha duas variantes:

a) O processo P que faz o Signal ainda continua executando, deixando um possível processo Q acordado esperando, até que P saia do monitor ou espere por uma condição ("signal-and-continue");

b) O processo P que faz o Signal cede o processador para o processo recém-acordado Q, esperando até que Q saia do monitor ou espere por uma condição ("signal-and-return").

A opção (a) era a preferida por Hoare, enquanto a opção (b) era a preferida por Brinch Hansen e Wirth. Em qualquer uma delas, como explica REYNOLDS (1990) em seu trabalho, o Signal ou corrompe a política de prioridades do despachante ou possibilita um bloqueio perpétuo. Tendo em vista todos esses problemas, sem solução por volta de 15 anos, Reynolds sugeriu recentemente uma nova construção, baseada na idéia dos monitores mas sem os problemas desses: as "Signalling Regions" [ver correção ao artigo em HANSON (1990)].

O autor argumenta em seu artigo que foi um erro sobrecarregar o conceito de monitor com encapsulamento e exclusão mútua. Seria melhor tê-los separados em duas construções, o módulo e o comando "Region". Isto resolve o problema das chamadas aninhadas de monitores, pois o programador é quem definiria em que procedimentos colocar e quando liberar a exclusão mútua. Tudo isso, ainda assim, usando a filosofia da programação modular. Esta política se justifica porque na maioria das vezes se quer encapsulamento aninhado sem exclusão mútua aninhada.

Para o problema do Signal, discutido anteriormente, há a proposta do "signal-and-continue-but-return". Esta nova semântica faz com que o processo que chama o Signal abandone a região crítica mas não largue a exclusão mútua, continuando a executar outros comandos fora da região. Ele ainda sinaliza um outro processo que porventura espera por aquela condição e o coloca na fila de prontos segundo a sua prioridade. Eventualmente este processo que foi acordado irá rodar, *de acordo com a polÀtica do despachante*, e supondo (corretamente) que as

mesmas condições da região crítica após o Signal ainda estão valendo. Se ninguém esperava o Signal, o processo que o chamou simplesmente libera a exclusão mútua.

Esta semântica, apesar de restringir o uso do Signal a uma única vez e ao fim da região (compromisso já assumido por Brinch Hansen), não corrompe a política de prioridades do despachante. O processo acordado por um Signal retorna à fila de prontos e aguarda a sua vez de rodar, de acordo com a sua prioridade. Ainda assim, ele tem a garantia de que a condição que ele esperava para prosseguir ainda vale, pois nenhum outro processo teve permissão para entrar na região crítica neste ínterim, nem mesmo o processo que executou o Signal (pelo fato de que um Signal, se utilizado, deve ser a última instrução em uma Signalling Region).

Uma Signalling Region é implementada em ShareMem através dos dois seguintes tipos:

Tipo em ShareMem: *SignalRegion*
Operações:

```
Procedure NewRegion;
Procedure Region;
Procedure EndRegion;
```

Tipo em ShareMem: *Condition*
Operações:

```
Procedure NewCondition(Var SR : SignalRegion);
Procedure Wait;
Procedure WaitP(Prio : Word);
Procedure Signal;
```

O tipo *SignalRegion* é um objeto de base, servindo apenas para ser herdado por uma variável do tipo Object que definirá a região crítica. A operação *NewRegion* apenas

inicia uma variável do tipo SignalRegion. *Region* e *EndRegion* delimitam o acesso à região crítica, tal qual P e V ou Lock/Unlock.

O tipo *Condition* também é um objeto, mas que só deve ser utilizado como um campo do objeto que herda SignalRegion. Podem ser declarados vários campos deste tipo, um para cada fila de condição existente na Signalling Region. Antes de utilizar uma variável deste tipo, a operação *NewCondition* deve ser executada para iniciar corretamente a fila de condição. Em geral esta operação deve ser feita logo após a iniciação da Signalling Region com NewRegion. Apesar da variável Condition declarada estar logicamente ligada a uma Signalling Region específica do programa, esta deve ser passada como parâmetro na operação NewCondition. Este fato se deve a esses dois tipos serem objetos diferentes em ShareMem, não estando diretamente interligados.

As operações *Wait* e *WaitP* (versão de Wait com prioridade, como nos semáforos discutidos na seção IV.2) fazem o processo ficar suspenso na fila de condição ligada a uma variável Condition. A operação *Signal* libera um processo desta fila de condição, se algum existir, com a semântica descrita anteriormente.

Dada a maior complexidade desta em relação às outras estruturas e a sua novidade, apesar de bem semelhante ao difundido conceito de monitor, é preferível demonstrá-la com um exemplo maior, que se encontra no Anexo VIII. O referido anexo é um programa que implementa um produtor e um consumidor dividindo uma área comum protegida por uma Signalling Region.

As únicas diferenças da implementação de uma Signalling Region em ShareMem e o especificado por Reynolds são:

a) A ausência de uma checagem do uso de Wait em variáveis do tipo Condition apenas no interior de uma Region;

b) Alteração da semântica do End Region e do Signal. Reynolds estabelece que o uso de um Signal deve ser o último comando de uma Region; em ShareMem esta checagem não é realizada, e neste caso o Signal substitui o EndRegion, que não deve ser utilizado (ver o exemplo no Anexo VIII);

c) A utilização condicional de um Signal deve ser feita normalmente através de um comando IF ou CASE do Pascal.

IV.8 Mecanismos de Exclusão Mútua e Sincronização do Módulo Kernel

Apesar deste capítulo tratar sobre o módulo ShareMem, vale mencionar que o módulo Kernel também dispõe de construções de exclusão mútua e sincronização. O par de procedimentos Lock e Unlock e o procedimento Barrier, já apresentados no Capítulo III, são os dois mecanismos existentes.

Lock/Unlock provêem a maneira mais primitiva, fora a inibição de interrupções, de se garantir exclusão mútua no ambiente Kernel. Lock desabilita o despachante, e Unlock o habilita. Já o procedimento Barrier, utilizado da forma Barrier(n), faz que todos os processos que o chamem só prossigam após a passagem do enésimo. Ele é, dessa maneira, um bom instrumento para uma sincronização simples, sem similar direto em ShareMem.

Quanto ao uso de Lock/Unlock deve ser feita uma ressalva. Apesar de simples de usar, eles não podem ser considerados totalmente seguros. Como os procedimentos do próprio Kernel, incluídos aí todos os módulos, os utilizam,

o usuário deve atentar para o conteúdo da região crítica protegida. Se esta invocar alguma rotina desses módulos, a exclusão mútua estará suspensa no seu retorno. Dessa forma o uso desses dois procedimentos deve ser deixado para trechos simples e pequenos, dir-se-ia até em casos extremos, evitando não só erros de programação como uma diminuição da concorrência entre os processos, visto que eles habilitam e desabilitam o próprio despachante. É recomendável nos outros casos a escolha de uma das várias estruturas de ShareMem.

IV.9 Outras Considerações

Para terminar este capítulo, seria interessante citar algumas estruturas não implementadas em ShareMem mas fartamente encontradas na literatura. O critério de escolha das construções deste módulo foi o das mais usadas e mais poderosas, além de se evitar mecanismos em que a dificuldade de implementação não valeria a pena pelo seu uso.

Temos entre os mecanismos pesquisados mas não incluídos o tipo *Event* e suas operações Await e Cause (BRINCH HANSEN, 1972); *Queues* e suas operações Delay e Continue, de Concurrent Pascal (BRINCH HANSEN, 1975); o comando *When* da linguagem Edison (BRINCH HANSEN, 1981); *Path Expressions* (KOLSTAD, 1980); *Mediator* e *Gladiator* (BOWEN, 1980) e *Sem foros de Prioridade* (FREISLEBEN, 1989). As referências citadas servem como base para consulta visando uma possível extensão deste módulo.

Também foram analisadas algumas idéias embutidas em certos artigos. Por exemplo, KEEDY (1979) sugere a implementação de P e V através de macros e não por chamada de procedimentos, visando reduzir um possível "overhead" de entrar e sair de uma rotina de um dado núcleo. Esta questão foi pesada e no caso do Kernel a idéia foi considerada sem tanto valor, pois o ganho em velocidade traria como prejuízo uma implementação mais complicada e obscura. De

qualquer forma, é bastante interessante, e também pode ser encontrada em KEEDY (1985).

Em regiões críticas curtas, WETTSTEIN (1977) sugere o uso de inibição das interrupções (no caso de CPUs 80x86, CLI para desabilitar e STI para habilitar). Este método deve ser usado com muito cuidado para não haver perda de interrupções de hardware como as do relógio, teclado e disco, e portanto deve ser evitado. É preferível deixar este tipo de procedimento apenas para programação de rotinas que fazem atendimento de interrupções e as de baixo-nível em geral.

A associação de um timeout a semáforos e variáveis de condição pode ser realizada diretamente pelo procedimento SetTimeout do Kernel, como visto no Capítulo III. Este procedimento, aliás, é válido para qualquer estrutura que inclua uma fila de espera. Autores como SEARS (1985) e BEMMERL (1987) incluíram este recurso em seus trabalhos.

Por último, vale observar quanto a dois procedimentos do próprio Turbo Pascal, INC e DEC. Eles servem para incrementar e decrementar variáveis de qualquer quantidade, tendo como default o valor 1 (BORLAND, 1989b). Eles podem, portanto, ser utilizados no lugar de uma operação de soma ou subtração. A importância desses dois procedimentos no presente contexto é a sua implementação, que gera um código de máquina de uma instrução ininterrupta se a variável ocupar 1 ou 2 bytes (tipos Byte e Integer, por exemplo). A vantagem disto é que se a variável for compartilhada a exclusão mútua já está assegurada, dispensando quaisquer mecanismos.

Ao contrário dos módulos Kernel e MsgPass, o módulo ShareMem não emite nenhuma mensagem de erro. Isto porque não há nenhuma estrutura que possa ter seu uso verificado a tempo de execução, mas apenas a nível sintático, durante a compilação.

CAPITULO V: O MÓDULO DE TROCA DE MENSAGENS

Neste capítulo apresentaremos o módulo *MsgPass*, responsável pela estrutura e primitivas para troca de mensagens entre processos. Ao contrário do módulo ShareMem, em que várias estruturas distintas estão presentes, este módulo implementa apenas uma construção para troca de mensagens. Contudo, seguindo a proposta desta Tese, existem primitivas e modos de uso os mais diversos, de maneira que boa parte dos mecanismos encontrados na literatura podem ser cobertos.

Os surveys de ANDREWS (1983) e BAL (1989) são muito boas referências, principalmente o último, por tratar especificamente de sistemas distribuídos. O trabalho de BITAR (1988), Tese da COPPE/UFRJ, também apresenta um módulo de troca de mensagens, mas com características bastante distintas do aqui implementado.

As seções que seguem apresentarão o módulo MsgPass. A seção V.1 discutirá questões relevantes a sistemas de trocas de mensagens e as decisões de implementação para o referido módulo. As demais seções detalham suas primitivas e outras considerações importantes.

V.1 Conceitos de Trocas de Mensagens

A comunicação entre processos através da troca de mensagens acontece quando um deles transmite alguma informação e um outro a recebe, sendo esta interação realizada por meio de um canal de comunicação. Desta forma, os processos não necessitam compartilhar memória, uma vez que é o próprio canal que se encarrega de entregar a informação transmitida ao seu destino.

Basicamente, o que define um dado sistema de troca de mensagens é a estrutura deste canal e as

primitivas de comunicação existentes nele. O termo canal é genérico, e serve para designar aqui qualquer mecanismo de comunicação entre processos. Dependendo das decisões de implementação tomadas este canal pode assumir várias formas, e as linguagens que dão suporte a este tipo de comunicação entre processos dão nomes diferentes a ele. Em MsgPass este canal se chama *Channel*, e é o tipo de dados que deve ser declarado sobre o qual as primitivas de comunicação vão operar.

A implementação do tipo Channel decorre das questões discutidas nas próximas seções.

V.1.1 Chamada Direta e Indireta

Uma comunicação pode ser realizada entre dois processos de forma direta ou indireta. Na primeira designa-se tanto na transmissão como na recepção o identificador do processo com que se deseja comunicar, que atua como o próprio canal de comunicação. Na segunda o canal existe como uma entidade própria, de forma que os processos não se comunicam diretamente entre si, mas através dele.

A forma direta de comunicação é muito limitante, pois impõe o nome dos processos como parte das primitivas de comunicação. É utilizada na proposta da lingugem CSP, de HOARE (1978). A forma indireta é mais modular e por isso mais popular, sendo definida em diversos trabalhos por estruturas como o tipo Channel de Occam-2 (INMOS, 1984), Mailbox de HOPPE (1980), Port de SILBERSCHATZ (1981), etc.

Naturalmente, por existir o tipo Channel em MsgPass, a forma indireta é a adotada neste módulo. O tipo Channel teve como fundamento conceitual o tipo Mailbox de HOPPE (1980), mas possui muito mais primitivas e versatilidade. A partir de agora o termo *Canal* se refere a um meio de comunicação genérico, enquanto *Channel* é a implementação do módulo MsgPass.

V.1.2 Criação, Utilização e Propriedade de um Canal

Como uma variável do tipo Channel é a forma de comunicação por troca de mensagens no Kernel, ela deve ser declarada de modo que seu escopo seja visível por todos os processos envolvidos. Além dessa declaração estática, todo Channel deve ser criado dinamicamente por um e somente um processo, para poder ser efetivamente utilizado. Esta criação inicia vários campos de controle de sua estrutura, e principalmente define seus parâmetros. O tipo Channel é bastante versátil, e dependendo desses parâmetros de criação vários mecanismos encontrados na literatura, definidos de forma rígida e sem variações, podem ser obtidos.

Para utilizar um Channel todos os processos devem ser identificados como usuários dele, em apenas uma das duas modalidades: Sender ou Receiver. Esta restrição é facilmente contornável declarando-se outro Channel com a modalidade de comunicação inversa para comunicações no sentido contrário. A finalidade da identificação de usuário do Channel é para checagem de uso das primitivas, bem como controlar se o número de usuários nas duas modalidades está dentro do especificado.

O processo que cria o Channel é seu proprietário. Isto não lhe dá nenhum privilégio de utilização, uma vez que ele nem precisa ser usuário do Channel. Sua única função é garantir que ele e somente ele pode desativar o Channel, evitando erros de programação. Em outros sistemas a propriedade de um canal está ligada ao fato do dono ser o único a receber mensagens por ele.

V.1.3 Capacidade de um Canal

Todo canal de comunicação possui uma dada capacidade de armazenamento de informações, ou seja, quantas mensagens podem ficar pendentes de entrega dentro

do canal. Existem três tipos de capacidade: zero, limitada e infinita.

Um canal de capacidade zero simplesmente não pode armazenar nenhuma mensagem dentro dele, e assim o processo que envia uma mensagem por ele deve esperar até que um outro a receba para continuar sua execução. Os dois processos, portanto, devem estar sincronizados para realizar esta transferência. Essa sincronização é conhecida como "rendezvous".

Um canal de capacidade limitada dispõe de um espaço determinado na sua fila de mensagens pendentes. Enquanto a fila não estiver cheia, os processos que enviam por esse canal podem prosseguir como se a mensagem já tivesse sido entregue ao seu destino. Se a fila estiver cheia, o canal se comporta exatamente como se tivesse capacidade zero.

Por último, um canal de capacidade infinita nunca suspende a execução de um processo transmissor, uma vez que a sua fila de mensagens tem espaço (teoricamente) para infinitas mensagens. Na prática, em geral, o "infinito" equivale à memória disponível do sistema.

O tipo Channel pode ser configurado para operar com qualquer um desses três tipos, através da sua primitiva de criação. A capacidade limitada é implementada alocando-se buffers para mensagens dinamicamente na medida do necessário, até atingir o valor limite. A capacidade infinita é feita da mesma forma, só que o limite é um valor tão alto que a alocação de buffers é feita até o esgotamento da "heap", local da memória administrada pelo Turbo Pascal onde são criadas as variáveis dinâmicas.

V.1.4 Envio de Mensagens por Cópia e por Referência

Esta questão é relativa a uma transmissão ter sua informação copiada realmente ou apenas ter um ponteiro para ela passado, até que seja recebida por um processo. Isto é o mesmo mecanismo de passagem de parâmetros por valor ou por referência.

Se a informação for copiada, em casos em que o tamanho da mensagem é maior que o de um ponteiro para ela, temos perda de velocidade. Em compensação, se passarmos somente a referência, corremos o risco de ter esta posição de memória alterada antes de sua devida recepção. Valorizando mais a questão da segurança, a opção em MsgPass foi implementar a transmissão com a cópia das mensagens.

V.1.5 Tamanhos das Mensagens

As mensagens transmitidas por um processo podem ser de tamanho fixo, variável ou tipadas. Tamanhos variáveis são mais abrangentes mas tremendamente mais complicados de implementar. Mensagens tipadas, em que o tamanho é definido automaticamente pelo tipo de dado associado a um canal, só podem ser conseguidas por uma mudança no compilador. Desta forma, Channel só manipula mensagens de tamanho fixo, tamanho este definido quando da sua criação.

Para superar esta limitação, pode ser utilizado o artifício dos records variantes encontrados no Pascal. O tamanho das mensagens seria o do total do record, mas um processo ora poderia utilizar uma parte do record, ora outra.

V.1.6 Canais Unidirecionais e Bidirecionais

Outro aspecto relevante ao funcionamento de um canal é se o sentido das informações que fluem por ele se

dá unidirecional ou bidirecionalmente. Quer dizer, se em um canal só é possível transmitir em um extremo e receber do outro ou se as duas operações são válidas nos dois extremos.

Dada a natureza do Channel ser tal qual uma caixa postal, conceito extraído da Mailbox de HOPPE (1980), ele é fundamentalmente um canal unidirecional, uma vez que cada processo usuário de um Channel só poder transmitir OU receber por ele.

V.1.7 Troca de Mensagens Bloqueante e Não-bloqueante

Uma troca de mensagens bloqueante impõe ao processo que realiza uma operação de transmissão (recepção) uma espera, de forma que ele só prossiga sua execução após um outro processo ter-se sincronizado com ele para receber (transmitir) a mensagem em jogo. Já em uma troca não-bloqueante o processo que transmite (recebe) não espera para saber o resultado da operação, isto é, quem transmite pressupõe que a mensagem chegou ao destino e quem recebe pressupõe que alguma mensagem foi recebida.

Quanto à operação de recepção, é comum fazê-la bloqueante, pois não é possível ter sempre certeza que alguma mensagem será recebida. Mas em uma operação de transmissão as duas formas são viáveis, e a escolha entre as duas vem gerando controvérsia há bastante tempo. Em artigo recente, GEHANI (1990) discute justamente este problema, e conclui que a melhor solução para uma linguagem é incluir as duas formas. Em MsgPass, esta foi a política.

A decisão entre as primitivas de transmissão serem ou não não-bloqueantes em MsgPass é realizada individualmente para cada variável Channel. Na sua criação, a capacidade do buffer é quem determina a política: se for zero, a operação é bloqueante; capacidades limitada ou infinita, operação não-bloqueante.

V.1.8 Tipos de Transmissão em um Canal

As transmissões feitas em um canal podem ser feitas tendo como destino um só processo (1 para 1), vários processos (1 para vários) ou todos os processos receptores (1 para todos). A primeira operação é um "Send" usual, e as duas últimas se chamam "Multicast" e "Broadcast" (BAL, 1989). Multicasts podem ser encontrados em LABRECHE (1990), e são uma forma bastante cômoda e eficiente de se fazer um "Send" múltiplo. Broadcasts foram implementados por LEBLANC (1984) e GEHANI (1984) em seus trabalhos, tendo este último dedicado uma linguagem exclusivamente para comunicações deste tipo (BSP). Gehani cita neste artigo algumas vantagens da programação com broadcast: não ser necessário conhecimento explícito de quem são os processos receptores; inclusão e exclusão de processos dinamicamente, sem alterar a estrutura do programa; monitoração de atividades; propriedade do mecanismo para o uso eficiente de uma arquitetura BPM (Broadcast Protocol Multiprocessor), como uma rede Ethernet; etc.

O tipo Channel dá suporte a todas essas três formas de transmissão, que serão mais bem explicadas em seções posteriores.

V.1.9 Tipos de Recepção em um Canal

As recepções em um canal podem ou não especificar um determinado processo de quem exclusivamente podem receber. Em Channel, pela sua estrutura de caixa postal, isto não foi feito, deixando as primitivas de recepção receberem mensagens de qualquer processo indistintamente. Esta opção é bastante usual, e pode ser encontrada, por exemplo, em BAGRODIA (1985).

V.1.10 Mensagens Com e Sem Prioridade

As mensagens transmitidas em um canal podem ser normais ou podem ter uma prioridade associada a elas. Esta última forma é bastante útil para o envio de mensagens de pânico ou simplesmente para facilitar determinados algoritmos, como o "Scan" utilizado para escalonamento de disco (PETERSON, 1985, p. 263).

O tipo Channel permite a transmissão de mensagens com prioridades através de qualquer uma das três formas apresentadas anteriormente (Send, Multicast e Broadcast).

V.1.11 Comandos Guardados

Os comandos guardados concebidos por DIJKSTRA (1975) são expressões booleanas e operações de transmissão/recepção que controlam a execução de comandos alternativos. Foram utilizados na definição da linguagem CSP (HOARE, 1978), mas excluíndo a operação de transmissão em um guarda. Extensões de CSP e a linguagem Joyce (BRINCH HANSEN, 1987a) incorporaram também a operação de transmissão em um comando guardado.

Em MsgPass os comandos guardados são modelados considerando todo o guarda como uma expressão booleana. Para isto, existem operações de transmissão e recepção próprias para determinar se é possível enviar ou receber uma mensagem sem ficar bloqueado. Essas operações são implementadas na forma de uma função que retorna um valor booleano, e são similares às funções GetReady e PutReady encontradas em PERIHELION (1989).

Mais sobre comandos guardados na seção específica sobre estas primitivas.

V.1.12 Timeouts nas Primitivas de Comunicação

Esta é uma característica encontrada em alguns sistemas de trocas de mensagens, utilizada basicamente nas operações bloqueantes. Incluindo um timeout em uma transmissão ou recepção bloqueante, podemos evitar bloqueios perpétuos devido a erros de programação ou utilizar as primitivas em aplicações de tempo real. Também é uma maneira simples de transformar um mecanismo bloqueante em não-bloqueante, através da especificação de um timeout igual a zero.

O tipo Channel não precisa de primitivas especiais para especificação de timeouts, como faz SEARS (1985). O módulo Kernel fornece o procedimento SetTimeout para quaisquer situações em que um processo entre em uma fila de espera, sendo de aplicação mais ampla.

V.1.13 Confiabilidade das Primitivas de Comunicação

Como discutido em BAL (1989), um sistema de troca de mensagens pode ter primitivas confiáveis ou não-confiáveis, isto é, que garantam ou não que uma mensagem foi entregue e recebida corretamente. Uma vez que esse problema só vem à tona em um sistema distribuído real, não caberia no caso de MsgPass tal discussão. Mesmo assim, como são propostos em uma extensão futura deste trabalho a incorporação de um módulo para simulação distribuída e também um subsistema de comunicação entre computadores, pode-se pensar na inclusão de primitivas não-confiáveis para a modelagem de algumas aplicações que as necessitem.

V.2 O Tipo Channel

Nas seções que seguem apresentaremos a estrutura interna de um tipo Channel e suas primitivas. A interface completa para o programador se encontra no Anexo IV.

V.2.1 A Estrutura

Temos as seguintes informações dentro de uma variável do tipo Channel:

* **O proprietário do canal**, representado pelo seu PID;

* **O número máximo de Senders e Receivers**, conforme a criação do Channel. Este número está limitado a 65535;

* **O número atual de Senders e Receivers** usuários do Channel;

* **Uma lista de identificação dos Senders** usuários do Channel, com seus PIDs;

* **Uma lista de identificação dos Receivers** usuários do Channel, com seus PIDs e número atual do Broadcast que está para receber;

* **Uma lista de Senders e outra de Receivers pendentes;**

* **O número do Broadcast atual** realizado no Channel. São permitidos no máximo 65534 broadcasts;

* **O número de mensagens que deram entrada no Channel**, para controle meramente estatístico;

* **O tamanho das mensagens transferidas;**

* **O número máximo de buffers alocados ao Channel e o número atual de buffers ocupados;**

* **Uma fila de armazenamento de mensagens.** Cada elemento desta fila tem as seguintes informações:

  - **Um "tag"** para distinguir mensagens normais de broadcasts e ainda identificar o número de um

broadcast, se for o caso. Uma mensagem de broadcast tem prioridade sobre mensagens normais, para maior eficiência do sistema, independentemente da prioridade explicitamente atribuída à mensagem;

- **A prioridade da mensagem.** O esquema é o mesmo adotado no módulo Kernel, ou seja, valores maiores indicam uma mensagem mais prioritária. A faixa de prioridades é a dos números inteiros, que vai de -32768 a +32767;

- **Número de receivers** para a mensagem. É utilizado na implementação de um multicast, evitando duplicações desnecessárias de uma mensagem. Ela só sai desta fila quando já tiver sido recebida por este número de receivers;

- **Um ponteiro para a mensagem** propriamente dita. Se a mensagem em si ocupa entre 1 e 4 bytes, ela é armazenada diretamente na posição deste ponteiro, evitando desperdício de memória (HOPPE, 1980). Se ocupa mais de 4 bytes, memória suficiente é alocada na heap para armazená-la;

- **Um ponteiro para a próxima mensagem** da fila.

V.2.2 As Primitivas de Manipulação

Constante: *Infinity*
Tipo: *UserType = (Sender,Receiver)*
Operações:

```
Procedure CreateChannel(Var C : Channel; MaxSenders,
                        MaxReceivers, MsgSize,
                        Buffers : Word);
Procedure DestroyChannel(Var C : Channel);
Procedure UseChannel(Var C : Channel; User : UserType);
Procedure FreeChannel(Var C : Channel);
```

Para utilizar um tipo Channel, um processo deve criá-lo através da primitiva *CreateChannel* e todos os usuários interessados devem chamar a primitiva *UseChannel*, no modo Sender ou Receiver. Se um processo usuário quiser deixar de sê-lo, ele pode fazer isso através da primitiva *FreeChannel*, conseqüentemente liberando uma vaga de usuário para um outro processo qualquer. Ao terminar completamente o uso de um Channel, o dono do canal (quem chamou CreateChannel) pode, se quiser, liberar a memória dinâmica utilizada por uma variável deste tipo, com a operação *DestroyChannel*. Esta operação é opcional, e se for feita com mensagens ainda em trânsito todas elas serão perdidas.

A primitiva CreateChannel merece maiores comentários sobre seus parâmetros. MaxSenders e MaxReceivers são, respectivamente, o número máximo de Senders e Receivers permitidos em um dado Channel. Sua função é estritamente checagem de uso de uma variável do tipo Channel. MsgSize é o tamanho das mensagens que fluirão por este Channel. É aconselhável o uso da função "Sizeof" do Turbo Pascal sobre o tipo de dados que representará a mensagem, pois ele retorna automaticamente o tamanho de uma variável e é menos suscetível a erros do que um valor constante diretamente inserido pelo programador. Por último, Buffers determina a capacidade do Channel, especificando o número máximo de buffers de mensagens que ele admite. Como discutido na seção V.1.3, o tipo Channel permite capacidade zero, passando este valor como o parâmetro Buffers; capacidade limitada, se for passado um valor maior que zero, e infinita se for utilizada a constante *Infinity* neste parâmetro.

V.2.3 As Primitivas de Transmissão

Variável: *Any*
Operações:

```
Procedure Send(Var C : Channel; Var Msg);
Procedure PriSend(Var C : Channel; Var Msg;
                  Prio : Integer);
Procedure Multicast(Var C : Channel; Var Msg;
                    Receivers : Word);
Procedure PriMulticast(Var C : Channel; Var Msg;
                       Receivers : Word; Prio : Integer);
Procedure Broadcast(Var C : Channel; Var Msg);
Procedure PriBroadcast(Var C : Channel; Var Msg;
                       Prio : Integer);
```

Como discutido na seção V.1.8, o tipo Channel implementa três tipos de transmissão: *Send, Multicast* e *Broadcast.* Também como apresentado na seção V.1.10, todas os três tipos possuem uma variante que inclui mensagens com prioridade *(PriSend, PriMulticast* e *PriBroadcast).* Para o uso de qualquer um deles é necessário que o processo que os chama seja usuário do Channel no modo Sender. Além disso, Msg é a mensagem que se quer transmitir, necessariamente uma variável. Msg utiliza a facilidade de variáveis "sem tipo" do Turbo Pascal, permitindo que qualquer tipo de variável sirva como parâmetro para uma mensagem. Logicamente, a mensagem transmitida deve ser uma variável com tamanho de acordo com o definido na criação do Channel, para evitar envio de "lixo" ou a mensagem ser truncada. A variável predefinida *Any* pode ser utilizada como parâmetro para Msg, que indica à primitiva para transmitir uma mensagem de tamanho zero (neste caso o tamanho de mensagens do Channel é desconsiderado). Esta característica existe em Occam-2 e serve como instrumento de sincronização (INMOS, 1984), mas deve ser utilizada com cuidado. A uma transmissão de um Any deve corresponder uma recepção de um Any também, ou pelo menos que a mensagem recebida seja

desconsiderada, pois *nada foi transmitido realmente*.

No caso de um Multicast, deve ser especificado o número de receivers para a mensagem sendo transmitida. Como já dito anteriormente, apenas uma cópia da mensagem é armazenada, tornando o sistema mais eficiente. O Broadcast também utiliza este mecanismo, mas considera como "default" o número total de receivers usuários do canal no momento da chamada da primitiva. Esta última observação é bastante importante, pois para um Broadcast funcionar corretamente é preciso que os processos interessados em recebê-lo já sejam usuários do Channel. Devido a condições de corrida isto pode não acontecer, tornando necessária a introdução de algum mecanismo de sincronização para tal.

A diferença mais marcante entre um Multicast e um Broadcast é que o Multicast entrega as múltiplas mensagens transmitidas para quaisquer processos prontos para recebê-las, mesmo para um dado processo mais de uma vez. A idéia por trás de um Multicast é simplesmente evitar repetições de um mesmo Send, economizando espaço em buffer e aumentando o desempenho do sistema. Pode-se imaginar como exemplo a impressão de várias cópias de um mesmo arquivo em um ambiente com três impressoras idênticas. O arquivo pode ter suas cópias impressas em cada uma delas, em duas ou em apenas uma impressora, dependendo de qual esteja disponível em cada momento.

O Broadcast, aparentemente um Multicast para todos os receivers do Channel, é bem mais complexo. As mensagens enviadas por broadcast devem ser entregues uma e somente uma vez para cada receiver usuário do Channel. Para isto ser viabilizado, o Channel controla o número atual do Broadcast irradiado por ele e cada receiver faz o mesmo, controlando até que número ele já recebeu. Uma mensagem só pode ser retirada por um processo se for uma mensagem normal ou de um broadcast ainda não recebido. Pela implementação, broadcasts mais antigos têm prioridade sobre

os mais recentes, que por sua vez têm prioridade sobre mensagens normais, tudo independentemente da prioridade explicitamente atribuída a cada mensagem. Por facilidade, mensagens normais são consideradas internamente como o último dos 65534 broadcasts permitidos. Este esquema permite uma detecção rápida se uma dada mensagem pode ser entregue a um receiver: se o "tag" dela for o de um broadcast ainda não recebido (incluindo-se assim mensagens normais), a transferência pode ser realizada.

V.2.4 As Primitivas de Recepção

Variável: *Any*
Operações:

Procedure Receive(Var C : Channel; Var Msg);
Procedure ReceiveSynch(Var C : Channel; Senders : Word);

A primitiva *Receive* é o par perfeito de Send. Desta forma, todas as explicações dadas na seção anterior sobre o parâmetro Msg e a constante predefinida *Any* também são válidas aqui. Vale ainda lembrar a discussão sobre troca de mensagens bloqueante e não-bloqueante da seção V.1.7. Para o uso desta primitiva em um destes modos deve-se definir a capacidade do Channel de acordo com o caso, como explica a referida seção.

O *ReceiveSynch* nada mais é que um Receive múltiplo de mensagens "Any" transmitidas (notar que a mensagem em si é descartada nesta primitiva), e substitui a codificação de "loops" apenas para receber o sincronismo de vários processos ao mesmo tempo.

Naturalmente, os processos que chamam qualquer uma dessas duas operações deve ser um usuário do Channel no modo Receiver.

V.2.5 As Primitivas de Comandos Guardados

Operações:

Function SendOk(Var C : Channel; Var Msg) : Boolean;
Function ReceiveOk(Var C : Channel; Var Msg) : Boolean;

Estas duas primitivas implementam os comandos guardados de Dijkstra discutidos na seção V.1.11. Com elas um guarda pode ser escrito integralmente como uma expressão booleana, uma vez que as duas retornam True ou False se a operação teve ou não êxito, respectivamente. Se um *SendOk* tem sucesso, ou seja, é possível ao processo enviar a mensagem sem ficar bloqueado, porque existe alguém para recebê-la ou o Channel tem um buffer vago para armazená-la, além de retornar True ele transfere a mensagem, se comportando tal qual um Send usual. Em caso contrário, ele simplesmente retorna False. Um *ReceiveOk* se comporta de maneira análoga, se também comparado ao Receive usual.

A vantagem destas duas primitivas é jamais bloquear o processo que as chama, condição básica para serem utilizadas em um comando guardado. Com isso, podem também ser usadas em casos isolados onde se deseja uma transmissão ou recepção apenas se existe algum outro processo pronto para completar a operação. Um exemplo deste uso é nos comandos Select e PriSelect do módulo Kernel. Com um Select e a primitiva ReceiveOk pode-se simular o funcionamento de um "Accept" da linguagem ADA (LEDGARD, 1983).

V.2.6 A Primitiva de Estatística

Operação:
Procedure ChannelStats(Var C : Channel; Var InMsgs,
OutMsgs : Longint);

A primitiva *ChannelStats* foi incluída tão somente para facilitar a avaliação de programas e algoritmos pelo

programador. Com ela podemos obter diretamente através de seus parâmetros de saída InMsgs e OutMsgs o número de mensagens que deram entrada e que já foram entregues em um Channel, respectivamente. Multiplicados pelo tamanho da mensagem obtemos o número de bytes que trafegaram pelo Channel até o momento. Deve-se notar que os valores informados correspondem a mensagens efetivamente veiculadas pelo Channel, quer dizer, apesar de na implementação de um broadcast ou de um multicast apenas uma única cópia ser considerada, para efeitos estatísticos todas elas são contabilizadas.

A vantagem de se ter esta primitiva é livrar o programador da incômoda tarefa de incluir em um programa variáveis para controle estatístico de um canal, tarefa muitas vezes necessária quando da avaliação de um sistema ou de um novo algoritmo, por exemplo.

V.2.7 A Lógica de Funcionamento de Senders e Receivers

A seguir descreveremos a lógica de funcionamento de Senders e Receivers, ou seja, como eles se comportam internamente face à estrutura de um Channel. Tal lógica é descrita em linguagem algorítmica.

Sender:

```
SE há um Receiver bloqueado esperando esta mensagem
     Copiar a mensagem para ele e acordá-lo
SENÃO
     SE há espaço em buffer
          Colocar a mensagem no buffer e sair
     SENÃO
          SE a operação é um SendOk
               Retornar da operação sem sucesso
          SENÃO
               Bloquear-se e esperar espaço em buffer
               OU um Receiver pronto para um rendezvous
```

Receiver:

SE há uma mensagem no buffer que se possa retirar
    Pegar a mensagem e sair
SENÃO
    SE a operação é um ReceiveOk
        Retornar da operação sem sucesso
    SENÃO
        Bloquear-se e esperar um Sender para um
          rendezvous

V.2.8 Mensagens de Erro do Módulo MsgPass

A utilização das primitivas do tipo Channel é rigorosamente testada, para maior segurança do usuário. Assim como no módulo Kernel, as mensagens de erro são exibidas no mesmo formato, apenas trocando "Kernel Error" para "MsgPass Error". Todas as outras observações já discutidas na seção II.14 continuam válidas aqui.

O Anexo VII contém todas as mensagens de erro exibidas por MsgPass, com sua devida explicação.

V.3 Considerações Finais e Extensões Futuras

Durante o levantamento bibliográfico desta Tese, ficou patente o volume de artigos publicados recentemente sobre sistemas de troca de mensagens. Isto se deve à crescente difusão dos sistemas distribuídos nas suas mais variadas formas e aplicações, e pelo fato de trocas de mensagens serem um modelo mais do que natural para sistemas deste tipo. Além disso, muitos autores sustentam que este é o melhor modelo em qualquer sistema, seja ele de memória centralizada ou distribuída. A discussão ainda está em aberto e há muito o que ser feito de pesquisa na área.

Alguns mecanismos pesquisados e não implementados neste trabalho são as RPCs (Remote Procedure Calls),

encontradas, por exemplo, em ANDREWS (1983) e CORMACK (1988), que visam modelar em mais alto nível a transmissão e recepção de mensagens; protocolos descritos como linguagens livres-de-contexto (FISHER, 1988) especificando tipo, ordem e número de ocorrências (esta última característica não existe em Occam-2), reduzindo o risco de deadlocks, provendo auto-documentação e diminuindo a taxa de erros de programação; primitivas não-confiáveis (BAL, 1989; CERRADA, 1988; RUGGIERO, 1982) para certas modelagens; Remote Memory Reference e Remote Variable Reference (LEBLANC, 1984), que seriam acessos em alto nível a posições de memória ou variáveis em outro processador, internamente manipulados com trocas de mensagens; história de um canal, incluindo quem fez uma operação sobre ele, qual operação e quando (GOOD, 1979); etc.

Há por sua vez mecanismos encontradas na literatura que não possuem equivalente direto no ambiente proposto por esta Tese, mas que podem ser modelados de alguma maneira. É o caso, por exemplo, da primitiva SendR de BEMMERL (1987), que é um Send com retransmissão automática após um timeout e que pode ser obtido por um simples loop; os Send's concorrentes bloqueantes encontrados em QUEIROZ (1985), que podem ser obtidos por Send's dentro de um Cobegin/Also/Coend sobre um Channel de capacidade zero; as opções Suchthat e By do Accept da linguagem Concurrent C (GEHANI, 1986), que podem ser obtidos através de mensagens com prioridade em um Channel e da própria sintaxe do Pascal.

Por fim, vale abordar o tema da implementação de um sistema real de troca de mensagens. O ambiente Kernel foi construído para rodar em um microcomputador apenas, mas uma futura extensão pode alterá-lo, sem grandes dificuldades, para rodar em um sistema distribuído. A base desta mudança é o módulo MsgPass, onde as primitivas podem ser facilmente adaptadas para tal se for incorporado um sub-sistema de comunicação que cuide da transmissão efetiva

de uma mensagem entre os computadores. Uma implementação plausível é utilizar os recursos de uma Rede Novell para micros PC-compatíveis.

Quanto à questão da chamada "terminação distribuída", discutida, por exemplo, em BAGRODIA (1985), ela só se faz presente em um sistema distribuído de fato. Neste ambiente, como tal sistema é simulado, não temos os problemas inerentes àquela questão. Naturalmente, se um sistema real de troca de mensagens for implementado, tal problema deve ser revisto.

CAPITULO VI: O MÓDULO DEBUGGER

Neste capítulo será apresentado um depurador específico para o Kernel, doravante chamado de *Debugger*. Será abordada a sua utilização e os problemas inerentes à depuração de programas concorrentes.

VI.1 A Depuração de Programas Concorrentes

A depuração de programas concorrentes é muito mais difícil do que a de programas seqüenciais. Como repetir a mesma execução de um programa duas vezes para localizar um erro, se a ordem em que as instruções são executadas é desconhecida? E se formos executando um programa passo a passo, como evitar a interferência deste tempo de interação do usuário com a ordem de execução do programa? Muitos programas (incorretos) podem funcionar ou deixar de funcionar pela simples inserção de "delays" dentro do programa ou por interação com o usuário, como comprova o estudo de GAIT (1985, 1986) sobre o que ele denominou de "Probe Effect". E em sistemas distribuídos, onde não temos a orientação de um relógio global (LAMPORT, 1978) e para um "breakpoint" funcionar corretamente precisamos que todos os processos em todos os processadores sejam suspensos instantaneamente, ou pelo menos num estado coerente (CHANDY, 1985)? Estes são apenas alguns dos muitos problemas que a Programação Concorrente traz para a depuração de um programa.

O Debugger do Kernel, apesar de simples, é uma ferramenta muito eficaz na localização de diversos tipos de erros. Basta dizer que ele próprio foi usado para depurar os outros módulos, pois são muito freqüentes os erros de uso das estruturas de concorrência. Naturalmente, como está ele é ainda apenas o início de um estudo sobre depuradores de programas concorrentes, pois há tanto o que se fazer que isto seria facilmente uma tese em separado. Ao final deste capítulo são sugeridas implementações para

versões futuras.

VI.2 O Debugger e o Turbo Pascal

Uma das vantagens de se estender uma linguagem para suporte de concorrência através de bibliotecas de rotinas é o aproveitamento dos compiladores existentes, como argumentado na Introdução desta Tese. No caso do Turbo Pascal, ainda há a vantagem do chamado "ambiente integrado", que é o compilador, o linker, o editor de textos e o depurador todos juntos no mesmo software. Seu depurador é muito bom para a maioria dos casos em programas seqüenciais, pois nos permite execução passo a passo, breakpoints, a observação de variáveis ("watches") e a alteração de seus valores. Dessa forma, já temos bastante suporte para uma boa depuração, mas não para programas concorrentes.

O Debugger adiciona-se então de forma a suprir a maior deficiência, que é ter uma visão dos processos e as filas a que eles pertencem em dados momentos da execução. Uma vez que os programas distribuídos feitos com o Kernel são simulados com a idéia das CPUs virtuais, não temos os problemas relativos a estes sistemas descritos na seção anterior. Sendo assim, podemos utilizar normalmente os breakpoints do ambiente Turbo Pascal.

Entretanto, a depuração de um programa concorrente que utiliza preempção traz alguns problemas, mas estes são quase que completamente contornáveis. O primeiro diz respeito a um breakpoint ou chamada do Debugger em um trecho de programa utilizado por mais de um processo ao mesmo tempo, em que o usuário quer parar apenas em um processo específico. Para resolver este problema, deve-se marcar o breakpoint ou chamar o Debugger condicionalmente, seja pelo PID do processo ou uma outra condição qualquer.

Quanto à utilização do depurador do Turbo Pascal em um programa com escalonamento preemptivo, algumas ressalvas devem ser feitas. Como uma depuração passo a passo é conseguida inserido-se um breakpoint automaticamente na próxima linha de programa fonte, e considerando que em um programa concorrente com preempção nem sempre esta linha será a executada, estes comandos (teclas F7 e F8 no Turbo Pascal) devem ser evitados. Em seu lugar, pode ser utilizado o comando da tecla F4 (não aparece no menu - consultar o manual da linguagem), que roda o programa até a linha onde o cursor se encontra. Quanto à preempção de um programa enquanto este estiver parado em um breakpoint, o programador nada tem com que se preocupar. O depurador do Turbo, enquanto esperando por um comando, restaura automaticamente as interrupções de relógio antigas (interrupções $08 e $1C), evitando, portanto, a presença do despachante de processos.

Outro problema, mais sério, refere-se a temporizações. Ao parar em um breakpoint estaríamos alterando seus valores corretos, mas isto não acontece nas operações de Delay e de timeout: estas funcionam com valores relativos e o relógio que as controla é desligado enquanto dura a depuração. No entanto, a operação de Schedule é baseada em tempo absoluto mas implementada considerando um tempo relativo, que é o do momento em que foi chamada até a hora desejada. Desta forma, se for interrompida por uma depuração, o processo acordará atrasado.

Problemas com depuração em programas com temporizações são próprios de sistemas de tempo real (MAGALHÃES, 1986), que é todo um conceito à parte.

Por último, ainda temos o problema de depurar um programa que roda com um "frame" de Time Slice diferente de zero, que é o normal. Neste caso o relógio do computador ficará adiantado, e a única solução para isso é não

utilizar esta característica em fase de testes, ou pelo menos restaurar o "frame" normal com um SetTimeSlice(0) antes de iniciar a depuração.

VI.3 Utilizando o Debugger

Como o Debugger é um módulo de rotinas, e feito especialmente para o Kernel (incluindo os módulos de memória compartilhada e troca de mensagens), ele só pode ser utilizado no contexto deste ambiente e através de uma cláusula "Uses" do Turbo Pascal. A vantagem de se fazer um módulo separado e não embutir o depurador no Kernel é a boa engenharia de software, pois ele pode ser alterado sem maiores problemas de efeitos colaterais. A interface de uso do módulo Debugger se encontra no Anexo V.

Existem duas maneiras de se chamar o Debugger: por um procedimento do próprio módulo e pelo teclado. No primeiro caso insere-se simplesmente no programa sendo depurado a chamada do procedimento *Debug* onde necessário. No segundo caso, podemos chamar o Debugger através de uma combinação de teclas predefinida (comumente chamada de "hotkey") e que pode ser alterada pelo usuário.

Na implementação atual esta chamada pelo teclado só é atendida se o programa sendo depurado estiver solicitando um leitura de teclado ou estiver parado no ambiente integrado do Turbo Pascal, devido a um breakpoint. Além disso, como o IDE (nome dado a este ambiente integrado, abreviatura de "Integrated Development Environment") recaptura algumas interrupções ao atender um breakpoint, dentre elas a que serve o teclado, é necessário a execução de um pequeno programa que acompanha o Kernel, chamado *DebugIDE,* antes de se chamar o Turbo Pascal. Este programa se instala residente na memória e filtra as leituras de teclado à procura da combinação especial que ativa o Debugger. Um teste é feito para que a "hotkey" só funcione se o Debugger também estiver na memória. O uso do

DebugIDE é bastante simples: chamando "debugide" no DOS, ele se instala residente na memória; chamando "debugide -", ele se retira da memória.

Para ativar ou desativar o efeito das chamadas do procedimento Debug existe a variável *DebugOn*. Atribuindo True a ela Debug executa normalmente; atribuindo False as chamadas passam a não ter mais efeito. O default, naturalmente, é True.

VI.4 O Procedimento Debug

Como dito na seção anterior, o procedimento Debug chama o Debugger propriamente dito. Nesta seção veremos os comandos disponíveis para depuração.

O Debugger aparece na tela se sobrepondo ao que estiver escrito nela, mas restaura todos os caracteres e atributos (cores, dimensões de janela, tamanho do cursor, etc) ao final. Desta forma ele pode entrar e sair sem alterar em nada o programa do usuário. Dada essa característica, ele só funciona se o modo corrente de vídeo for texto em 80 colunas (sobre modo gráfico ele simplesmente não entra). O Anexo X mostra a tela de entrada do Debugger e o seu menu principal. Pressionando-se a tecla <Ctrl> e a mantendo neste estado, o menu de comandos muda para o da segunda parte do referido Anexo.

A seguir descreveremos a tela de informações de processos e a função de cada um dos comandos. Os comandos do menu com a tecla <Ctrl> pressionada são precedidos de "<Ctrl>". Para um maior entendimento das informações exibidas, consultar os capítulos anteriores.

VI.4.1 A Tela de Informações dos Processos

Todas as telas de informações são mostradas da mesma forma, ou seja, no centro do vídeo envoltas por uma

moldura. A tela de processos é a que mostra as informações pertinentes a um dado processo, que sempre começa com o processo correntemente executando (o *Running*) ao se entrar no Debugger. São os seguintes os dados relativos a um processo:

* **Process' PID.** O PID do processo, seguido de seu nome, se houver. Todos os processos MAIN de cada CPU virtual possuem o nome default "MAIN-n", onde n é o número de sua CPU. O processo Idle se chama "IDLE".
* **Virtual CPU.** A CPU virtual aonde o processo reside atualmente.
* **State.** O estado corrente do processo.
* **Priority.** O valor da prioridade corrente do processo.
* **Address.** O endereço onde se encontra executando (no caso do Running) ou suspenso um processo.
* **Stack Size.** O tamanho da pilha com que foi criado o processo.
* **Stack Used.** Quantos bytes de sua pilha o processo está usando no momento.
* **Children.** Número de filhos que o processo possui.
* **Dependents.** Número de filhos dependentes (os criados por um Cobegin) que o processo possui. São contados também no campo Children.
* **Save 80x87.** Indica com Yes/No se o processo salva o contexto do co-processador aritmético quando de uma preempção.
* **Waiting Prio.** Prioridade de espera (-32768 a +32767) de um processo numa dada fila em que se encontre bloqueado. Quanto maior o valor, maior sua prioridade.
* **Timeout.** Valor do timeout, em milissegundos, para o caso do processo ficar bloqueado em alguma fila de espera.
* **Rest.** O quanto falta, em milissegundos, para esgotar o timeout do processo, se bloqueado em uma fila de espera.
* **Timed-Out.** Indica com Yes/No se o processo saiu de uma fila de espera devido a um timeout.

No topo da tela, centralizado, se encontra o nome da fila onde o processo se encontra. Esta pode ser a Ready List, a Time List ou uma lista "espionada" pelo usuário (mais sobre isso na seção VI.5).

VI.4.2 Os Comandos do Debugger

- *F1-Running*. Mostra na tela de processos informações sobre o processo correntemente rodando (Running).

- *F2-Run Next*. Mostra na tela de processos informações sobre o próximo processo a ser executado. No caso de uma CPU virtual apenas, idem ao Next (ver abaixo). No caso de mais de uma, mostra o próximo a rodar na próxima CPU virtual ativa.

- *F3-Previous*. Volta ao último processo exibido na tela de informações.

- *F4-Main*. Exibe informações sobre o MAIN da CPU virtual a que pertence o processo exibido na tela.

- *F5-Next*. O próximo do processo exibido na tela, na fila em que ele se encontra.

- *F6-Last*. O anterior ao processo exibido na tela, na fila em que ele se encontra (com exceção da Time List, que não é duplamente encadeada).

- *F7-Next List*. Passa para o primeiro processo da próxima fila espionada pelo usuário. Um sinal audível é emitido se não há mais nenhuma.

- *F8-Last List*. Passa para o primeiro processo da última fila espionada pelo usuário. Um sinal audível é emitido se não há mais nenhuma.

- *F9-CPU Info*. Mostra a tela de informações de CPUs virtuais, uma por vez (ver Anexo XI). Esta tela informa o número da CPU virtual, seu status (Active/Inactive), o número de processos residentes nela e o número de processos ativos em cada fila de prioridade.

- *F10-General Info*. Mostra a tela de informações gerais (ver Anexo XI). Nela constam o número de CPUs virtuais ativas no momento; o número de processos criados, vivos, ativos, esperando, suspensos e temporizando; se o programa está rodando com preempção ou não; o tipo de Time Slice; o tamanho de pilha e prioridade default na criação de processos.

Vale observar quanto à contagem dos processos exibida nesta última tela: *Processes Created* são todos os processos que já foram criados, acabados ou não; *Live Processes* são os criados menos os que já acabaram; *Active Processes* são os que estão com estado Ready, ou seja, prontos para rodar ou rodando; *Waiting Processes* são os que se encontram em alguma fila de espera, à exceção dos exclusivamente na Time List (fila de temporização); *Suspended Processes* são os processos suspensos devido a um Suspend ou um Cobegin; e *Timing Processes* são os que se encontram na Time List, devido a uma temporização solicitada ou a uma espera com timeout. Neste último caso, os processos pertencem a duas filas simultaneamente (a de espera e a Time List), e por isso são contados em "Waiting Processes" e "Timing Processes". Este é o único caso no Kernel em que isso acontece.

- *<Ctrl>F1-Next Prio*. Mostra o primeiro processo na próxima fila de prioridade da CPU virtual corrente (a que pertence o processo que se encontra na tela).

- *<Ctrl>F2-Last Prio*. Mostra o primeiro processo na última fila de prioridade da CPU virtual corrente (a que pertence o processo que se encontra na tela).

- *<Ctrl>F3-Parent*. Mostra o pai do processo exibido na tela. Se não existir, um sinal audível é emitido.

- *<Ctrl>F4-Child*. Mostra o filho mais novo do processo exibido na tela. Se não existir, um sinal audível é emitido. Para se alcançar os outros filhos, usar os comandos abaixo.

- *<Ctrl>F5-Older Bro*. Mostra o irmão mais velho do processo exibido na tela. Se não existir, um sinal audível é emitido.

- *<Ctrl>F6-Younger Bro*. Mostra o irmão mais novo do processo exibido na tela. Se não existir, um sinal audível é emitido.

- *<Ctrl>F7-User Screen*. Desaparece a tela do Debugger momentaneamente para mostrar integralmente a tela do usuário. Pressionando-se qualquer tecla reaparece a tela do Debugger.

- *<Esc>*. Abandona o Debugger. Este comando não consta do menu por serem bastante usuais finalizações com esta tecla.

Um uso em especial do procedimento Debug é no processo Idle. Se fizermos de Debug a rotina de "idle" do usuário, atribuindo-a à variável do Kernel UsrIdleProc, teremos acesso ao Debugger sempre que não houver nenhum processo pronto para executar. Esta facilidade é útil em diversas situações.

VI.5 Espionando Filas de Espera

Observar os processos que estão ativos é trivial, uma vez que podemos acessá-los a partir do processo Running com os comandos *F2-Run Next*, *F5-Next* e *F6-Last*. Mas, e os processos bloqueados em alguma fila de espera? Podemos

acessá-los com os comandos *F7-Next List* e *F8-Last List*, desde que estas filas tenham sido informadas ao Debugger para serem "espionadas".

Para tanto, estão disponíveis os procedimentos *SpyChannel* (para o tipo Channel), *SpySemaphore* (para o tipo Semaphore) e *SpyCondition* (para o tipo Condition). Todos recebem como primeiro parâmetro uma variável do tipo em questão e como segundo parâmetro um nome para a fila, de até 10 caracteres. No caso do tipo Channel, são criadas automaticamente duas filas, uma para "Senders" e outra para "Receivers".

O efeito destes procedimentos é incluir a fila de espera associada a esses três tipos nas listas observadas pelo Debugger. A Time List (uma lista, não uma fila) é sempre incluída por default.

VI.6 Inserindo Delays de Prova

Como dito na seção VI.1, GAIT (1985, 1986) observou que interferências do programador (como leituras de teclado ou breakpoints) ou a inserção de delays em um programa incorreto (deliberadamente pelo programador ou acidentalmente pelo sistema operacional) podem fazê-lo funcionar ou parar seu funcionamento, devido a erros de sincronização. Para testar erros desta natureza, o Debugger dispõe do procedimento *ProbeDelay*, que recebe um valor em milissegundos e fica inerte por este período de tempo, sem deixar nenhum processo ser escalonado.

Segundo o referido autor, os valores críticos para tais testes são na ordem dos décimos de segundo, mas dependem bastante do número de processos existentes.

VI.7 Personalizando o Debugger

O Debugger pode ser adaptado às preferências do usuário em dois itens: hotkey para ser ativado e cores de apresentação.

Para alterar a hotkey, basta atribuir à variável *DebugHotKey* um valor de tecla com o código de "scan" da BIOS. Para facilitar esta tarefa, as constantes F1 a F10, CtrlF1 a CtrlF10 e AltF1 a AltF10 estão disponíveis, por serem as teclas mais elegíveis. A hotkey default é CtrlF10, por não conflitar com nenhuma hotkey do Turbo Pascal.

As cores de fundo, caracteres, moldura e título das telas de informações podem ser alteradas pelas variáveis BackGndColor, ForeGndColor, FrameColor e HeaderColor, respectivamente. Basta para isso atribuir um valor de 0 a 15 para cada uma, segundo o código usual de cores, ou utilizar as constantes predefinidas no módulo CRT do Turbo Pascal. A combinação default é LightGray, Black, Blue e Red.

VI.8 Possíveis Implementações Futuras

Ainda há muito o pode ser feito para se melhorar a capacidade de depuração do Debugger. Trabalhos anteriores incluíram o tempo rodado de cada processo e o tempo útil de cada processador; a seqüência de transmissões e o par origem/destino em uma troca de mensagens; procedimentos para delimitar regiões de "trace" e coletar informações; etc. Talvez a característica mais importante a ser adicionada é a de reprodutibilidade de um programa concorrente, baseada no P-Method (CARVER, 1986) ou no Instant Replay (LEBLANC, 1987). Mais pesquisa neste sentido deve ser feita, pois a área é carente e o assunto bastante complexo.

Outros trabalhos envolvendo depuradores de programas concorrentes podem ser encontrados em WEBER (1983), SMITH (1985), GRIFFIN (1988), ELSHOFF (1988) e PAN (1988). Um compêndio de referências sobre depuradores para programas paralelos pode ser encontrado em PANCAKE (1990).

CAPITULO VII: CONCLUSÕES

O suporte à Programação Concorrente fornecido pelo ambiente aqui apresentado parece preencher uma enorme lacuna que tanto prejudica a difusão deste paradigma. Um software com diversas características, que podem deixar o programador bastante à vontade, aliado a um equipamento hoje popular como um IBM-PC compatível e uma linguagem conhecida e de alto nível como Turbo Pascal formam um conjunto apropriado para resolver a carência da área.

Se compararmos o ambiente Kernel a outros trabalhos do gênero, podemos notar como diferença básica a sua gama de recursos, visando uma maior aplicabilidade. Não obstante, sua implementação levou em consideração o desemepenho de suas primitivas, o que o torna bastante eficiente, além de ser fácil de usar. O maior problema, na verdade, é quanto ao programa concorrente sendo desenvolvido. A falta de prática com seus conceitos, como experimentou o próprio Autor, leva a erros de programação inexistentes em um programa seqüencial, bem mais difíceis de se depurar. Por vezes um erro em um programa de teste foi pensado estar na implementação do Kernel, quando na verdade o programa é que estava errado. Para isso, o depurador incluído no ambiente já é de grande auxílio.

Apesar do ambiente no estágio em que está sendo entregue para uso da comunidade já ser uma ferramenta bastante poderosa, há muito o que pode ser feito ainda. Como o ambiente é composto de quatro módulos distintos, foi preferível apresentar sugestões para futuras implementações ao final dos capítulos dedicados a cada um deles.

ANEXO I: FICHA TÉCNICA DO KERNEL

1. HARDWARE

Microcomputador IBM-PC/XT/AT/386/486 compatível, com mínimo de 640K RAM e qualquer placa gráfica (Hércules, CGA, EGA, VGA, etc).

2. SISTEMA OPERACIONAL

Qualquer versão do MS-DOS (ou similares 100% compatíveis), a partir da 3.0.

3. LINGUAGEM DE PROGRAMAÇÃO

Turbo Pascal a partir da versão 5.0. As facilidades de exclusão mútua automática no uso da heap em alocação de objetos e as estruturas Shared e SignalRegion (do módulo ShareMem) só estão disponíveis a partir da versão 5.5 (com Programação Orientada a Objeto).

4. ARQUIVOS EM DISCO

- **KERNEL.TPU** : Unit principal, para gerência de processos
- **SHAREMEM.TPU**: Unit com estruturas de Memória Compartilhada
- **MSGPASS.TPU** : Unit para troca de mensagens

- **DEBUGGER.TPU**: Unit para depuração de programas concorrentes
- **DEBUGIDE.EXE**: Utilitário auxiliar para depuração

5. PRINCIPAIS CARACTERISTICAS

- Número de processos limitado somente pela memória;

- Três formas diferentes de criação de processos;

- Mecanismos implícitos e explícitos para destruição de processos;

- Modelo das CPUs Virtuais, que permite a presença de mais processadores (lógicos) do que o número de físicos existentes;

- Escalonamento com e sem preempção;

- Escalonamento com política de prioridades;

- Suporte de co-rotinas;

- Migração de processos entre CPUs Virtuais;

- Alteração do time-slice do relógio de hardware;

- Suporte de Tempo Real;

- Implementação de timeouts;

- Salvamento de contexto do co-processador aritmético;

- Comando de execução não-determinística;

- Exclusão mútua automática no manuseio da Heap;

- Exclusão mútua em operações de E/S facilitada;

- Diversas estruturas de comunicação e sincronização, em modelo de Memória Compartilhada ou Distribuída;

- Ferramenta para depuração, permitindo visualizar variáveis do ambiente e blocos de processos, através do percorrimento de listas encadeadas.

ANEXO II: A INTERFACE DO MÓDULO KERNEL

```
Unit Kernel;

Interface

Uses Crt,Dos;

Const
   Version          = '1.00';
   MaxCPUs          = 8;
   MaxPrio          = 2;
   ProcessNameSize  = 8;
   MinStackSize     = 256;
   StackLowerBound  = 192;

Const
   Save80x87        : Boolean = False;
   CopyParentStack  : Boolean = True;

Const
   On       = 254;
   Off      = 255;
   MaxPid   = 65535;

Type
   PidType    = Word;
   StateType  = (Ready,Waiting,Suspended,Timing,
                 Halted,Unknown);
   NameStr    = String[ProcessNameSize];
   ProcFar    = Procedure;

Var
   UsrIdleProc : ProcFar;


Procedure Cobegin;
Procedure Coend;
Procedure Also;
Procedure Fork(Var Pid : PidType);
Procedure LongFork(Prio,StackSize : Word;
                   Var Pid : PidType);
Function  ChildProcess : Boolean;
Function  Spawn(Task : ProcFar) : Word;
Function  LongSpawn(Task : ProcFar;
                    Prio,StackSize : Word) : Word;
Function  GetStackDefault : Word;
Procedure SetStackDefault(StackSize : Word);
Function  GetPrioDefault : Word;
Procedure SetPrioDefault(Prio : Word);

Procedure Kill(Pid : PidType);
Procedure KillMyself;

Procedure Lock;
Procedure Unlock;
Procedure LockFile(Var F : Text);
```

```
Procedure SetHeapMutex;
Procedure ResetHeapMutex;

Procedure Join(n : Word);
Procedure Barrier(n : Word);

Procedure Yield;
Procedure Transfer(Pid : PidType);
Procedure Detach;
Function  CPUid : Word;
Procedure SetProcessors(n : Word);
Procedure SetTimeSlice(NewFrameType : Word);

Procedure Suspend(Pid : PidType);
Procedure Resume(Pid : PidType);
Procedure Migrate(Pid : PidType; NewCPU : Word);
Procedure SetPrio(Prio : Word);
Procedure SetChildPrio(Pid : PidType; Prio : Word);
Function  GetPid : PidType;
Function  GetParentPid : PidType;
Function  GetState(Pid : PidType) : StateType;
Procedure SetName(Name : String);
Function  GetName : NameStr;
Procedure OrphanChild(Pid : PidType);
Procedure OrphanChildren;
Procedure Enable80x87Save;
Procedure Disable80x87Save;

Procedure SetTimeout(ms : Longint);
Function  Timedout : Boolean;
Procedure Delay(ms : Longint);
Procedure Schedule(Hour,Min,Sec : Word);

Procedure HaltAll;
Procedure InstallKernelExit;
Procedure AddCSeg(CS : Word);

Function  Select(n : Word) : Word;
Function  PriSelect(n : Word) : Word;
Procedure When(Guard : Boolean);
```

ANEXO III: A INTERFACE DO MÓDULO SHAREMEM

```
Unit ShareMem;

{$ifdef Ver55}
  {$define OOP}
{$endif}
{$ifdef Ver60}
  {$define OOP}
{$endif}

Interface

Uses Kernel;

Type
   Semaphore     = Record
                     Count : Integer;
                     List  : PCBPtrList;
                   end;
   BinSemaphore  = Boolean;
   EventCount    = Longint;
   Sequencer     = Longint;
   RSemaphore    = Record
                     S      : Semaphore;
                     RSet   : Set of Byte;
                     SetPos : Byte;
                     RMax   : Word;
                   end;

{$ifdef OOP}
   Shared        = Object
                     Mutex     : BinSemaphore;
                     Procedure Init;
                     Procedure Lock;
                     Procedure Unlock;
                   end;
   SignalRegion  = Object
                     Mutex     : BinSemaphore;
                     Procedure NewRegion;
                     Procedure Region;
                     Procedure EndRegion;
                   end;
   Condition     = Object
                     List      : PCBPtrList;
                     MutexPtr  : ^BinSemaphore;
                     Procedure NewCondition(Var SR :
                                            SignalRegion);
                     Procedure Wait;
                     Procedure WaitP(Prio : Word);
                     Procedure Signal;
                   end;
{$endif}
```

```
Procedure InitSemaphore(Var S : Semaphore;
                        Value : Integer);
Procedure P(Var S : Semaphore);
Procedure PPrio(Var S : Semaphore; Prio : Integer);
Procedure V(Var S : Semaphore);
Function  SemaphoreCount(Var S : Semaphore) : Integer;
Function  Awaited(Var S : Semaphore) : Boolean;


Procedure InitBinSemaphore(Var B : BinSemaphore);
Procedure PBin(Var B : BinSemaphore);
Procedure VBin(Var B : BinSemaphore);


Procedure InitEventCount(Var E : EventCount);
Procedure Advance(Var E : EventCount);
Function  ReadE(Var E : EventCount) : EventCount;
Procedure Await(Var E : EventCount; v : Longint);


Procedure InitSequencer(Var Seq : Sequencer);
Function  Ticket(Var Seq : Sequencer) : Longint;


Procedure  InitRSemaphore(Var RS : RSemaphore;
                          Resources : Word);
Procedure RP(Var RS : RSemaphore;
             Var ResourceAlloc : Byte);
Procedure RV(Var RS : RSemaphore;
             Var ResourceFreed : Byte);
```

ANEXO IV: A INTERFACE DO MÓDULO MSGPASS

```
Unit MsgPass;

Interface

Uses Kernel;

Const
   Infinity = 65535;

Type
   BufferType    = Array[1..1] of Byte;
   SenderType    = PCBPtr;
   SenderArray   = Array[1..1] of SenderType;
   ReceiverType  = Record
                     PCB          : PCBPtr;
                     BroadcastNum : Integer;
                   End;
   ReceiverArray = Array[1..1] of ReceiverType;
   UserType      = (Sender,Receiver);
   MsgRecPtr     = ^MsgRecType;
   MsgRecType    = Record
                     Tag       : Integer;
                     Prio      : Integer;
                     Receivers : Word;
                     MsgPtr    : Pointer;
                     NextMsg   : MsgRecPtr;
                   End;
   Channel       = Record
                     Owner        : PidType;
                     MaxSenders   : Word;
                     MaxReceivers : Word;
                     SendersNum   : Word;
                     ReceiversNum : Word;
                     Senders      : ^SenderArray;
                     Receivers    : ^ReceiverArray;
                     SenderList   : PCBPtrList;
                     ReceiverList : PCBPtrList;
                     BroadcastNum : Integer;
                     InMsgs       : Longint;
                     MsgSize      : Word;
                     Buffers      : Word;
                     BufSize      : Word;
                     BufHead      : MsgRecPtr;
                     BufTail      : MsgRecPtr;
                   end;

Var
   Any : Byte Absolute 0:0;
```

```
Procedure CreateChannel(Var C : Channel; MaxSenders,
                        MaxReceivers, MsgSize,
                        Buffers : Word);
Procedure DestroyChannel(Var C : Channel);
Procedure UseChannel(Var C : Channel; User : UserType);
Procedure FreeChannel(Var C : Channel);


Procedure Send(Var C : Channel; Var Msg);
Procedure Multicast(Var C : Channel; Var Msg;
                    Receivers : Word);
Procedure Broadcast(Var C : Channel; Var Msg);
Procedure PriSend(Var C : Channel; Var Msg;
                  Prio : Integer);
Procedure PriMulticast(Var C : Channel; Var Msg;
                       Receivers : Word;
                       Prio : Integer);
Procedure PriBroadcast(Var C : Channel; Var Msg;
                       Prio : Integer);


Procedure Receive(Var C : Channel; Var Msg);
Procedure ReceiveSynch(Var C : Channel; Senders : Word);


Function  SendOk(Var C : Channel; Var Msg) : Boolean;
Function  ReceiveOk(Var C : Channel; Var Msg) : Boolean;


Procedure ChannelStats(Var C : Channel;
                       Var InMsgs, OutMsgs : Longint);
```

ANEXO V: A INTERFACE DO MÓDULO DEBUGGER

```
Unit Debugger;

Interface

Uses Crt,Dos,Kernel,MsgPass,ShareMem;

Const
   F1     = 59;      CtrlF1  = 94;     AltF1  = 104;
   F2     = 60;      CtrlF2  = 95;     AltF2  = 105;
   F3     = 61;      CtrlF3  = 96;     AltF3  = 106;
   F4     = 62;      CtrlF4  = 97;     AltF4  = 107;
   F5     = 63;      CtrlF5  = 98;     AltF5  = 108;
   F6     = 64;      CtrlF6  = 99;     AltF6  = 109;
   F7     = 65;      CtrlF7  = 100;    AltF7  = 110;
   F8     = 66;      CtrlF8  = 101;    AltF8  = 111;
   F9     = 67;      CtrlF9  = 102;    AltF9  = 112;
   F10    = 68;      CtrlF10 = 103;    AltF10 = 113;

   HotKeyDefault = CtrlF10;
   ListNameLen   = 10;

Var
   ICA           : Array[1..16] of Byte Absolute 0:$4F0;
   DebugHotKey   : Byte Absolute ICA;

Type
   FrameCharType = (UpperLeft,LowerLeft,UpperRight,
                    LowerRight,Horiz,Vert);
   FrameArray    = Array[FrameCharType] of Char;
   ListNameStr   = String[ListNameLen];

Const
   FrameChars      : FrameArray = '╔╚╗╝═║';
   LeftCol         = 22;
   RightCol        = 56;
   TopRow          = 06;
   BottomRow       = 21;
   BackGndColor    : Byte = LightGray;
   ForeGndColor    : Byte = Black;
   FrameColor      : Byte = Blue;
   HeaderColor     : Byte = Red;
   MaxSpyLists     = 10;
   VideoPage       = 3;

Const
   DebugOn : Boolean = True;

Procedure Debug;
Procedure ProbeDelay(ms : Word);

Procedure SpyChannel(Var C : Channel; Name : ListNameStr);
Procedure SySemaphore(Var S : Semaphore;
                      Name : ListNameStr);
Procedure SpyCondition(Var C : Condition;
                       Name : ListNameStr);
```

ANEXO VI: MENSAGENS DE ERRO DO MÓDULO KERNEL

ERRO 1: INVALID STACK SIZE

O tamanho de pilha passado como parâmetro na criação de um processo com LongFork ou LongSpawn, ou ainda na alteração do tamanho default com SetStackDefault, está fora da faixa permitida. Valores legais estão entre 256 (MinStackSize) e 65520.

ERRO 2: INVALID PRIORITY

O valor de prioridade passado como parâmetro na criação de um processo com LongFork ou LongSpawn, na alteração do valor default com SetPrioDefault ou ainda na mudança de prioridade de um processo com SetPrio ou SetChildPrio está fora da faixa permitida. Valores legais estão entre 0 e 2 (MaxPrio).

ERRO 3: INVALID PROCESSOR ALLOCATION

Utilização errônea de SetProcessors. Ou o parâmetro passado como número de CPUs virtuais a serem alocadas supera o máximo (MaxCPUs), ou esta operação já foi realizada antes. Só é permitido um único uso de SetProcessors durante todo o programa.

ERRO 4: INVALID PROCESSOR

Utilização errônea de Migrate. A CPU virtual destino do processo sendo migrado não existe.

ERRO 5: INVALID FRAME TYPE

Valor de "frame" passado como parâmetro em SetTimeSlice é maior que 15.

ERRO 6: INVALID PID

Foi feita uma referência a um processo cujo PID é desconhecido, seja pelo processo já ter sido destruído, estar fora da família de quem executou a operação ou simplesmente por ser um valor de PID inválido. Todos os procedimentos que manipulam um processo pelo seu PID podem acarretar este erro.

ERRO 7: STACK OVERFLOW

Não há mais espaço na pilha de um processo para alocação de variáveis locais, chamada de procedimentos, passagem de parâmetros ou qualquer outro tipo de empilhamento. A solução é criar o processo com uma pilha maior.

ERRO 8: INVALID TRANSFER

Uma tentativa ilegal de transferência de uso do processador aconteceu. Um Transfer não pode ser feito para um processo que não esteja em estado Ready ou tenha prioridade menor que a do processo Running.

ERRO 9: PROCESS CREATION IMPOSSIBLE

Não há mais memória disponível na heap para criação de processos. Este é o único fator limitante do número máximo de processos manipulados pelo Kernel. Deve-se aumentar o tamanho da heap com a diretiva M+, desalocar variáveis dinâmicas fora de uso ou rodar o programa com mais memória do sistema disponível. Isto pode ser feito reduzindo o número de programas residentes na memória ou retirando-se algum "device driver" sem uso do CONFIG.SYS, ou em último caso rodando o programa como um .EXE a partir do DOS (ganhando toda a memória ocupada pelo Turbo Pascal).

ERRO 10: INVALID COBEGIN USE

Um Cobegin/Also/Coend está sintaticamente incorreto, como, por exemplo, sem o Coend; tem apenas um único processo no seu interior, o que é ilegal (muitas vezes ocasionado pelo esquecimento do Also); ou ainda o código de um processo em seu interior superou 1024 bytes (MaxCobeginCode).

ERRO 11: PROCESS KILLED OR ALREADY SUSPENDED

Um Suspend foi tentado sobre um processo já suspenso ou que está sendo destruído.

ERRO 12: INVALID RESUME

Um Resume foi tentado sobre um processo que não está suspenso.

ERRO 13: DEADLOCK

O programa entrou em deadlock.

ERRO 14: ALSO OR COEND MISUSED

Um processo chamou um Also ou um Coend fora de uma estrutura Cobegin/Also/Coend.

ERRO 15: PREVIOUS TIMEOUT UNRECOGNIZED

Um processo sofreu um segundo timeout sem ter ainda reconhecido o primeiro, através da função Timedout.

ANEXO VII: MENSAGENS DE ERRO DO MÓDULO MSGPASS

ERRO 1: ALREADY USER OF THIS CHANNEL

Tentativa ilegal de usar um mesmo canal (UseChannel) já sendo usuário dele.

ERRO 2: NO MORE SENDERS ALLOWED

O número máximo de Senders permitido no canal, definido com CreateChannel, foi atingido.

ERRO 3: NO MORE RECEIVERS ALLOWED

O número máximo de Receivers permitido no canal, definido com CreateChannel, foi atingido.

ERRO 4: NOT A USER OF THIS CHANNEL

Operação ilegal sobre um canal, pois o processo não é usuário dele.

ERRO 5: NOT A SENDER OF THIS CHANNEL

Operação de transmissão ilegal, pois o processo não é usuário do canal no modo Sender.

ERRO 6: NOT A RECEIVER OF THIS CHANNEL

Operação de recepção ilegal, pois o processo não é usuário do canal no modo Receiver.

ERRO 7: NOT THE CHANNEL'S OWNER

Operação DestroyChannel ilegal, pois o processo que a chamou não é o dono do canal. O dono de um canal é o processo que executou CreateChannel.

ERRO 8: OUT OF MEMORY

Falta de memória na heap para criação de um novo canal ou alocação de uma mensagem pendente. Deve-se aumentar o tamanho da heap com a diretiva M+, desalocar variáveis dinâmicas fora de uso ou rodar o programa com mais memória do sistema disponível. Isto pode ser feito reduzindo o número de programas residentes na memória ou retirando-se algum "device driver" sem uso do CONFIG.SYS, ou em último caso rodando o programa como um .EXE a partir do DOS (ganhando toda a memória ocupada pelo Turbo Pascal).

ANEXO VIII: EXEMPLO DE PROGRAMA COM O KERNEL

```
Program Producer_Consumer;

Uses Crt,Kernel,ShareMem,Debugger;

Const
   BufSize    = 10;
   TotalItems = 50;

Type
   SharedBufferType = Object(SignalRegion)
                        Buffer   : Array[1..BufSize] of
                                   Word;
                        Head     : Word;
                        Tail     : Word;
                        Size     : Word;
                        NotEmpty : Condition;
                        NotFull  : Condition;
                      end;

Var
   SharedBuffer : SharedBufferType;
   Finished     : Boolean;


Procedure Insert(x : Word);

begin
  With SharedBuffer do begin
    Region;
      if Size = BufSize then
        NotFull.Wait;
      Buffer[Tail]:=x;
      Inc(Size);
      Tail:=(Tail mod BufSize)+1;
    NotEmpty.Signal;
  end;
end;


Procedure Remove(Var x : Word);

begin
  With SharedBuffer do begin
    Region;
      if Size = 0 then
        NotEmpty.Wait;
      x:=Buffer[Head];
      Dec(Size);
      Head:=(Head mod BufSize)+1;
    NotFull.Signal;
  end;
end;
```

```
Procedure InitSharedBuffer;

begin
  With SharedBuffer do begin
    NewRegion;
    Head:=1;
    Tail:=1;
    Size:=0;
    NotEmpty.NewCondition(SharedBuffer);
    NotFull.NewCondition(SharedBuffer);
  end;
end;


Procedure Producer;

Var
   i : Word;

begin
  SetName('Producer');
  for i:=1 to TotalItems do begin
    Delay(Random(500));
    Insert(i);
    write('P',i,'  ');
  end;
  Finished:=True;
end;


Procedure Consumer;

Var
   i : Word;

begin
  SetName('Consumer');
  While not Finished or (SharedBuffer.Size > 0) do begin
    Delay(Random(500));
    Remove(i);
    write('C',i,'  ');
  end;
end;


begin
  SpyCondition(SharedBuffer.NotEmpty,'Not Empty');
  SpyCondition(SharedBuffer.NotFull,'Not Full');
  Clrscr;
  Randomize;
  InitSharedBuffer;
  Finished:=False;
  cobegin;
    Producer; Also;
    Consumer;
  coend;
end.
```

ANEXO IX: PERFORMANCE DOS MECANISMOS DE EXCLUSÃO MUTUA DO AMBIENTE KERNEL

O programa ShareMem_Example1, listado ao final deste Anexo, realiza a mesma tarefa com todos os mecanismos de exclusão mútua do ambiente Kernel. A tarefa em questão é incrementar de duas unidades uma variável de somatório global, realizada por dois processos que fazem um loop de "n" iterações. O programa rodou em um micro PC-AT com clock de 20MHz, fazendo n igual a 30.000 (ou seja, a variável de somatório deve ter um valor final de 120.000).

Os tempos de execução dos mecanismos testados encontram-se no quadro abaixo.

MECANISMO	TEMPO ABS. (s)	TEMPO RELATIVO
Sem exclusão mútua	0.5435	1
Lock/Unlock	0.6234	1.147
Semáforo Binário	1.1525	2.121
Variável Shared I	1.5343	2.823
Signalling Region	1.5353	2.825
Variável Shared II	1.5690	2.887
Semáforo	21.7050	39.936

Os tempos exibidos neste quadro não devem ser tomados como fixos de uma maneira geral, mas somente para se ter uma noção de comparação entre os diversos mecanismos. O mesmo programa foi executado em um AT com

clock de 16MHz, causando algumas diferenças nos tempos relativos (principalmente no tocante ao mecanismo "semáforo"), mas sempre conservando a mesma ordem de tempos de execução.

O importante realmente é a conclusão que podemos chegar observando o quadro: o mecanismo Lock/Unlock do Kernel é sem dúvida o mais rápido de todos, mas, como dito no Capítulo III, deve ser utilizado com parcimônia. É preferível usar um semáforo binário, mecanismo simples e que possui tempo comparável ao Lock/Unlock. Variável Shared I, que é o uso de Shared tal qual um semáforo binário, tem performance semelhante a uma Signalling Region. Variável Shared II, em que Shared é utilizado na forma usual de objeto, fazendo parte da variável a ser protegida, tem performance um pouco pior que as duas anteriores. Por último, temos os semáforos, com um tempo extremamente elevado. Isto se deve a haver, neste caso, muitas entradas e saídas de listas do Kernel, que são operações mais demoradas. Contudo, pode ser utilizado com bastante vantagem em situações em que isto não aconteça.

```
Program ShareMem_Example1;

Uses Crt,Kernel,ShareMem,debugger;

{$ifdef Ver55}
  {$define OOP}
{$endif}
{$ifdef Ver60}
  {$define OOP}
{$endif}

Const
   n = 30000;

Var i,j,Sum : Longint;
```

```
Procedure Demo;

begin
  Sum:=0;
  cobegin;
    for i:=1 to n do
      Sum:=Sum+2;
    Also;
    for j:=1 to n do
      Sum:=Sum+2;
  coend;
  writeln('Soma sem exclusao mutua: ',Sum);
end;

Var s : Semaphore;

Procedure Demo_Semaforo;

begin
  InitSemaphore(s,1);
  Sum:=0;
  cobegin;
    for i:=1 to n do begin
      P(s);
      Sum:=Sum+2;
      V(s);
    end;
    Also;
    for j:=1 to n do begin
      P(s);
      Sum:=Sum+2;
      V(s);
    end;
  coend;
  writeln('Soma com exclusao mutua por semaforo: ',Sum);
end;

Var b : BinSemaphore;
```

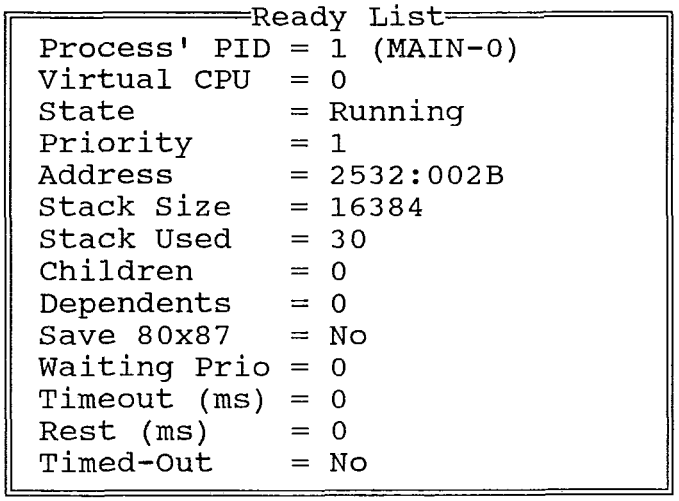
```
Procedure Demo_SemaforoBinario;

begin
  InitBinSemaphore(b);
  Sum:=0;
  cobegin;
    for i:=1 to n do begin
      PBin(b);
      Sum:=Sum+2;
      VBin(b);
    end;
    Also;
    for j:=1 to n do begin
      PBin(b);
      Sum:=Sum+2;
      VBin(b);
    end;
  coend;
  writeln(
     'Soma com exclusao mutua por semaforo binario: ',Sum);
end;

{$ifdef OOP}

Var sh : Shared;

Procedure Demo_Shared1;

begin
  sh.init;
  Sum:=0;
  cobegin;
    for i:=1 to n do begin
      sh.lock;
      Sum:=Sum+2;
      sh.unlock;
    end;
    Also;
```

```
    for j:=1 to n do begin
      sh.lock;
      Sum:=Sum+2;
      sh.unlock;
    end;
  coend;
  writeln(
      'Soma com exclusao mutua por Variavel Shared: ',Sum);
end;

Type SomaType = Object(Shared)
                  x : Longint;
                end;

Var Soma : SomaType;

Procedure Demo_Shared2;

begin
  Soma.init;
  Soma.x:=0;
  cobegin;
    for i:=1 to n do begin
      Soma.lock;
      Soma.x:=Soma.x+2;
      Soma.unlock;
    end;
    Also;
    for j:=1 to n do begin
      Soma.lock;
      Soma.x:=Soma.x+2;
      Soma.unlock;
    end;
  coend;
  writeln(
'Soma com exclusao mutua por Variavel Shared Herdada: ',
Soma.x);
end;
```

```
Var sr : SignalRegion;

Procedure Demo_SignalRegion;

begin
  sr.NewRegion;
  Sum:=0;
  cobegin;
    for i:=1 to n do begin
      sr.region;
        Sum:=Sum+2;
      sr.endregion;
    end;
    Also;
    for j:=1 to n do begin
      sr.region;
        Sum:=Sum+2;
      sr.endregion;
    end;
  coend;
  writeln(
   'Soma com exclusao mutua por Signalling Region: ', Sum);
end;
{$endif}
```

```
Procedure DemoKernel;

begin
  Sum:=0;
  cobegin;
    for i:=1 to n do begin
      Lock;
      Sum:=Sum+2;
      Unlock;
    end;
    Also;
    for j:=1 to n do begin
      Lock;
      Sum:=Sum+2;
      Unlock;
    end;
  coend;
  writeln(
  'Soma com exclusao mutua do Kernel (Lock/Unlock): ',Sum);
end;


begin
  Clrscr;
  Demo;
  Demo_Semaforo;
  Demo_SemaforoBinario;
  {$ifdef OOP}
  Demo_Shared1;
  Demo_Shared2;
  Demo_SignalRegion;
  {$endif}
  DemoKernel;
end.
```

ANEXO X: A TELA DE ENTRADA DO MÓDULO DEBUGGER



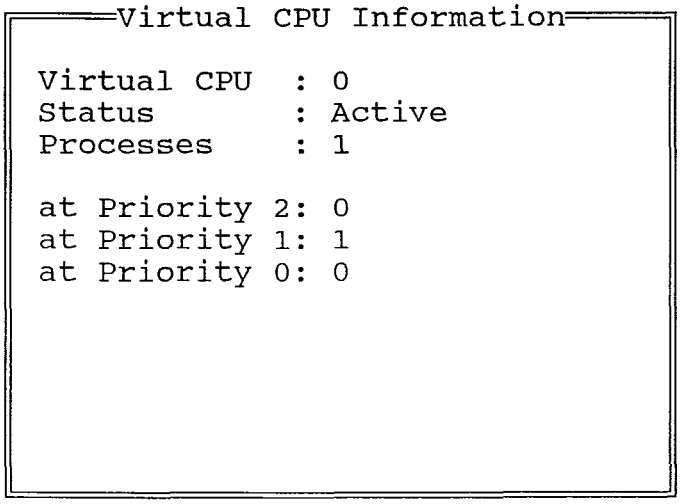
F1-Running **F3**-Previous **F5**-Next **F7**-Next List **F9**-CPU Info
F2-Run Next **F4**-Main **F6**-Last **F8**-Last List **F10**-Gen. Info

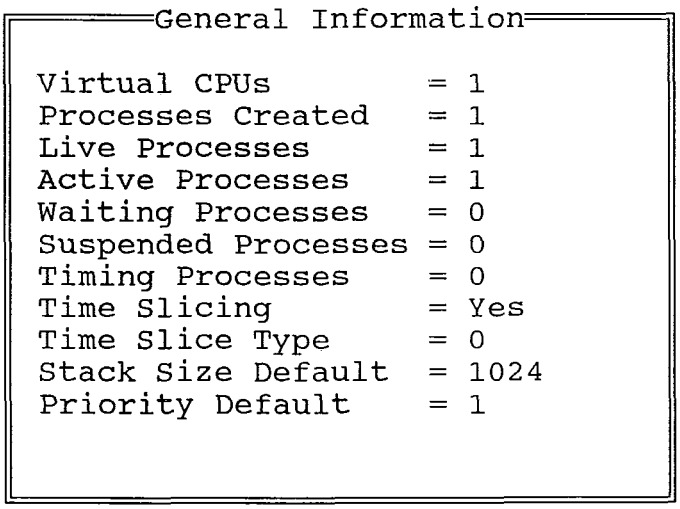
---

Ao pressionar <Ctrl>, a parte inferior da tela muda para o seguinte menu:

F1-Next Prio **F3**-Parent **F5**-Older Bro **F7**-User Screen
F2-Last Prio **F4**-Child **F6**-Younger Bro

ANEXO XI: AS TELAS DE CPUS VIRTUAIS E INFORMAÇÕES GERAIS DO MODULO DEBUGGER





REFERÊNCIAS BIBLIOGRAFICAS

AHO (1983) A.V. Aho, J.E. Hopcroft, J.D. Ullman, *Data Structures and Algorithms*, Addison-Wesley.

AHO (1986) A.V. Aho, R. Sethi, J.D. Ullman, *Compilers - Principles, Techniques and Tools*, Addison-Wesley.

ALMES (1986) G. Almes, "The Impact of Language and System on Remote Procedure Call Design", *Proceedings of the 6th International Conference on Distributed Computing Systems*, 414-421.

ANDREWS (1982a) G.R. Andrews, "The Distributed Programming Language SR - Mechanisms, Design and Implementation", *Software - Practice and Experience*, 12 (8), 719-753.

ANDREWS (1982b) G.R. Andrews, "Distributed Programmming Languages", *Proceedings of the ACM 1982 Conference*, 113-117.

ANDREWS (1983) G.R. Andrews, F.B. Schneider, "Concepts and Notations For Concurrent Programming", *ACM Computing Surveys*, 15 (1), 3-44.

ARAÚJO (1984) C.C.F. Araújo, *Especificação de um Núcleo de um Sistema Operacional Para Suporte dos Atributos de Concorrência da Linguagem Chill em Sistemas Distribuídos: Uma Aplicação em Telefonia*, Tese M.Sc. COPPE/UFRJ, Rio de Janeiro, Brasil.

BAGRODIA (1985) R. Bagrodia, K. M. Chandy, "A Micro-Kernel For Distributed Applications", *Proceedings of the 5th International Conference on Distributed Computing Systems, IEEE*, 140-149.

BAL (1989) H.E. Bal, J.G. Steiner, A.S. Tanenbaum, "Pro-

grammming Languages For Distributed Computing Systems", *ACM Computing Surveys* 21 (3), 261-322.

BARBOSA (1982) V.C. Barbosa, *Uma Proposta Para Estender o Sistema Pascal UCSD a Processamento Concorrente*, Tese M.Sc. COPPE/UFRJ, Rio de Janeiro, Brasil.

BASU (1987) J. Basu, L. M. Patnaik, A. K. Goswami, "Ordered Ports- A Language Concept For High Level Distributed Programming", *The Computer Journal*, 30 (6), 487-497.

BEAUMONT (1978) W.P. Beaumont, "An Implementation of Structured Multiprogramming", *Software - Practice and Experience*, 8 (3), 313-322.

BEMMERL (1987) T. Bemmerl, G. Schoder, "A Portable Real-time Multitasking Kernel For Embedded Microprocessor Systems", *Microprocessing and Microprogramming*, 21 (1-5), 181-188.

BEN-ARI (1982) M. Ben-Ari, *Principles of Concurrent Programming*, Prentice-Hall.

BERGMANN (1986) E.E. Bergmann, "Concurrency and Turbo Pascal", *Dr. Dobb's Journal of Software Tools*, 11 (3), 36-N39.

BERRY (1986) K. Berry, "A Multitasking Kernel", *Dr. Dobb's Journal of Software Tools*, 11 (12), 16-N27.

BITAR (1988) J.M.N. Bitar, *Um Sistema de Troca de Mensagens Para Comunicação e Sincronização entre Processos em Modula-2*, Tese M.Sc. COPPE/UFRJ, Rio de Janeiro, Brasil.

BORLAND (1989a) Borland International Inc., *Turbo Pascal 5.5 Object- Oriented Programming Guide*.

BORLAND (1989b) Borland International Inc., *Turbo Pascal Owner's Handbook*.

BOWEN (1980) B.A. Bowen, R.J.A. Buhr, *The Logical Design of Multiple-microprocessors Systems*, Prentice-Hall.

BOWLING (1989) "Real-Time Modeling With MS-DOS", *Dr. Dobb's Journal of Software Tools*, February, 26-34.

BREITINGER (1983) J.L. Breitinger, *Núcleo de Um Sistema Operacional Multiprogramado*, Tese M.Sc. COPPE/UFRJ, Rio de Janeiro, Brasil.

BRINCH HANSEN (1972) P. Brinch Hansen, "Structured Multiprogramming", *Comm. of the ACM*, 15, 574-578.

BRINCH HANSEN (1973) P. Brinch Hansen, *Operating System Principles*, Prentice-Hall, Englewood Cliffs, New Jersey.

BRINCH HANSEN (1975) P. Brinch Hansen, "The Programming Language Concurrent Pascal", *IEEE Trans. on Soft. Eng.* SE-1 (2), 199-207.

BRINCH HANSEN (1978) P. Brinch Hansen, "Distributed Processes: a Concurrent Programming Concept", *Comm. ACM* 21 (11), 934-941.

BRINCH HANSEN (1981) P. Brinch Hansen, "Edison - A Multiprocessor Language", *Software - Practice and Experience*, 12 (4), 325-361.

BRINCH HANSEN (1983) P. Brinch Hansen, "Using Personal Computers in Operating Systems Courses", *ACM SIGOPS Operating Systems Review* 17 (3), 41-50.

BRINCH HANSEN (1987a) P. Brinch Hansen, "Joyce - A Programming Language For Distributed Systems", *Software -*

Practice and Experience 17 (1), 29-50.

BRINCH HANSEN (1987b) P. Brinch Hansen, "A Joyce Implementation", *Software - Practice and Experience* 17 (4), 267-276.

BUHLER (1989) P. Buhler, D. Wybranietz, "Tools For Distributed Programming in the INCAS Project", *Microprocessing and Microprogramming*, 27 (1-5), 199-206.

BUHR (1990) P.A. Buhr, R.A. Strooboscher, "The μSystem: Providing Light-Weight Concurrency on Shared-memory Multiprocessors Computers Running Unix", *Software - Practice and Experience*, 20 (9) 929-964.

CARVER (1986) R. Carver, K. Tai, "Reproducible Testing of Concurrent Programs Based on Shared Variables", *Proceedings of the 6th International Conference on Distributed Computing Systems*, 428-433.

CERRADA (1988) J. A. Cerrada, M. Collado, "Implementation of a CSP-Based Extension of Pascal", *Microprocessing and Microprogramming*, 22 (4), 233-242.

CHANDRA (1988) P. Chandra, "Programming the 80387 Coprocessor", *Byte* 13 (3), 207-215.

CHANDY (1985) K.M. Chandy, L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", *ACM Trans. on Comp. Systems*, 3 (1), 63-75.

CHERITON (1979) D.R. Cheriton, M.A. Malcolm, L.S. Melen, G.R. Sager, "Thoth, a Portable Real-Time Operating System", *Comm. ACM* 22 (2), 105-115.

COMER (1984) D. Comer, *Operating System Design - The XINU Approach*, Prentice-Hall.

CONWAY (1963) M. Conway, "A Multiprocessor System Design", *Proceedings of the AFIPS Fall Joint Computer Conference*, 139-146.

COOPER (1987) R. Cooper, "Pilgrim: A Debugger for Distributed Systems", *Proceedings of the 7th International Conference on Distributed Computing Systems*, 458-465.

CORMACK (1988) G.V. Cormack, "A Micro-Kernel For Concurrency in C", *Software - Practice and Experience*, 18 (5), 485-491.

CRAMER (1988) B. Cramer, "Writing Real-Time Programs Under Unix", *Dr. Dobb's Journal of Software Tools*, 13(6) 18-33.

CROOKES (1984) D. Crookes, J.W.G. Elder, "An Experiment in Language Design For Distributed Systems", *Software - Practice and Experience*, 14 (10), 957-971.

DE BUZIN (1985) P.F.W.K. de Buzin, "Um Modelo Para a Concepção de Sistemas Distribuídos", *Anais do XII SEMISH, V Congresso da SBC*, 447-454.

DE RIDDER (1986) T. De Ridder, "Coroutines For C Reconsidered", *Software - Practice and Experience*, 16 (3), 301-302.

DEITEL (1984) H.M. Deitel, *An Introduction to Operating Systems* (Revised First Edition), Addison-Wesley.

DIJKSTRA (1965) E.W. Dijkstra, "Cooperating Sequential Processes", *Technical Report EWD-123, Technological University, Eindhoven, The Netherlands*. Reprinted in F. Genuys (Editor), "Programming Languages", Academic Press, London (1968).

DIJKSTRA (1975) E.W. Dijkstra, "Guarded Commands, Nondeterminancy and Formal Derivation of Programs", *Comm. ACM* 18 (8), 453-457.

DOS REIS (1988) A.J. Dos Reis, "A Note On Ben-Ari's Concurrent Programmming System", *ACM SIGOPS Operating System Review* 22 (3), 41-42.

DOWNES (1982) V.A. Downes, S.J. Goldsack, *Programming Embedded Systems with Ada*, Prentice-Hall.

DUBNICKI (1988) C. Dubnicki, J. Madey, W. Wygladala, "Edison-N - An Edison Implementation For a Network of Microcomputers", *Software - Practice and Experience* 18 (4), 349-363.

DUGLOSZ (1988) J.M. Duglosz, "A Multitasking Kernel For C Programmers", *Computer Language* 5(10), 49-61.

DUGLOSZ (1989) J.M. Duglosz, "Preemptive Tasking in C++", *Computer Language*, November, 36-51.

ELSHOFF (1988) I.J.P. Elshoff, "A Distributed Debugger For Amoeba", *Proceedings of the ACM/SIGPLAN and SIGPOS Workshop on Parallel and Distributed Debugging* (SIGPLAN Notices 24, 1-2), 1-10.

FARRAN (1981) Y. Farran, M. Stanton, "Programação Concorrente numa Linguagem de Alto Nível - Uma Implementação", *Anais do VIII SEMISH, I Congresso da SBC*, 125-135.

FIELD (1988) T. Field, "The IBM PC and the Intel 8087 Coprocessor, Part 1: Overview and Floating-Point Assembly-language Support", *Byte* 8 (8), 331-374.

FIELD (1988) T. Field, "The IBM PC and the Intel 8087 Coprocessor, Part 2: Interfacing to IBM Pascal", *Byte* 8

(9), 331-355.

FISHER (1988) A.J. Fisher, "A Critique of OCCAM Channel Types", *Computer Languages*, 13 (2), 95-105.

FREISLEBEN (1989) B. Freisleben, J. L. Keedy, "Priority Semaphores", *The Computer Journal*, 32 (1), 24-28.

GAIT (1985) J. Gait, "A Debugger For Concurrent Programs", *Software - Practice and Experience* 15 (6), 539-554.

GAIT (1986) J. Gait, "A Probe Effect in Concurrent Programs", *Software - Practice and Experience*, 16 (3), 225-233.

GARETTI (1982) P. Garetti, P. Laface, S. Rivoira, "MODOSK: A Modular Distributed Operating System Kernel For Real-Time Process Control", *Microprocessing and Microprogramming* 9 (4), 201-213.

GEHANI (1984) N. H. Gehani, "Broadcasting Sequential Processes (BSP)", *IEEE Trans. on Software Eng.*, SE-10 (4), 343-351.

GEHANI (1986) N. H. Gehani, W. D. Roome, "Concurrent C", *Software - Practice and Experience* 16 (9), 821-844.

GEHANI (1988) N.H. Gehani, W.D. Roome, "Concurrent C++: Concurrent Programming With Class(es)", *Software - Practice and Experience*, 18 (12), 1157-1178.

GEHANI (1989) N.H. Gehani, W.D. Roome, "Concurrent C For Real-Time Programming", *Dr. Dobb's Journal of Software Tools*, November, 38-45 .

GEHANI (1990) N. H. Gehani, "Message Passing in Concurrent C: Synchronous Versus Asynchronous", *Software - Practice and Experience* 20 (6), 571-592 .

GENTLEMAN (1981) W.M. Gentleman, "Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept", *Software - Practice and Experience*, 11 (5), 435-466.

GOOD (1979) D.I. Good, R.M. Cohen, "Principles of Proving Concurrent Programs in Gypsy", *Sixth Annual ACM Symposium on Principles of Programming Languages*, 42-52.

GOULART (1989) C.F.R.A. Goulart, L.E.B. Moura, *Um Compilador Para a Linguagem CRIS*, Projeto Final do Curso de Informática, IM/UFRJ, Rio de Janeiro, Brasil.

GRASS (1986) J. Grass, R. Campbell, "Mediators: A Synchronization Mechanism", *Proceedings of the 6th International Conference on Distributed Computing Systems*, 468-477.

GREEN (1989) T. Green, "A C++ Multitasking Kernel", *Dr. Dobb's Journal of Software Tools*, February, 45-51.

GRIFFIN (1988) J. H. Griffin, H. J. Wasserman, L. P. McGavran, "A Debugger For Parallel Processes", *Software - Practice and Experience*, 18 (12), 1179-1190.

HABERMANN (1980) A.N. Habermann, "Efficient Implementation of ADA Tasks", *Technical Report CMU-CS-80-103, Carnegie-Mellon University*.

HANSON (1990), D. R. Hanson, Letter to the Editor, *Software - Practice and Experience*, 20 (9), 965.

HARLAND (1985) D.M. Harland, "Towards a Language For Concurrent Processes", *Software - Practice and Experience*, 15 (9), 839-888.

HEMMENDIGER (1988) D. Hemmendiger, "A Correct Implementa-

tion of General Semaphores", *ACM SIGOPS Operating Systems Review* 22 (3) 42-44.

HEMMENDIGER (1989) D. Hemmendiger, "Comments on 'A Correct and Unrestrictive Implementation of General Semaphores'", *ACM SIGOPS Operating Systems Review* 23 (1), 7-8.

HIKITA (1985) T. Hikita, K. Ishihata, "A Method of Program Transformation Between Variable Sharing and Message Passing", *Software - Practice and Experience*, 15 (7), 677-692.

HOARE (1974) C.A.R. Hoare, "Monitors: An Operating System Structuring Concept", *Comm. ACM* 17 (10), 549-557. Erratum in Comm. ACM 18 (2), p. 95 (1975).

HOARE (1978) C.A.R. Hoare, "Communicating Sequential Processes", *Comm. ACM* 21 (4), 666-677.

HOLT (1978) R.C. Holt, G.S. Graham, E.D. Lazowska, M.A. Scott, *Structured Concurrent Programming With Operating Systems Applications*, Addison- Wesley.

HOPPE (1980) J. Hoppe, "A Simple Nucleus Written in Modula-2: A Case Study", *Software - Practice and Experience*, 10 (4), 697-706.

HOPPE (1986) J. Hoppe, "Another Approach to the Implementation of Synchronization Primitives", *Software - Practice and Experience* 16 (12), 1109-1116.

HORTON (1986) I. A. Horton, S. J. Turner, "Using Coroutines in Pascal", *Software - Practice and Experience*, 16 (1), 45-61.

HOYER (1985) W. Hoyer, "Intertask Communication Realized With an Interrupt Mechanism", *Proceedings of the ACM 1985*

Conference, 463-470.

HULL (1986) M. E. C. Hull, "Implementation of The CSP Notation For Concurrent Systems", *The Computer Journal*, 29 (6), 500-505.

HULL (1987) M. E. C. Hull, "OCCAM: a Programming Language For Multiprocessor Systems", *Computer Languages*, 12 (1), 27-37.

IBM (1983) IBM Corporation, *IBM Personal Computer XT Technical Reference Manual*.

INMOS (1984) INMOS Ltd., *Occam Programming Manual*, Prentice-Hall, Englewood Cliffs, New Jersey.

JUSTO (1988) G.R.R. Justo, P.R.F. Cunha, "Modelo de Programação Distribuída", *XXI Congresso Nacional de Informática (SUCESU '88)*, 411-417.

KAUBISCH (1976) W.H. Kaubisch, R.H. Perrot, C.A.R. Hoare, "Quasiparallel Programming", *Software - Practice and Experience* 6 (3), 341-356.

KEARNS (1988). P. Kearns, "A Correct and Unrestrictive Implementation of General Semaphores", *ACM SIGOPS Operating Systems Review* 22 (4), 46-48.

KEEDY (1979) J. L. Keedy, K. Ramamohanarao, J. Rosenberg, "On Implementing Semaphores With Sets", *The Computer Journal*, 22 (2), 146-150.

KEEDY (1985) J.L. Keedy, B. Freisleben, "On The Efficient Use of Semaphore Primitives", *Information Processing Letters* 21 (4), 199-205.

KERNIGHAN (1978) B.W. Kernighan, D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs,

New Jersey.

KESSELS (1977) J.L.W. Kessels, "An Alternative to Event Queues For Synchronization in Monitors", *Comm. ACM* 20 (7) 500-503.

KIRNER (1986) C. Kirner, *Desenvolvimento de Suporte Básico Para Sistemas Operacionais Distribuídos*, Tese D.Sc., COPPE/UFRJ, Rio de Janeiro, Brasil.

KIRNER (1989) C. Kirner, "Sistemas Operacionais Para Ambientes Paralelos", *IX Congresso da SBC, Apostila da VIII Jornada de Atualização em Informática*.

KNOP (1990) F. Knop, "Primitivas de Sincronização Para um Sistema Operacional Altamente Paralelo", *Anais do III Simpósio Brasileiro de Arquitetura de Computadores de Processamento Paralelo*, 282-297.

KOLSTAD (1980) R. B. Kolstad, R. H. Campbell, "Path Pascal User Manual", *ACM/SIGPLAN Notices* 15 (9), 15-25.

KRISHNAMOORTHY (1987) M.S. Krishnamoorthy, S. Agnarsson, "Concurrent Programming in Turbo Pascal", *Byte*, April, 127-133.

KRIZ (1980) J. Kriz, H. Sandmayr, "Extension of Pascal By Coroutines and its Application to Quasi-Parallel Programming and Simulation", *Software - Practice and Experience*, 10, 773-789.

LABRECHE (1990) P. Labreche, L. Lamarche, "Interactors: A Real-time Executive With Multiparty Interactions in C++", *ACM SIGPLAN Notices* 25 (4), 20-32.

LAMPORT (1978) L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System", *Comm. of the ACM* 21 (7), 558-565.

LAMPORT (1985) L. Lamport, "Solved Problems, Unsolved Problems, and Nonproblems in Concurrency", *ACM SIGPOS Operating Systems Review* 19 (4), 33-44.

LAMPSON (1980) B. W. Lampson, D. D. Redell, "Experience With Processes And Monitors in Mesa", *Comm. ACM* 23 (2), 105-117.

LEBLANC (1984) T.J. Leblanc, R.H. Gerber, R.P. Cook, "The StarMod Distributed Programming Kernel", *Software - Practice and Experience*, 14 (12), 1123-1134.

LEBLANC (1987) T.J. Leblanc, J.M. Mellor-Crummey, "Debugging Parallel Programs With Instant Replay", *IEEE Transactions on Computers* C-36 (4), 471-481.

LEDGARD (1983) H. Ledgard, *ADA - an Introduction*, Springer-Verlag New York Inc.

LINDLEY (1987) C.A. Lindley, "Multitasking With Turbo Pascal", *Dr. Dobb's Journal of Software Tools*, July, 42-50.

LISKOV (1979) B. Liskov, "Primitives For Distributed Computing", *Proceedings of the 7th Symposium on Operating Systems Principles*, 33-42.

LISKOV (1988) B. Liskov, L. Shrira, "Promises: Linguistic Support For Efficient Asynchronous Procedure Calls in Distributed Systems", *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design an Implementation* (SIGPLAN Notices 23 (7)), 260-267.

LISTER (1976) A.M. Lister, K.J. Maynard, "An Implementation of Monitors", *Software - Practice and Experience*, 6 (3), 377-385.

MADNICK (1974) S.E. Madnick, J.J. Donovan, *Operating Systems*, McGraw-Hill.

MAEKAWA (1987) M. Maekawa, A.E. Oldehoeft, R.R. Oldehoeft, *Operating Systems - Advanced Concepts*, The Benjamin/Cummings Publishing Company.

MAGALHÃES (1986) M.F. Magalhães, *Software Para Tempo Real*, EBAI.

MANIECKI (1984) M. Maniecki, "Universal Real-Time Kernel", *Microprocessing and Microprogramming* 14 (3-4), 161-163.

MCDOWELL (1989) C. McDowell, D. Helmbold, "Debugging Concurrent Programs", *ACM Computing Surveys* 21(4), 593-622.

MEDEIROS (1981) G.C.R. Medeiros, "O Sistema Pascal Concorrente e sua Implementação no LABO-8034", *Anais do VIII SEMISH, I Congresso da SBC*, 187-198.

MICROSOFT (1986) Microsoft Corporation, *Microsoft MS-DOS Programmer's Reference*.

MILENKOVIC (1987) M. Milenković, *Operating Systems - Concepts and Design*, McGrawHill.

MILLER (1988) B.P. Miller, J. Choi, "A Mechanism For Efficient Debugging of Parallel Programs", *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, 135-144.

MOODY (1980) K. Moody, M. Richards, "A Coroutine Mechanism for BCPL", *Software - Practice and Experience* 10, 765-771.

NIEUWENHUIS (1983) L.J.M. Nieuwenhuis, "MIRTEX: A Real-Time Multitasking Executive For Microprocessors",

Microprocessing and Microprogramming 12 (3-4), 193-198.

NORTON (1985) P. Norton, *The Peter Norton Programmer's Guide to the IBM-PC*, Microsoft Press.

PADILHA (1981) A.C.M. Padilha, S.S. Toscani, "Implementação de Ferramentas Para Programação Concorrente nos Computadores ED300", *Anais do VIII SEMISH, I Congresso da SBC*, 199-212.

PALMER (1983) J.F. Palmer, *The 8087 Primer*, John Wiley & Sons.

PAN (1988) D.Z. Pan, M.A. Linton, "Supporting Reverse Execution of Parallel Programs", *Proceedings of the ACM/SIGPLAN and SIGPOS Workshop on Parallel and Distributed Debugging* (SIGPLAN Notices 24, 1-2), 124-129.

PANCAKE (1990) C.M. Pancake, S. Utter, "A Bibliography of Parallel Debuggers, 1990 Edition", *ACM SIGPLAN Notices* 26 (1), 21-37.

PAULI (1980) W. Pauli, M.L. Soffa, "Coroutine Behavior and Implementation", *Software - Practice and Experience*, 10 (3), 189-204.

PERIHELION (1989) Perihelion Software Ltd, *The Helios Operating System*, Prentice-Hall.

PERROT (1987) R.H. Perrot, *Parallel Programming*, Addison-Wesley.

PETERSON (1985) J.L. Peterson, A. Silberschatz, *Operating System Concepts*, 2nd Edition, Addison-Wesley.

QUEIRÓZ (1985) J.A. Queiróz, "O Kernel Distribuído Constructor", *Anais do XII SEMISH, V Congresso da SBC*, 455-463.

RAJA (1990) D. Raja, K. Shivakumar, S. Rao, "A Portable Real-Time Kernel For 8086/80186/80286/80386 Based Systems on the IBM-PC", *Microprocessing and Microprogramming*, 28 (1-5), 145-150.

RANGEL NETTO (1989) J.L.M. Rangel Netto, "Projeto de Linguagens de Programação", *IX Congresso da SBC, Apostila da VIII Jornada de Atualização em Informática*.

REED (1979) D.P. Reed, R. K. Kanodia, "Synchronization With Eventcounts And Sequencers", *Comm. ACM* 22 (2), 115-123.

REYNOLDS (1990) C. W. Reynolds, "Signalling Regions: Multiprocessing in a Shared-Memory Reconsidered", *Software - Practice and Experience*, 20 (4), 325-356.

RIBAR (1988) L.J. Ribar, "Cooperative Tasking in Modula-2", *Computer Language*, October 5 (10), 63-66.

ROBIE (1988) J. Robie, "Fair Share", *Byte*, July, 229-236.

RUGGIERO (1982) W.V. Ruggiero, G. Bressan, "Um Modelo de Programação para Sistemas Distribuídos", *Anais do Segundo Simpósio Sobre Desenvolvimento de Software Básico Para Micros*, 23-34.

SCANLON (1986) L.J. Scanlon, *Assembly Language Programming for the IBM PC- AT*, Brady Books.

SCHIPER (1989) A. Schiper, R. Simon, Ph. Desarzens, J.A. Sengstag, "Efficient Implementation of Rendez-vous", *The Computer Journal* 32 (3), 267-272.

SEARS (1985) K.H. Sears, A.E. Middleditch, "Software Concurrency in Real Time Control Systems: A Software Nucleus", *Software - Practice and Experience*, 15 (8), 739-759.

SEGRE (1987) L.M. Segre, *Um Estudo de Ambientes de Programação Distribuída: Proposta de Extensões Para Modula-2*, Tese D.Sc. COPPE/UFRJ, Rio de Janeiro, Brasil.

SEGRE (1981a) L. Segre, S. M. Santos, "O Conceito de Monitor Como Instrumento de Sincronização em Programação Concorrente", *Relatório Técnico do Programa de Engenharia de Sistemas de Computação* ES-03-81, COPPE/UFRJ.

SEGRE (1981b) L. Segre, "Mecanismos de Comunicação Para Processos Concorrentes", *Relatório Técnico do Programa de Engenharia de Sistemas de Computação* ES-09-81, COPPE/UFRJ.

SEWRY (1988) D.A. Sewry, "Process Scheduling in Modula-2", *ACM SIGPLAN Notices*, 23 (12), 95-97.

SILBERSCHATZ (1981) A. Silberschatz, "Port Directed Communication", *The Computer Journal*, 24 (1), 78-82.

SMITH (1985) E.T. Smith, "A Debugger For Message-based Processes", *Software - Practice and Experience* 15 (11), 1073-1086.

STONE (1987) H.S. Stone, *High-Performance Computer Architecture*, Addison-Wesley.

STONE (1988) J.M. Stone, "Debugging Concurrent Process: A Case Study", *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, 145-153.

TANEMBAUM (1985) A.S. Tanembaum, R. Van Renesse, "Distributed Operating Systems", *ACM Computing Surveys* 17 (4), 419-470.

TANENBAUM (1987) A.S. Tanenbaum, *Operating Systems: The Design and Implementation*, Prentice-Hall.

TERRY (1986a) P.D. Terry, *Programming Language Translation - A Practical Approach*, Addison-Wesley.

TERRY (1986b) P.D. Terry, "A Modula-2 Kernel For Supporting Monitors", *Software - Practice and Experience*, 16 (5), 457-N472.

TSUJINO (1984) Y. Tsujino, M. Ando, T. Araki, N. Tokura, "Concurrent C: A Programming Language For Distributed Multiprocessor Systems", *Software - Practice and Experience*, 14 (11), 1061-1078.

U.S. Dept. of Defense (1981) United States Department of Defense, "Programming Language ADA: Reference Manual", Vol. 106, *Lecture Notes in Computer Science* - Springer-Verlag, New York.

VASCONCELOS (1988) N.Q. Vasconcelos, E.S.T. Fernandes, "Núcleo Para Programação Concorrente em Pascal", *Relatório Técnico do Programa de Engenharia de Sistemas de Computação* , COPPE/UFRJ, ES-147/88.

VASCONCELOS (1989) N.Q. Vasconcelos, E.S.T. Fernandes, "An Environment For Concurrent Programming in Pascal", *Microprocessing and Microprogramming* 25 (1-5), 381-386.

WALDEN (1972) D.C. Walden, "A System For Interprocess Communication in a Resource Sharing Computer Network", *Comm. ACM* 15 (4), 221-230.

WEBER (1983) J. C. Weber, "Interactive Debugging of Concurrent Programs", *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Software Engineering High Level Debugging* (SIGPLAN Notices 18,8), 112-113.

WEGNER (1983) P. Wegner, S. A. Smolka, "Processes, Tasks and Monitors: A Comparative Study of Concurrent Programming Primitives", *IEEE Trans. on Software Engineering* SE-9 (4), 446-462.

WELSH (1979) J. Welsh, D.W. Bustard, "Pascal-Plus - Another Language For Modular Multiprogramming", *Software - Practice and Experience* 9 (11), 947-957.

WELSH (1981) J. Welsh, A. Lister, "A Comparative Study of Task Communication in Ada", *Software - Practice and Experience*, 11 (3), 257-290.

WETTSTEIN (1977) H. Wettstein, "The Implementation of Synchronizing Operations in Various Environments", *Software - Practice and Experience* 7 (1), 115-126.

WETTSTEIN (1978) H. Wettstein, "The Problem of Nested Monitor Calls Revisited", *ACM SIGOPS Operating System Review*, 12 (1), 19-23.

WILLEN (1983) D.C. Willen, J.I. Krantz, *8088 Assembler Language Programming: The IBM PC*, Howard W. Sams & Co., Inc.

WILLIAMSON (1984) R. Williamson, E. Horowitz, "Concurrent Communications and Synchronization Mechanisms", *Software - Practice and Experience* 14 (2), 135-151.

WIRTH (1977) N. Wirth, "Design and Implementation of Modula", *Software - Practice and Experience* 7 (1), 67-84.

WIRTH (1971) N. Wirth, "The Programming Language Pascal", *Acta Informatica* 1 (1), 35-63.

WRENCH (1988) K.L. Wrench, "CSP-i: An Implementation of Communicating Sequential Processes", *Software - Practice and Experience*, 18 (6), 545-560.

ZEHENDER (1990) E. Zehender, M. Haak, "Recent Improvements on the Concept of Conditional Critical Regions", *Microprocessing and Microprogramming*, 28 (1-5), 15-18.