UMA PROPOSTA PARA ESTENDER O SISTEMA PASCAL
UCSD A PROCESSAMENTO CONCORRENTE

Valmir Carneiro Barbosa

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 CIÊNCIA (M.Sc.)

Aprovada por:

Sueli Mendes dos Santos
(Presidente)

Leslie Afranio Terry

Paulo Mário Bianchi França

Edil Severiano Tavares Fernandes

RIO DE JANEIRO, RJ - BRASIL
MAIO DE 1982

BARBOSA, VALMIR CARNEIRO

Uma Proposta para Estender o Sistema Pascal UCSD a Processamento Concorrente [Rio de Janeiro] 1982.

XVI, 147p., 29,7cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1982).

Tese - Univ. Fed. Rio de Janeiro, Fac. Engenharia.

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

A meus pais
e avó materna

AGRADECIMENTOS

A Leslie Afranio Terry e José Motta da Rocha Lopes pela concepção do projeto e das principais diretrizes de seu desenvolvimento.

A Leslie Afranio Terry, José Motta da Rocha Lopes, Mário Veiga Ferraz Pereira, Lidia Segre e Sérgio Henrique Ferreira da Cunha pela participação nas discussões que definiram o projeto.

A Leslie Afranio Terry e Sueli Mendes dos Santos pelo trabalho de orientação.

A Leslie Afranio Terry, Sueli Mendes dos Santos, Mário Veiga Ferraz Pereira, Lidia Segre e José Motta da Rocha Lopes pela revisão do trabalho.

A Valeria Carneiro Barbosa pelo inestimável auxílio na confecção gráfica da tese; também à Secretaria do Departamento de Sistemas do CEPEL (especialmente a Ruth Maria de Souza) pela ajuda fornecida nesse sentido.

Ao CEPEL por ter possibilitado o desenvolvimento da tese.

SINOPSE

Este trabalho apresenta uma proposta para estender a linguagem Pascal UCSD a programação concorrente. Inicialmente, o Sistema Pascal UCSD é apresentado em sua versão original como sistema mono-usuário para desenvolvimento de programas seqüenciais e é feita uma revisão das principais técnicas de programação concorrente encontradas na literatura.

A extensão proposta sobre o Pascal UCSD baseia-se em uma arquitetura a multiprocessadores fortemente conectados e inclui os seguintes pontos principais:

(i) possibilidade de associar processos concorrentes a rotinas;
(ii) adoção de comandos estruturados para a especificação da concorrência entre processos em tempo de execução;
(iii) adoção de semáforos binários e filas de eventos como instrumentos para a sincronização de processos concorrentes, permitindo tratar problemas de exclusão mútua e sinalização entre processos.

Uma característica importante dessa extensão é a flexibilidade advinda do nível de abstração relativamente baixo das ferramentas de sincronismo incorporadas e da possibilidade de explicitar prioridades em tempo de execução para todas as operações de escalonamento do Sistema.

É também proposto um núcleo que realize as funções do processamento concorrente introduzido na linguagem Pascal UCSD.

ABSTRACT

This work presents a proposal to extend the UCSD Pascal language to concurrent programming. The UCSD Pascal System is initially presented in its original form as a single - user system for the development of sequential programs. In addition the main concurrent programming techniques found in the literature are surveyed.

The extension proposed over the UCSD Pascal is based on a tightly connected multiprocessor architecture, and comprises the following main points:

(i) possibility of associating concurrent processes to routines;

(ii) adoption of structured commands for the runtime specification of the concurrency among processes;

(iii) adoption of binary semaphores and event queues as concurrent process synchronizing mechanisms, allowing to tackle problems of mutual exclusion and signal exchange among processes.

An important feature of the proposed extension is the flexibility resulting from the relatively low level of abstraction of the embedded synchronizing tools, and from the possibility of explicitly assigning priorities during runtime in all scheduling operations of the System.

A kernel to implement the concurrent processing introduced in the UCSD Pascal language is also proposed.

ÍNDICE

ÍNDICE DE FIGURAS

CAPÍTULO II

CAPÍTULO III

CAPÍTULO IV

CAPÍTULO V

APÊNDICE B

ÍNDICE DE ALGORITMOS

CAPÍTULO II

CAPÍTULO III

CAPÍTULO IV

CAPÍTULO V

ÍNDICE DE TABELAS

CAPÍTULO II

CAPÍTULO I

INTRODUÇÃO

Em sistemas de computação é geralmente possível identificar tarefas com um certo grau de independência entre si e que possam, por essa razão, ser executadas com algum paralelismo. O aproveitamento dessas possibilidades de execução simultânea está ligado à eficiência do sistema em questão e à otimização do uso de seus recursos. A palavra recurso engloba todos os entes de que um programa possa precisar para sua execução, ou seja, processadores, memória, dispositivos periféricos, variáveis, etc. Assim, por exemplo, a execução paralela de tarefas relativamente independentes otimiza o aproveitamento dos recursos de processamento do sistema, no sentido em que o bloqueio de uma determinada tarefa libera o processador para a execução de uma outra.

A classe de programas que podem beneficiar-se com a possibilidade de execução paralela é muito ampla e engloba tipicamente Sistemas Operacionais, Sistemas para Simulação e Sistemas para operação em Tempo Real, entre outros. Nesses programas é usual atribuir-se a denominação de processos às diversas tarefas que os compõem, sendo particularmente chamados processos disjuntos ou processos concorrentes, respectivamente nos casos de serem ou não completamente independentes entre si. A execução de processos concorrentes é particularmente problemática, uma vez que eles compartilham recursos do sistema, quer por necessidades de competição ou de cooperação. De qualquer forma, para que possa ser garantida a integridade desses recursos compartilhados e não sejam imprevisíveis os resultados de um processamento concorrente sobre eles, é imprescindível que sua utilização seja disciplinada. Especificamente, apenas um processo deve utilizá-los de cada vez e fazê-lo por meio de operações compatíveis com sua natureza. Os trechos de código através dos quais processos concorrentes manipulam recursos compartilhados são denomi-

nados regiões críticas desses processos sobre os recursos. O problema de sincronizar a execução dessas regiões críticas no tempo é, portanto, de importância fundamental no desenvolvimento de programas concorrentes e é usualmente denominado problema de exclusão mútua entre processos concorrentes.

Os desenvolvimentos mais recentes em processamento concorrente têm-se dirigido no sentido de definir linguagens de alto nível para programação concorrente (ver "discussão" em GOOS[11]). Através dessas linguagens podem ser identificados processos concorrentes como partes de programas, podem ser especificadas as relações de precedência (e de concorrência) entre esses processos, e podem-se programar regiões críticas devidamente sincronizadas entre si. Além das vantagens óbvias de se utilizar linguagens de alto nível, há ainda a possibilidade de realizar testes em tempo de compilação que visem auxiliar na manutenção da integridade dos recursos envolvidos. A realização em tempo de execução das funções programadas em alto nível é então deixada a cargo de um pequeno conjunto de procedimentos, usualmente chamado núcleo, onde é então gerenciada a execução concorrente dos diversos processos.

O objetivo deste trabalho é propor uma extensão do Sistema Pascal UCSD (também conhecido como Sistema P) para processamento concorrente. Esse Sistema foi desenvolvido pela Universidade da Califórnia em San Diego e tem o objetivo de operar como sistema mono-usuário para o desenvolvimento de programas escritos em Pascal (WIRTH [42], JENSEN & WIRTH[28]) em micro/minicomputadores. A principal característica do Sistema P é a existência de uma máquina virtual (a máquina P) sobre a qual ele funciona. Essa máquina virtual é um interpretador de um código intermediário (o código P) resultante da compilação dos programas escritos em Pascal e é responsável pelo alto grau de portabilidade do Sistema : apenas o interpretador precisa ser reescrito para que o Sistema funcione em diferentes ambientes. A escolha desse Sistema como base para uma proposta de programação concorrente foi devida a essa alta portabilidade e ao fato de haver uma grande quantidade de software desenvolvido para ele. Além disso, a linguagem Pas -

cal tem-se tornado bastante popular entre os usuários de sistemas de pequeno e médio porte.

Na extensão do Sistema P para processamento concorrente podem ser identificadas duas linhas principais de trabalho: a adaptação da arquitetura da máquina P e a extensão da linguagem Pascal UCSD. A primeira linha encontra-se descrita em MOTTA[32], onde são investigadas arquiteturas para multiprocessamento. A extensão da linguagem Pascal UCSD para programação concorrente é o tema principal do presente trabalho, que propõe também um núcleo que realize as funções do processamento concorrente. Ao longo dos diversos capítulos são feitas referências a MOTTA[32], em pontos onde se tornam necessários contatos mais íntimos com a arquitetura utilizada.

Este trabalho encontra-se organizado em mais cinco capítulos e quatro apêndices. O capítulo II apresenta as principais características do Sistema P em sua versão original como sistema para programação seqüencial. Nele são descritos a máquina P e o Sistema Operacional, sendo dada especial ênfase às características relevantes à extensão para processamento concorrente. No capítulo III são apresentadas em uma ordem aproximadamente histórica as principais técnicas já desenvolvidas para a sincronização de processos concorrentes e para a especificação da concorrência entre eles. O principal objetivo desse capítulo é fornecer uma visão geral do desenvolvimento das técnicas de programação concorrente, com vistas a apoiar uma escolha para a extensão do Sistema P. O capítulo IV apresenta algumas arquiteturas a multiprocessadores virtuais adequadas ao processamento concorrente pretendido para o Sistema P e propõe extensões ao Pascal UCSD. Um núcleo que realize essas extensões é descrito no capítulo V, que utiliza a linguagem Pascal para especificação dos algoritmos. O capítulo VI fornece uma breve discussão de alguns pontos especiais. Ao final da documentação encontram-se quatro apêndices (A, B, C e D) que fornecem respectivamente informações sobre alguns procedimentos intrínsecos exclusivos ao Pascal UCSD, exemplos de aplicação dos métodos de sincronização dis

cutidos no capítulo III, um resumo das extensões propostas sobre o Pascal UCSD, e descrição de algumas extensões do interpretador necessárias para dar suporte a processamento concorrente.

CAPÍTULO II

O SISTEMA P

II.1- INTRODUÇÃO

Este capítulo destina-se a dar uma visão geral do Sistema P em sua versão I.5. O material apresentado fornece as informações necessárias à compreensão dos próximos capítulos, e portanto é dada ênfase às características de alguma forma relevantes à sua extensão para desenvolvimento de programas concorrentes. Uma descrição detalhada de toda essa versão do Sistema P e de seu uso pode ser encontrada em SHILLINGTON & ACKLAND[36] e em BOWLES[3]. Ao longo do texto serão encontradas referências a procedimentos intrínsecos da linguagem Pascal que não fazem parte de sua definição original (JENSEN & WIRTH[28]), sendo recomendado que se consulte o apêndice A desta documentação, onde são descritas essas particularidades.

O Sistema P foi desenvolvido pelo Instituto de Sistemas de Informação do campus de San Diego da Universidade da Califórnia. Seu projeto foi orientado de modo a que pudesse funcionar em micro/minicomputadores como sistema mono-usuário para o desenvolvimento de programas seqüenciais em Pascal. Com o objetivo de reduzir a dependência do Sistema sobre o hardware em que funciona, foi criada uma máquina virtual, a máquina P , que executa um código intermediário, o código P. O resultado é um sistema escrito quase todo em Pascal, com alto grau de portabilidade dado pela máquina P. Apenas o interpretador do código P é escrito na linguagem da máquina real, constituindo a única parte do Sistema que deve ser reescrita para adaptá-lo ao funcionamento em outro ambiente.

A figura II.1 ilustra a organização hierárquica do Sistema P. No nível mais baixo encontra-se o processador real, cuja atribuição é executar o interpretador; para ele o código P

é um conjunto de dados que devem ser manipulados de acordo com os algoritmos de interpretação. O nível intermediário é o próprio interpretador, que simula uma máquina virtual através da execução do código P existente em seu espaço de endereçamento. O Sistema Operacional, que engloba todos os programas escritos em Pascal, ocupa o nível hierarquicamente mais alto.

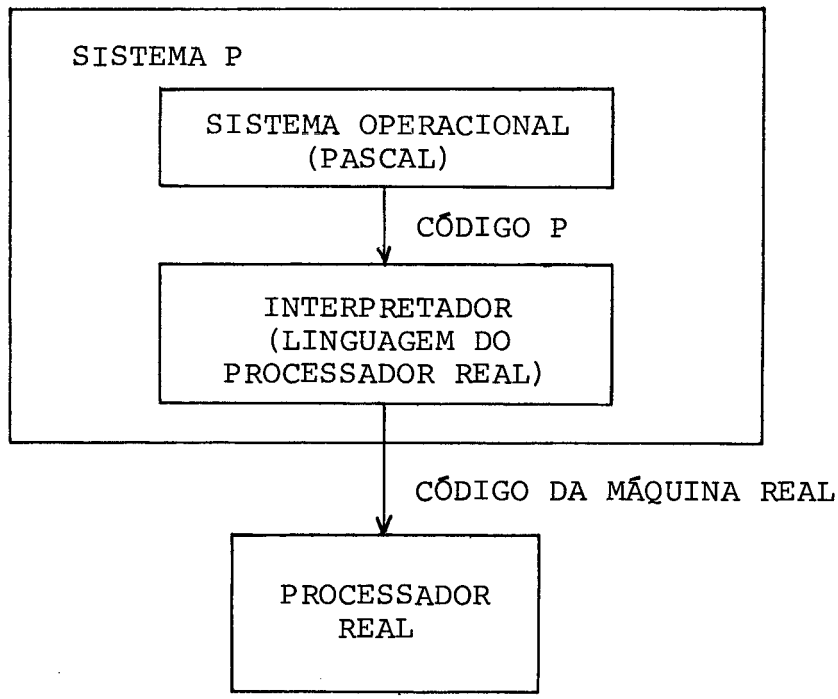


FIGURA II.1 - Organização Hierárquica do Sistema P

As próximas seções deste capítulo são dedicados à apresentação do Sistema P em sua estrutura e funcionamento. A seção II.2 é dedicada à técnica de segmentação adotada no projeto do Sistema P. Na seção II.3 é apresentada a arquitetura da máquina P, incluindo interpretador, memória e unidades de entrada e saída de dados (uma descrição mais detalhada pode ser encontrada em MOTTA [32]). O Sistema Operacional é descrito na seção II.4, onde são apresentados seus principais algoritmos e estruturas de dados.

II.2- SEGMENTAÇÃO NO SISTEMA P

O Sistema P foi concebido para funcionar em computadores de pequeno e médio porte (micro/minicomputadores) e, por essa razão, o espaço de endereçamento de seu processador virtual foi limitado em um número relativamente pequeno de palavras (32K palavras de 16 bits, ver item II.3.2). Aliada a esse fato existe a restrição usual de quantidade de memória disponível na classe dos micro/minicomputadores. Por exemplo, microprocessadores 8085 e minicomputadores PDP-11, para os quais existem implementações do Sistema P, endereçam apenas 64K bytes de memória, parte dos quais é utilizada pelo interpretador, reduzindo ainda mais seu espaço de endereçamento.

Com o objetivo de otimizar a utilização do espaço de endereçamento do processador virtual, o Sistema P incorpora o conceito de segmentação, que possibilita a execução de um programa sem que todo o código P esteja na memória, sendo carregado quando necessário.

O controle dessa segmentação é feito no próprio texto do programa em Pascal, através da utilização de uma classe especial de rotinas, chamadas segment procedure e segment function. Essas funcionam da mesma forma que as procedures e functions do Pascal standard, exceto quanto à criação, pelo compilador, de um segmento de código separado para cada uma delas, sendo estes trazidos para a memória da máquina P quando da chamada das rotinas respectivas.

A compilação de qualquer programa Pascal no Sistema P gera um arquivo de código que contém pelo menos um segmento de código (correspondente ao programa principal, exceto em casos de compilação separada) e um dicionário de segmentos, que identifica e localiza cada segmento de código dentro do arquivo. A figura II.2 ilustra a compilação do programa EXEMPLO, com a geração de um arquivo com cinco segmentos de código.

Cada segmento de código ocupa espaço na memória da má

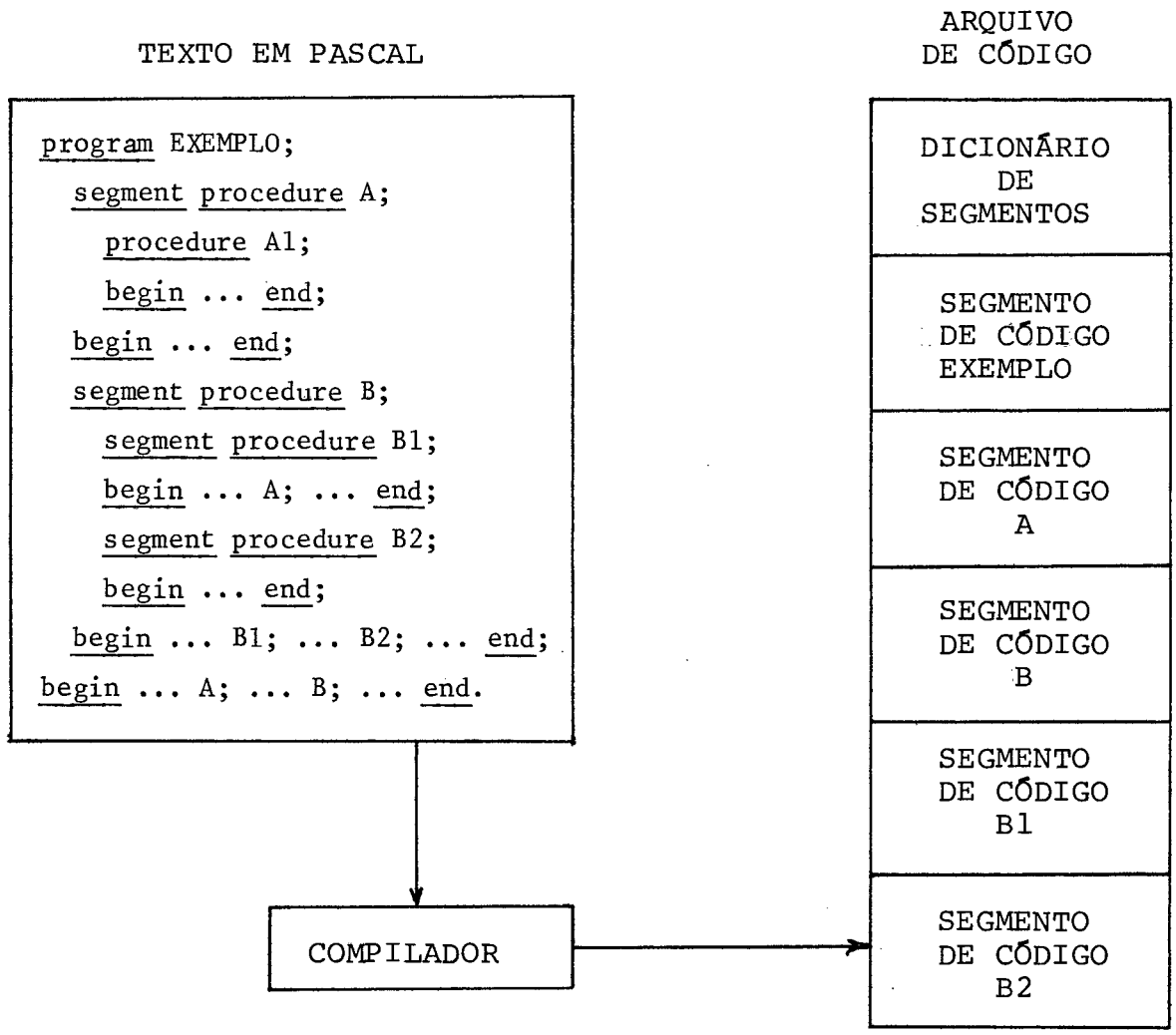
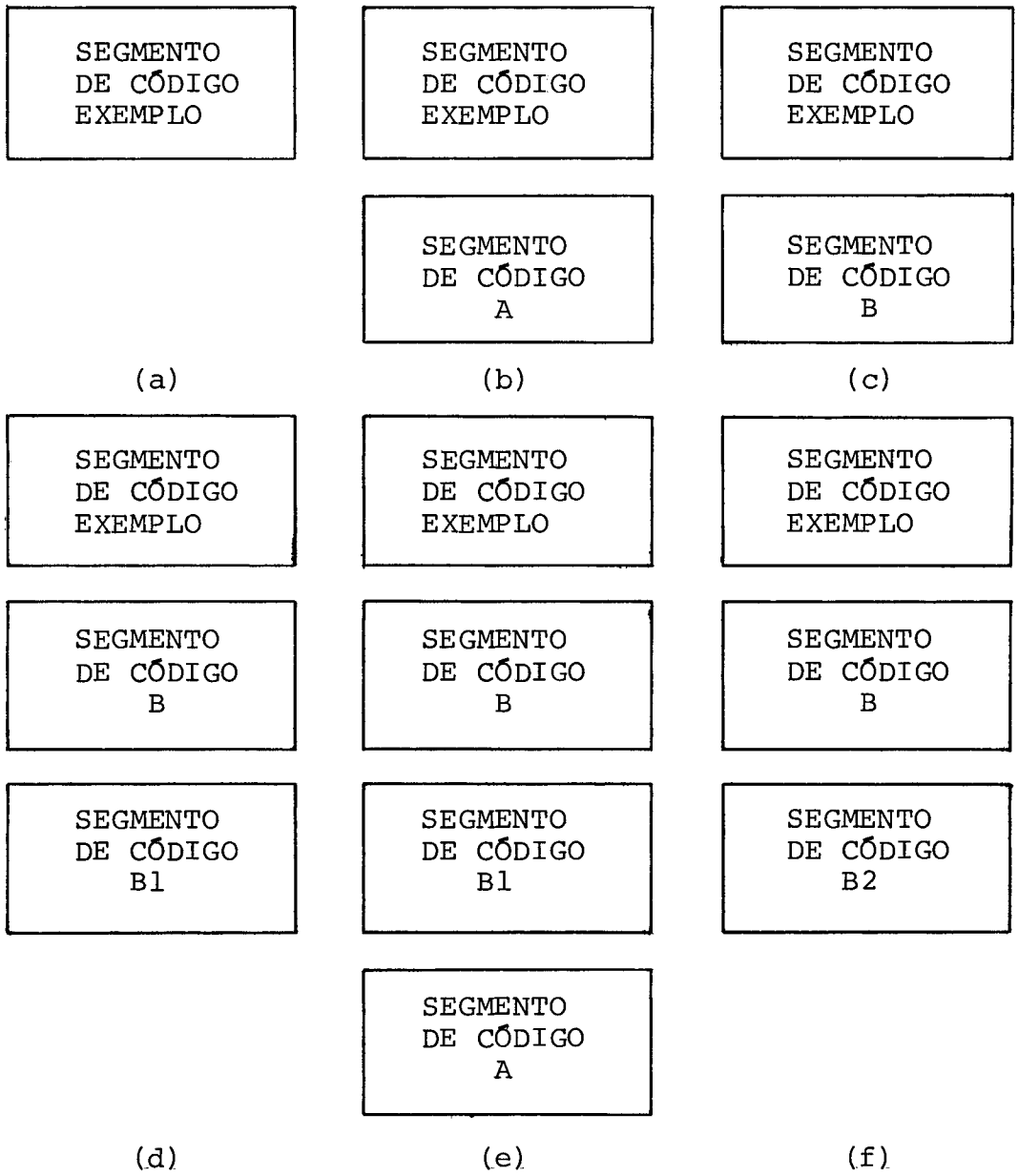


FIGURA II.2 - Geração de Segmentos de Código pelo Compilador

quina P somente durante a execução da segment procedure ou segment function correspondente. Pelo fato de o arquivo de código estar armazenado em dispositivos de memória de massa (geralmente discos), o projeto de programas segmentados deve evitar chamadas não-recursivas muito freqüentes a rotinas do tipo segment, uma vez que a cada uma dessas chamadas corresponde uma carga do segmento de código respectivo.

A segmentação de programas em Pascal no Sistema P tem um efeito semelhante ao do "overlay" utilizado em certos computadores, que é o de dois ou mais trechos de código compartilharem a mesma área de memória. Na figura II.3 está re-

presentada a ocupação da memória da máquina P durante a execução do programa EXEMPLO da figura II.2. Observe-se que o seg - mento de código correspondente ao programa principal está sempre residente na memória, e que segmentos chamados de um mesmo segmento compartilham a mesma área de memória.



(a) início da execução de EXEMPLO
(b) chamada de A em EXEMPLO
(c) chamada de B em EXEMPLO
(d) chamada de B1 em B
(e) chamada de A em B1
(f) chamada de B2 em B

FIGURA II.3 - Realização de "overlay" na Máquina P

II.3- A ARQUITETURA DA MÁQUINA P

II.3.1- ORGANIZAÇÃO GERAL

A máquina P possui um processador virtual que manipula palavras de 16 bits organizadas em pilhas, sendo cada palavra composta por 2 bytes de 8 bits. A figura II.4 apresenta a organização geral da máquina P. Nela podem ser vistos o interpretador, a memória e unidades para entrada e saída de dados.

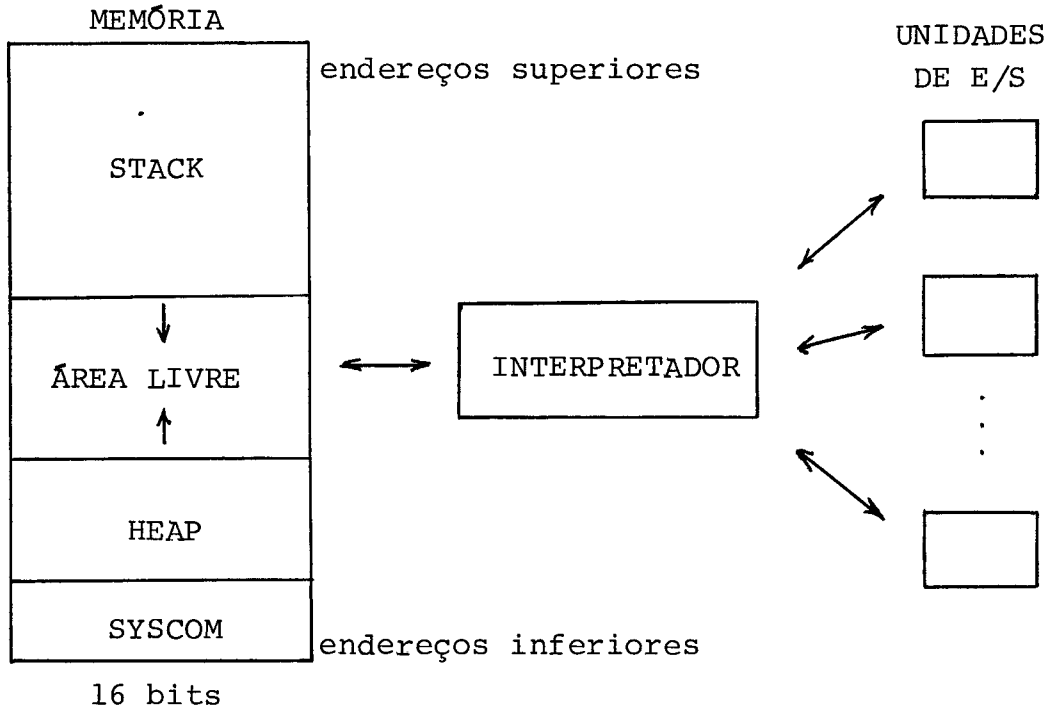


FIGURA II.4 - Organização Geral da Máquina P

A memória da máquina P está dividida em duas pilhas que crescem em sentidos contrários: o stack (item II.3.3), que contém segmentos de código e dados de alocação estática, e o heap (item II.3.4), que contém dados de alocação dinâmica. Na base do heap existe uma área de dados chamada SYSCOM, a Área de Comunicação do Sistema, que contém dados comuns ao interpre

tador e ao Sistema Operacional.

II.3.2- O INTERPRETADOR

O Interpretador é a Unidade Central de Processamento da máquina P, sendo encarregado da execução do código P. Para tanto, utiliza o conjunto de registradores internos descrito na tabela II.1. O significado desses registradores encontra-se esclarecido nos próximos itens da seção II.3. Todos os registradores são de 16 bits e alinhados por palavras, com exceção de IPC, alinhado por bytes (o que limita em 64K bytes a memória da máquina P).

SÍMBOLO	NOME	DESCRIÇÃO
SP	Stack Pointer	Aponta para o topo do stack
NP	New Pointer	Aponta para o topo do heap
JTAB	Jump TABle pointer	Aponta para a tabela de atributos da rotina em execução
SEG	SEGment pointer	Aponta para o dicionário de rotinas do segmento ao qual pertence a rotina em execução
MP	Most recent Procedure	Aponta para o registro de ativação da rotina em execução
BASE	BASE procedure	Aponta para o registro de ativação da rotina de nível léxico 0 de ativação mais recente
IPC	Interpreter Program Counter	Aponta para o próximo código P a ser executado

TABELA II.1 - Registradores da Máquina P

A função do interpretador pode ser resumida segundo o algoritmo II.1. As diversas técnicas existentes para a interpretação de códigos intermediários encontram-se descritas em MOTTA [32].

```
begin
  IPC := endereço inicial de execução;
  while true do
    begin
      PCODE := byte endereçado por IPC;
      IPC := IPC + 1;
      interpreta PCODE
    end
end;
```

ALGORITMO II.1 - Funcionamento do Interpretador de Código P

Os códigos P executados pelo interpretador estão divididos nas seguintes classes:

- Busca, indexação, armazenamento e transferência de variáveis;
- Aritmética e comparações de topo de stack;
- Desvios;
- Chamadas e retornos de rotinas;
- Rotinas de suporte dos programas do Sistema.

Alguns desses códigos acarretam o crescimento do stack (carga no topo do stack e chamadas de rotinas), outros causam seu decrescimento (retirada do topo do stack, retorno de rotinas e operações aritméticas).

Existem dois códigos P de importância especial para este trabalho. São os códigos de chamada a rotina externa (CXP) e de chamada a rotina standard (CSP). É considerada rotina externa qualquer rotina cujo código se encontre em um segmento

de código diferente daquele de que a chamada se originou. Sua identificação é feita pelo número do segmento e pelo número da rotina dentro do segmento (o corpo do segmento é considerado sua rotina de número 1). Rotinas standard são parte do próprio interpretador, sendo em sua maioria procedimentos de controle em tempo de execução, ou que requerem uma certa proximidade ao hardware (como operações básicas de entrada e saída). São identificadas por um único número.

II.3.3- O STACK

O stack da máquina P é a região de sua memória destinada à execução do código P, ao armazenamento de variáveis de alocação estática, e ao armazenamento temporário de dados utilizados como operandos na avaliação de expressões. O topo do stack é apontado pelo registrador SP, como mostrado na figura II.5(a).

A figura II.5(a) mostra o stack da máquina P durante a execução de uma determinada rotina. O segmento de código contém, entre outros, o código da rotina em execução. Quando ocorre a chamada da rotina, o interpretador compõe um segmento de dados, sobre o qual se desenvolve dinamicamente uma área (a pilha de avaliação) durante a execução da rotina.

O segmento de código e o segmento de dados aparecem um pouco mais detalhados na figura II.5(b), na qual estão indicadas as funções de alguns dos registradores da máquina P. O segmento de código compõe-se de seções de código (uma para cada rotina), identificadas por um dicionário de rotinas. Esse dicionário é apontado pelo registrador SEG, e os registradores JTAB e IPC apontam para a seção de código da rotina em execução. O segmento de dados é composto por uma área de parâmetros e variáveis locais (com dados de alocação estática), e por um registro de ativação para o qual aponta o registrador MP. O registrador BASE aponta para o registro de ativação da rotina de nível léxico 0 (em cujo segmento de dados encontram-se as variáveis globais), coincidindo com MP quando ela está

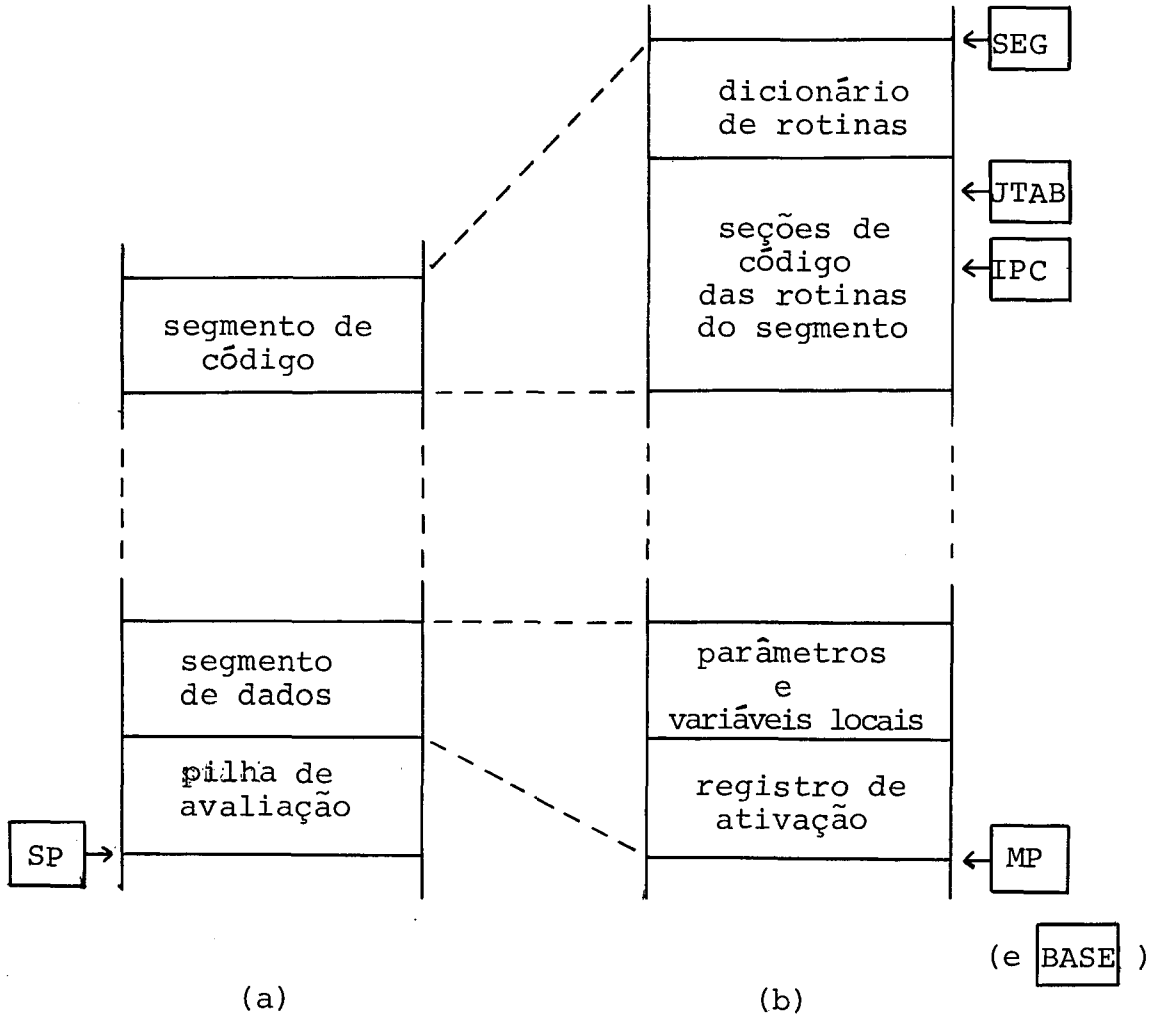


FIGURA II.5 - O Stack da Máquina P

em execução.

A finalidade do registro de ativação é armazenar o estado do processador virtual quando da chamada da rotina. Nele são guardadas todas as informações que permitirão restaurar o ambiente do processador para continuar a execução da rotina que gerou a chamada. Além disso, o registro de ativação contém os chamados link dinâmico e link estático. O link dinâmico é o endereço do registro de ativação da rotina que originou a chamada da que se encontra em execução. É utilizado para restaurar o valor de MP na hora do retorno. O link estático aponta para o registro de ativação da rotina que envolve a rotina em execução e que tem nível léxico imediatamente inferior ao des-

ta. Os links estáticos formam uma cadeia que permite acessar todas as variáveis globais à rotina em execução. Mais detalhes sobre esses links podem ser encontrados em BARBOSA & VALLE [1].

II.3.4- O HEAP

A manipulação de variáveis de alocação dinâmica no Sistema P é feita sobre o heap através das rotinas standard new, mark e release, que atuam como descrito na tabela II.2 . A figura II.6 ilustra a utilização desses recursos na execução do programa TESTAHEAP (algoritmo II.2). Deve ser observado que a operação release elimina todos os itens alocados no heap depois da operação mark, independentemente do escopo dos apontadores envolvidos.

```
program TESTAHEAP;
type estrutura = record ... end;

procedure A;
var p2: ↑estrutura;
begin new(p2) end;

procedure B;
var p1: ↑estrutura; markptr: ↑integer;
begin
  mark(markptr);
  new(p1);
  A;
  release(markptr)
end;

begin B end.
```

ALGORITMO II.2 - Exemplo de Alocação Dinâmica de Variáveis

new(p)	p := NP; NP := NP + tamanho de p↑
mark(p)	p := NP
release(p)	NP := p

TABELA II.2 - Operações para Alocar e Remover Variáveis Dinamicamente

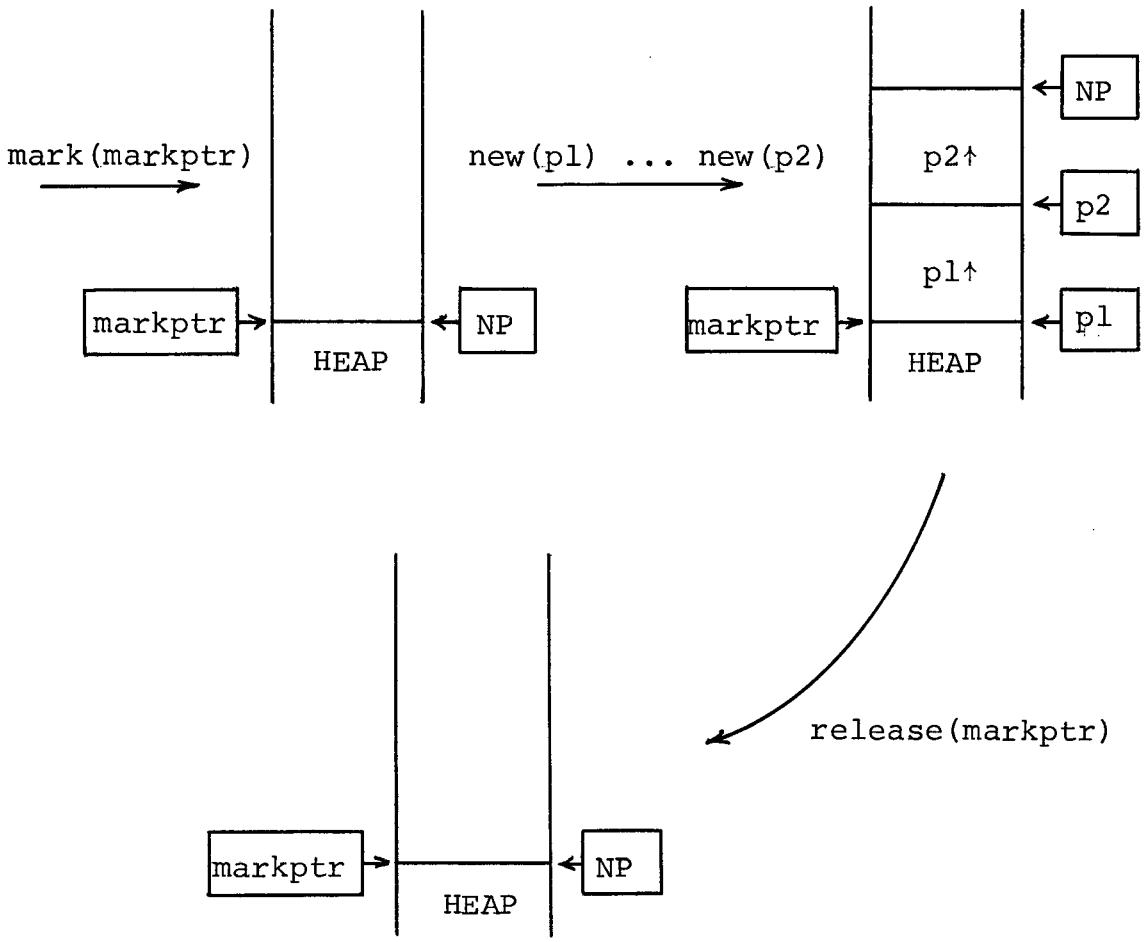


FIGURA II.6 - Exemplo de Alocação Dinâmica de Variáveis

II.3.5- UNIDADES DE ENTRADA E SAÍDA DE DADOS

A máquina P possui um conjunto de unidades virtuais para entrada e saída de dados, mapeadas em dispositivos periféricos reais pelo interpretador. Essas unidades estão divididas em dois grupos: unidades blocadas (mapeadas em discos ou dispositivos semelhantes) e unidades não-blocadas (mapeadas em terminais de vídeo, impressoras, etc.).

As unidades blocadas são tratadas como uma seqüência de blocos de tamanho (número de bytes) fixo, como mostrado na figura II.7. O número de blocos de cada unidade depende do dispositivo físico particular ao qual ela é associada.

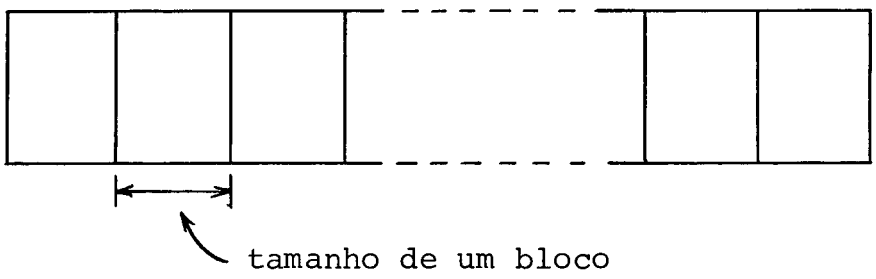


FIGURA II.7 - Unidade Blocada para E/S

As unidades não-blocadas têm no Sistema P um significado mais abstrato, sendo vistas simplesmente como fonte ou destino de uma seqüência de caracteres.

As unidades de e/s são manipuladas por duas rotinas standard do interpretador, unitread para a entrada de dados e unitwrite para saída. Esses são os procedimentos de mais baixo nível para entrada e saída de dados no Sistema P e permitem a entrada ou saída de um número qualquer de bytes de maneira síncrona ou assíncrona. Se a operação for síncrona, o programa que a solicitou ficará bloqueado até que ela termine. Caso contrário (operação assíncrona), o programa será imediatamente reativado, e o término da operação requerida poderá

ser verificado mais tarde através das rotinas standard unitwait e unitbusy. Após a realização de uma operação qualquer de entrada ou saída de dados, a rotina standard ioresult fornecerá um valor inteiro correspondente ao sucesso com que a operação foi completada. Esse valor encontra-se armazenado em um campo da SYSCOM (item II.3.1).

II.4- O SISTEMA OPERACIONAL

II.4.1- OS NÍVEIS DE COMANDOS

O Sistema P utiliza em geral um terminal de vídeo para interação com o usuário. Através desse terminal o usuário emite comandos ao Sistema Operacional e dele recebe respostas. Os comandos existentes estão dispostos em níveis que podem ser organizados em uma árvore de comandos, conforme representado na figura II.8. A cada nodo dessa árvore está associada uma lista de comandos e esta lista ocupa a primeira linha da tela quando a interação entre o usuário e o Sistema Operacional encontra-se na fase correspondente. Por exemplo, quando a interação se encontra no nível mais externo (correspondente à raiz da árvore), a lista de comandos é

E(dit, R(un, F(ile, C(omp, L(ink, X(ecute, A(ssem, D(ebug

onde as letras maiúsculas indicam as chaves para passar a um outro nível de comandos (no caso da raiz, necessariamente a um nível mais interno).

II.4.2- O DESENVOLVIMENTO DE PROGRAMAS NO SISTEMA P

Um conceito central ao projeto da estrutura do Sistema P foi o de Arquivo de Trabalho ("workfile"). Um Arquivo de Trabalho pode ser visto como uma área de rascunho utilizada para armazenar programas temporariamente durante seu desenvolvimento. No Sistema P, o Arquivo de Trabalho é constituído por duas partes:

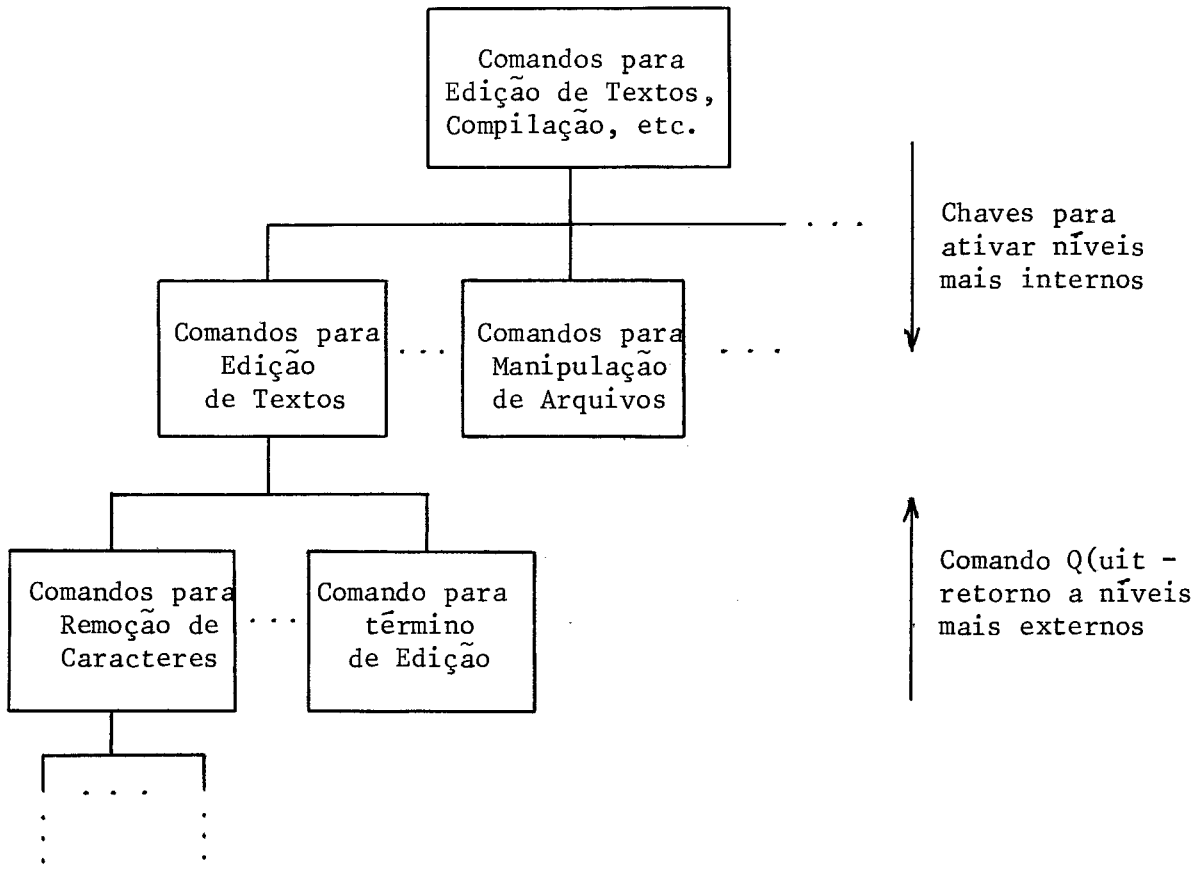


FIGURA II.8 - Hierarquia dos Comandos no Sistema Operacional

- Arquivo fonte, que contém o texto de um programa escrito geralmente em Pascal ou em linguagem de máquina;
- Arquivo de código, que contém o código P resultante da compilação do texto em alto nível (geralmente em Pascal) ou o código do processador real, resultante da montagem do texto em linguagem de máquina.

Não pode haver mais de um Arquivo de Trabalho presente no Sistema P ao mesmo tempo. O único Arquivo de Trabalho existente é então utilizado como objeto "default" das operações de edição, compilação (ou montagem), "linkedição" e execução.

As operações de criação, remoção, etc., do Arquivo de Trabalho são realizadas através do Gerente de Arquivos do Sis-

tema P ("File Manager" ou "Filer"), que é ativado através da chave F (item II.4.1). A seleção dessa chave substitui a linha de comandos por

```
>Filer: G(et, S(ave, W(hat, N(ew, L(dir, R(em, C(hng, T(rans, D(ate, Q(uit
```

onde são apresentados alguns dos comandos disponíveis para a manipulação de arquivos. Os quatro primeiros comandos (de chaves G, S, W e N) são específicos para a manipulação do Arquivo de Trabalho. Os demais comandos podem ser utilizados no tratamento de quaisquer outros arquivos e realizam as funções gerais de remoção, transferência, etc.

A parte de texto do Arquivo de Trabalho é manipulada através do Editor de Textos, ativado pela chave E (item II.4.1). Os comandos disponíveis são:

```
>Edit: A(djust C(py D(lete F(ind I(nsrt J(mp R(place Q(uit X(chng Z(ap
```

O Editor de Textos do Sistema P é do tipo conhecido como "orientado para tela" ("screen-oriented"), o que significa que a tela do terminal de vídeo utilizado funciona como uma janela que permite observar uma certa porção do texto que se está editando (figura II.9).

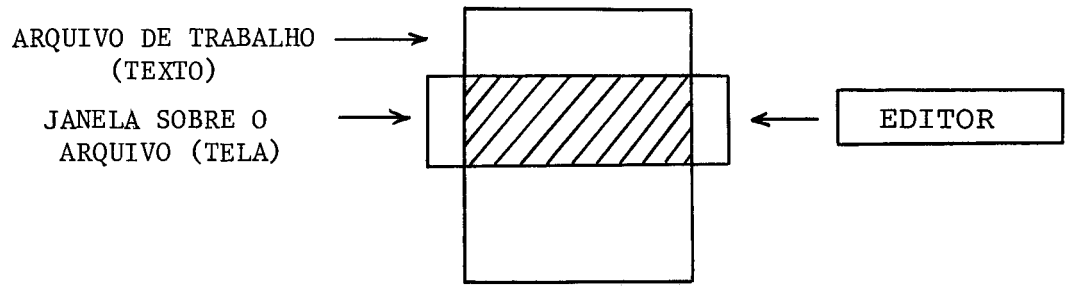


FIGURA II.9 - Operação do Editor de Textos

A etapa de compilação ou montagem, quando não detetados erros de sintaxe, gera a parte de código do Arquivo de Trabalho (figura II.10). O Compilador é ativado através da chave C e o Montador através da chave A (item II.4.1). O compilador

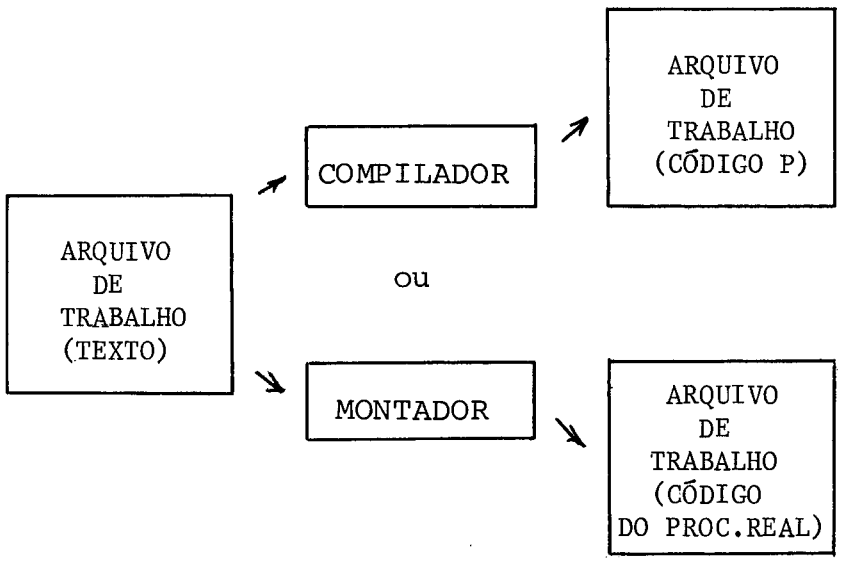


FIGURA II.10 - Operação do Compilador e do Montador

ativado dessa maneira é um compilador Pascal e gera código P, executável pelo processador virtual. Esse compilador fornece ao usuário a possibilidade de compilar separadamente trechos de seu programa. O montador ativado pela chave A é o chamado "montador genérico" do Sistema P. Esse montador possui uma parte que depende do processador real e que deve ser reescrita a cada implementação do Sistema P, a fim de poder gerar código para esse processador. O Compilador e o Montador podem, opcionalmente, gerar um arquivo com a listagem resultante da compilação ou montagem.

Se o código gerado não possui referências externas e é um programa completo, então ele pode ser transferido para um outro arquivo e executado (através da chave X, item II.4.1). No caso de haver referências externas ou não se tratar de um programa completo, deve então ser realizada a operação de "linkedição", ativada pela chave L (item II.4.1). O "Linkeditor" assim ativado realiza a composição de um arquivo de código executável a partir de outros arquivos de código. Um desses pode ser a parte de código do Arquivo de Trabalho, como mostrado na figura II.11. Observe-se que arquivos de código P podem

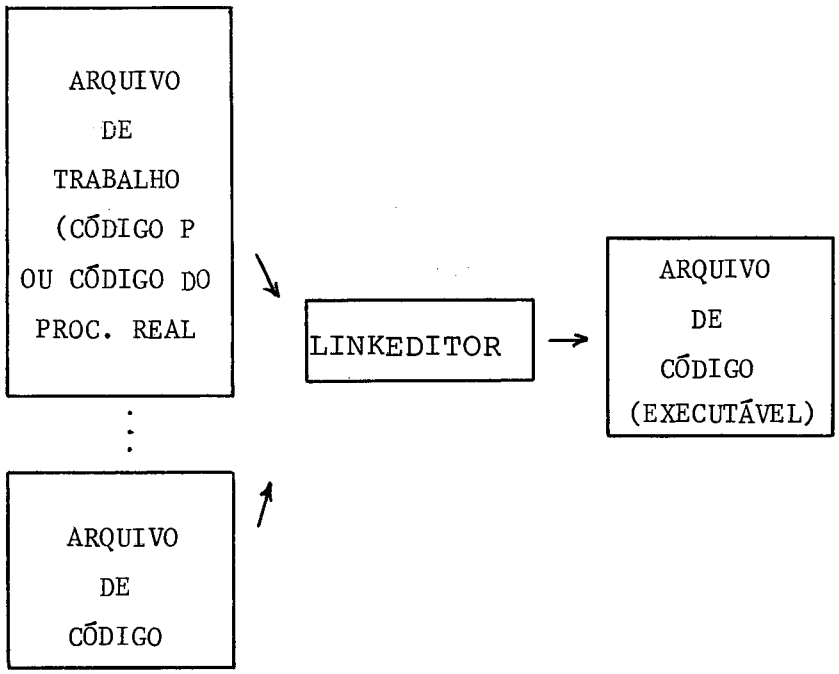


FIGURA II.11 - Operação do Linkeditor

ser ligados a arquivos de código do processador real na composição do código final executável. Esse arquivo pode então ser executado através da chave X já mencionada.

No nível da raiz da árvore de comandos (figura II.8) existe também o comando R(un, ativado pela chave R, que realiza a seqüência de operações compreendida pela compilação, "linkedição" e execução, sem intervenção do usuário.

O Sistema P fornece ainda ao usuário uma série de programas utilitários, ativados pela chave X. Esses programas compreendem, entre outros, um Editor de Textos orientado para linhas, um Gerente de Bibliotecas e um compilador para código P da linguagem BASIC.

II.4.3- ALGORITMO BÁSICO

O Sistema Operacional pode ser visto como um grande programa escrito em Pascal, do qual o Compilador, o Editor de Textos, o Gerente de Arquivos, etc., são rotinas do tipo segment, como descritas na seção II.2, ativadas de acordo com o andamento da interação com o usuário. Todavia, os utilitários mencionados acima são programas independentes e o algoritmo II.3 fornece uma representação realista do Sistema Operacional. Segundo esse algoritmo, a execução de qualquer função so

```
program PASCALSYSTEM;
...
const maxseg = ... ;
...
type syscomrec = record
                   ...
                   SEGTABLE: array[0..maxseg] of ... ;
                   ...
                 end;
...
var SYSCOM: ↑syscomrec;
    ch: char;
...
  segment procedure USERPROGRAM;  (* segmento nº 1 *)
  begin ... end;
...
(* código do segmento principal (nº 0) de PASCALSYSTEM;
   inclui alguns procedimentos intrínsecos do Pascal UCSD e
   alguns procedimentos para controle em tempo de execução  *)
...
begin read(ch);
  while ch <> 'H' (* halt *) do
    begin
      case ch of
        'C': busca o código do Compilador em uma
             unidade blocada de e/s;
        ...
      end;
      altera as posições 1 e 7 a maxseg de SYSCOM↑.SEGTABLE;
      USERPROGRAM;
      read(ch)
    end
end.
```

ALGORITMO II.3 - Algoritmo Básico do Sistema Operacional

licitada pelo usuário é realizada através de uma chamada à rotina USERPROGRAM, do tipo segment. Essa chamada, traduzida em código P, é uma chamada à primeira rotina externa do segmento de nº 1 (ver item II.3.2, chamadas a rotinas externas). Como o interpretador executa essas chamadas através da consulta à tabela SYSCOM↑.SEGTABLE, então essa tabela deve ser modificada a cada solicitação do usuário. A figura II.12 ilustra a composição de SYSCOM↑.SEGTABLE a partir de dicionários de segmentos de código. Essa tabela possui maxseg posições (máximo de segmentos permitidos no Sistema), sendo que as de números 0 e 2 a 6 são reservadas aos segmentos de PASCALSYSTEM, preenchidas assim que o Sistema é ativado, a partir do dicionário de segmentos gerado em sua compilação. As demais entradas de SYSCOM↑.SEGTABLE, de números 1 e 7 a maxseg, são preenchidas de acordo com a interação com o usuário, também a partir de um dicionário de segmentos. Dessa forma, o segmento de número 0 corresponde sempre ao segmento principal de PASCALSYSTEM e o de número 1 é sempre relativo ao segmento principal do programa que o usuário deseja executar.

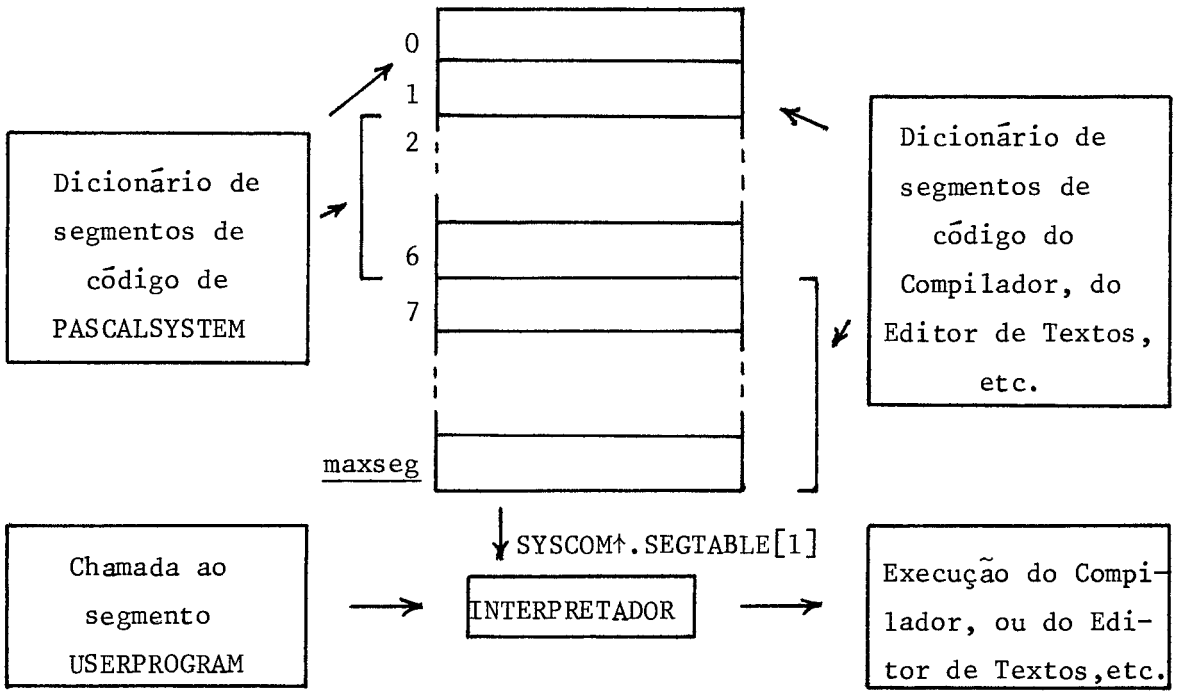


FIGURA II.12 - Formação da Tabela de Segmentos para Execução das Funções do Sistema Operacional

II.4.4- PROCEDIMENTOS INTRÍNSECOS DO PASCAL UCSD

Conforme foi dito no item II.3.2, os códigos P de chamada a rotinas externas (CXP) e de chamada a rotinas standard (CSP) foram de grande importância no projeto do Sistema P e o serão em sua extensão ao desenvolvimento de programas concorrentes. É através deles que os procedimentos intrínse - cos da linguagem podem ser acessados por qualquer programa.

Além das operações intrínsecas à linguagem Pascal definidas em JENSEN & WIRTH[28] (como reset, put, succ, read, etc.), o Pascal do Sistema P incorpora uma série de outras (para manipulação de strings, operação das unidades de entrada e saída, etc.).

Algumas dessas operações são implementadas em linguagem de máquina (no interpretador), por questões de eficiência ou por necessidade de contato com o hardware. Estas pertencem à classe de rotinas standard e são chamadas através do código CSP.

As operações que não necessitam ser implementadas na linguagem do processador real são executadas de forma inter - pretada a partir do código P. São escritas em Pascal e fazem parte do segmento principal de PASCALSYSTEM (algoritmo II.3), sempre residente na memória da máquina P. Esse segmento ocupa a posição de número 0 em SYSCOM↑.SEGTABLE (item II.4.3) e dentro dele as rotinas intrínsecas encontram-se definidas em uma ordem fixa. O Compilador pode, portanto, gerar chamadas do tipo CXP a uma determinada rotina do segmento 0 sempre que necessário.

II.4.5- PROCEDIMENTOS DE CONTROLE EM TEMPO DE EXECUÇÃO

O conjunto de rotinas standard e o conjunto de rotinas do segmento principal de PASCALSYSTEM (algoritmo II.3) não são restritos à implementação de procedimentos intrínsecos da linguagem (item II.4.4), mas também incluem vários procedimen

tos necessários ao controle da execução de programas. Esses procedimentos incluem tipicamente a inicialização de algumas estruturas de dados especiais, a verificação de ocorrência de erros após uma operação de entrada ou saída de dados, o con - trole dos limites de arrays e sets durante o processo de inde xação, etc.

No Sistema P essas funções são implementadas através de rotinas standard e rotinas do segmento principal de PASCAL SYSTEM, tratadas como externas durante a compilação de outros programas. Algumas operações (como o controle de erros em pro cedimentos de e/s) são implementadas como rotinas standard e outras (como a inicialização automática de variáveis do tipo file) implementadas como rotinas externas. O Compilador se en carrega de gerar chamadas a essas rotinas de maneira transparente ao programador.

II.4.6- ORGANIZAÇÃO DAS UNIDADES BLOCADAS DE E/S

As unidades blocadas de entrada e saída de dados do Sistema P são mapeadas pelo interpretador em discos ou dispositivos similares. Cada uma dessas unidades é caracterizada pelas constantes fblksize e ueovblk, que indicam respectiva - mente o tamanho de cada bloco ("file block size") em bytes e o último bloco da unidade ("unit end-of-volume block"). A cons tante fblksize é uma só para todas as unidades e o valor usado na versão I.5 do Sistema P é 512. Os blocos de cada unidade são numerados de 0 a ueovblk, sendo os blocos de números 0 e 1 reservados para o "bootstrap" do Sistema. Os próximos qua tro blocos (de números 2 a 5) são utilizados para armazenar o diretório da unidade, sendo que opcionalmente os blocos de nú meros 6 a 9 podem conter uma cópia desse diretório (figura II.13). Os demais blocos, numerados de 6 (ou de 10) a ueovblk são ocupados por arquivos, sendo que cada arquivo ocupa um nú mero qualquer de blocos contíguos.

O diretório de arquivos de uma unidade blocada pode ser definido em Pascal pela seguinte estrutura:

```
const maxdir = 77;

type dirrange = 0..maxdir;
     direntry = record
                  dfirstblk,
                  dlastblk: 0..ueovblk;
                  ...
                end;
     directory = array[dirrange] of direntry;
     dirp = ↑directory;
```

Essa estrutura representa o diretório como um array com maxdir + 1 posições. Cada posição (direntry) é destinada à identificação de um arquivo, informando, entre outras coisas, os números do primeiro e do último bloco (respectivamente dfirstblk e dlastblk) ocupados por ele na unidade. As demais informações de cada posição são relativas ao nome do arquivo, ao número de bytes ocupados no último bloco e à data em que foi realizada a última modificação. A primeira posição (directory[0]) é reservada à descrição da unidade, informando seu nome, número de blocos, número de arquivos, etc.

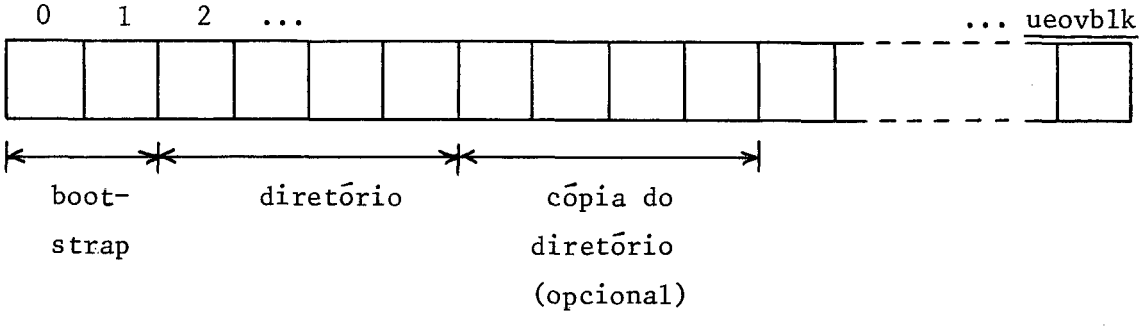


FIGURA II.13 - Organização das Unidades Blocadas de E/S

O Sistema Operacional manipula os diretórios das diversas unidades blocadas através da variável

```
gdirp: dirp;
```

Sempre que uma operação de e/s exige um acesso a algum dos diretórios, a variável gdirp↑ é alocada sobre o heap e para ela o diretório desejado é copiado. O apontador gdirp faz parte da SYSCOM (item II.3.1) e o interpretador executa

```
SYSCOM↑.gdirp := nil;
```

antes da alocação de qualquer novo campo sobre o heap. Dessa forma, o diretório só ocupa espaço na memória enquanto está sendo efetivamente utilizado.

II.4.7- OPERAÇÕES DE ENTRADA E SAÍDA DE DADOS

Programas escritos em Pascal UCSD podem realizar operações de entrada e saída de dados de duas formas: através do uso de arquivos e através dos comandos unitread e unitwrite de manipulação direta das unidades de e/s mencionados no item II.3.5.

A utilização de arquivos é feita por meio de procedimentos implementados no primeiro segmento de PASCALSYSTEM (item II.4.3). Esses procedimentos incluem as operações para manipulação de arquivos definidas em JENSEN & WIRTH [28] (tais como reset, read e get, por exemplo), além de outros, como seek. Sua finalidade é dar ao usuário uma estrutura de e/s baseada no uso de variáveis do tipo file, independentes das unidades de e/s da máquina P, através de operações transparentes de formatação e bufferização que terminam em chamadas às rotinas básicas unitread e unitwrite.

A figura II.14 ilustra a organização das operações de e/s no Sistema P.

A rotina standard ioresult mencionada no item II.3.5 pode também ser usada na verificação da ocorrência de erros em operações de e/s para arquivos.

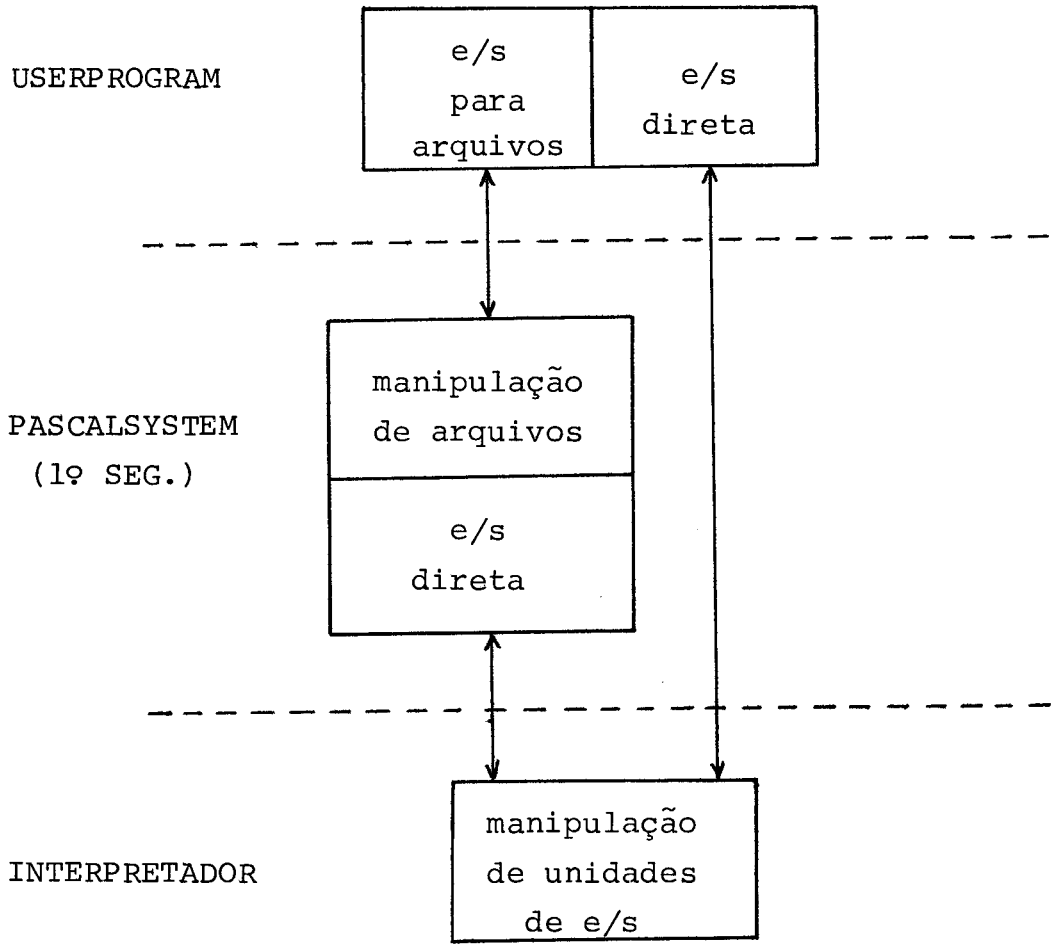


FIGURA II.14 - Hierarquia das Operações de E/S no Sistema P

CAPÍTULO III

PROCESSOS CONCORRENTES

III.1- INTRODUÇÃO

Como foi mencionado no capítulo I, a introdução de processamento paralelo em sistemas de computação visa essencialmente aumentar sua eficiência e otimizar a utilização de seus recursos. Muitas têm sido as formas de paralelismo adotadas em sistemas modernos. Essas englobam tipicamente a divisão de tarefas de um processador entre sub-unidades de processamento, a divisão de tarefas do sistema entre processadores e a multiprogramação de um processador. São vários, portanto, os níveis em que se pode encontrar paralelismo em sistemas de computação. Ao nível de processadores, por exemplo, algumas arquiteturas de computadores atuais utilizam processadores compostos por sub-unidades de processamento especializadas que operam em paralelo, realizando tarefas que apresentam um certo grau de independência entre si. Alguns exemplos (HAYNES et al[23] e STERLING[38]) são arquiteturas com características de "pipe-lining", processadores de vetores ("array processors") e coprocessadores. Também pode ser encontrado paralelismo em um nível um pouco mais alto, o de arquiteturas que empregam vários processadores para a execução simultânea de tarefas. Essas arquiteturas são conhecidas como arquiteturas a multiprocessadores e em WEITZMAN [39] é sugerida uma classificação que as divide em fortemente conectadas e fracamente conectadas (figura III.1).

Como pode ser visto na figura III.1, são ditas arquiteturas fortemente conectadas aquelas em que os processadores compartilham módulos de memória. A ligação entre os processadores e os módulos de memória é realizada das mais variadas formas, como por exemplo através de uma via única, várias vias com memórias multi-portas, ou várias vias reconfiguráveis em "cross-bar", entre outras. São chamadas arquiteturas fracamente conectadas a

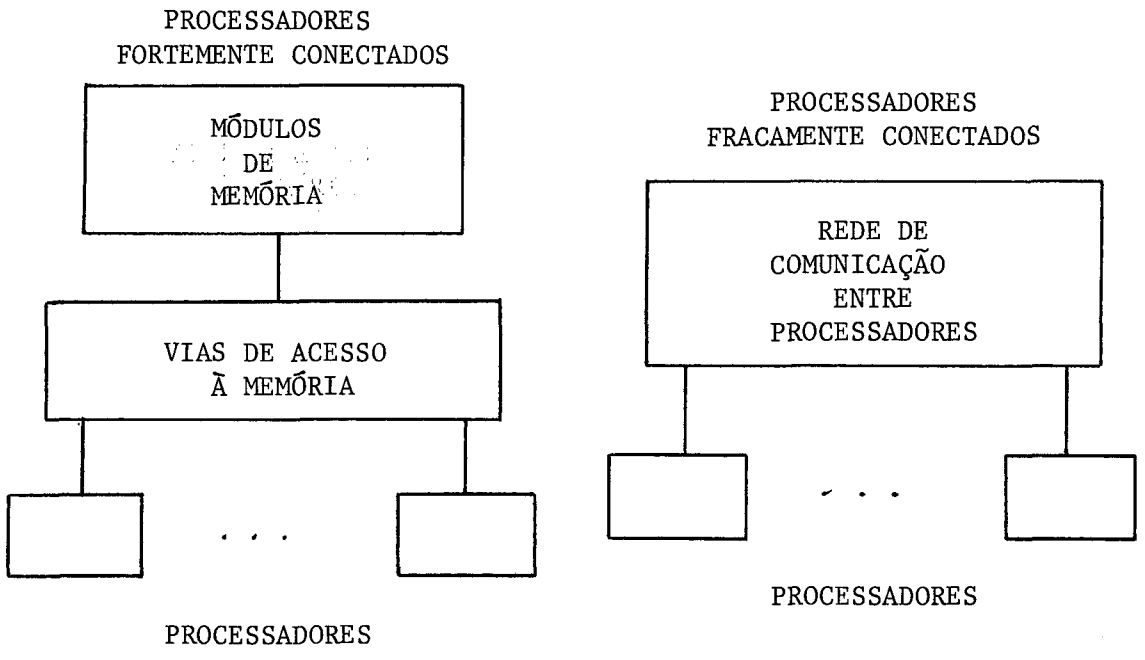


FIGURA III.1 - Arquiteturas a Multiprocessadores

quelas em que os processadores comunicam-se apenas através de canais de entrada e saída de dados. Também são várias as maneiras em que esses processadores podem ser interligados, formando redes. Alguns exemplos são redes em "bus", redes em anel, redes em estrela, etc.

A viabilidade econômica de arquiteturas a multiprocessadores é um fenômeno recente. Em anos anteriores, os sistemas de computação incorporavam características de processamento paralelo através do que se chama multiprogramação do único processador (ou dos poucos processadores) do sistema. Multiprogramar um processador consiste em alocá-lo a uma tarefa diferente de tempos em tempos. Essa alocação temporária do processador a uma tarefa pode ser modelada como se a cada tarefa fosse dado um processador permanente, virtual, de velocidade inferior à do processador real. Também nos sistemas atuais a multiprocessadores pode ser necessário multiprogramar cada processador, e isto depende de quantos e quais são os processadores do sistema, bem como da maneira como tarefas são alocadas aos diversos processadores.

Em sistemas a multiprocessadores (ou mesmo a um único processador multiprogramado), é usual atribuir-se a denominação processo a cada tarefa a ser realizada por cada processador. O seqüenciamento da execução dos diversos processos pode ser expresso por um grafo, como o da figura III.2. Nessa figura, cada nodo do grafo representa um processo que deve ser executado segundo as normas de precedência estabelecidas pela orientação e estrutura do grafo, possivelmente com algum paralelismo. Se os processos em execução paralela são totalmente independentes entre si, então eles são denominados processos disjuntos, e os resultados obtidos com sua execução paralela são idênticos aos que seriam obtidos caso sua execução ocorresse de qualquer outra maneira. Mais freqüentemente, porém, existe a competição entre processos paralelos por recursos do sistema, quer por razões de escassez desses recursos, quer pela necessidade de cooperação entre os processos (HANSEN [15]). Deve ser observado que o termo "recurso" aqui utilizado tem um significado bastante amplo e engloba processadores, memórias, dispositivos periféricos, variáveis, etc. Esses processos paralelos entre os quais há competição por recursos são chamados processos concorrentes e seus trechos que manipulam recursos compartilhados são chamados regiões críticas. Como ilustração, imaginem-se dois processos A e B em execução paralela e suponha-se que

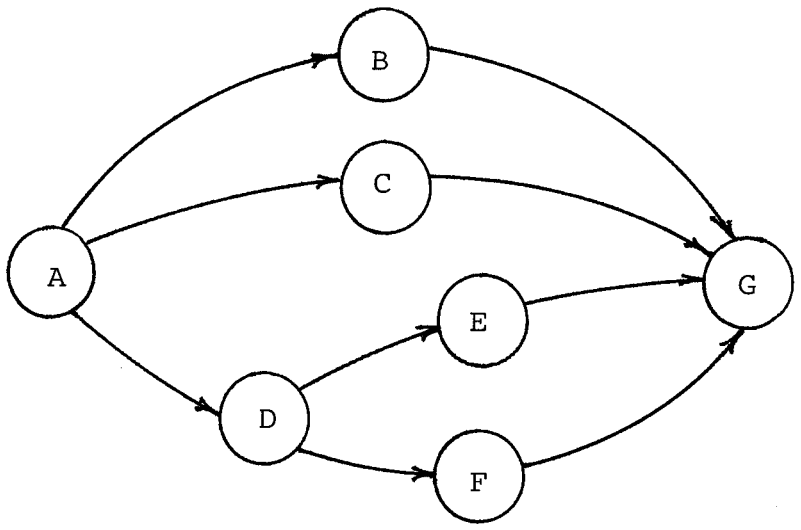


FIGURA III.2 - Um Grafo de Precedência entre Processos

eles competem pelo uso de um determinado recurso R durante a execução de um de seus trechos. Suponha-se também que A pode ser dividido em três subtarefas A1, A2 e A3, executáveis nessa ordem, e que A2 é a região crítica de A sobre R. Admitindo também a divisão de B nas subtarefas B1, B2 e B3, sendo B2 a região crítica de B sobre R, a execução de A e B realiza-se segundo uma das formas ilustradas na figura III.3, indistintamente. Observe

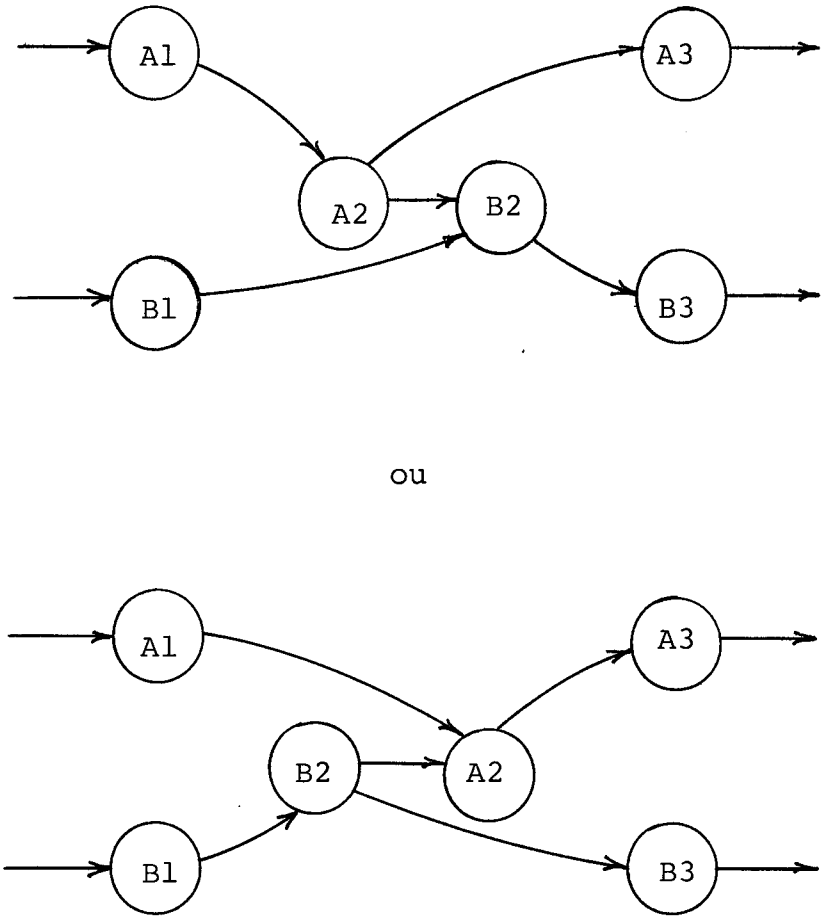


FIGURA III.3 - Execução de Regiões Críticas por Processos Concorrentes

-se que a execução paralela torna-se seqüencial durante as regiões críticas sobre R (A2 e B2). A necessidade de disciplinar o uso de recursos compartilhados está ligada à manutenção da inte

gridade desses recursos, o que pode ser conseguido garantindo que apenas um processo os use de cada vez e que o faça através de operações compatíveis com sua natureza. Também o problema de reprodutibilidade de resultados depende do compartilhamento disciplinado de recursos. Por exemplo, a atualização desordenada de uma variável por processos concorrentes produz resultados imprevisíveis quanto a seu valor final, fazendo-o depender de fatores como as velocidades de execução dos processos, etc. Em HOLT et al[27] há exemplos de erros desse tipo causados pela execução paralela desordenada de regiões críticas.

O projeto de sistemas concorrentes envolve dois problemas fundamentalmente: o do gerenciamento do acesso a recursos compartilhados e o da especificação da concorrência entre os processos. O problema de gerenciar o acesso a recursos compartilhados por processos concorrentes está tratado na seção III.2, onde encontram-se discutidas as principais ferramentas existentes para delimitar regiões críticas e executá-las com exclusão mútua. (Uma discussão sobre o mesmo tema pode ser encontrada em SEGRE[34]). Essa discussão encontra-se apresentada em ordem aproximadamente histórica com relação ao aparecimento das ferramentas citadas na literatura. O que está se chamando de especificação da concorrência é algum método que realize as relações de precedência entre as execuções de determinados processos. Esse problema é tratado na seção III.3, onde são apresentadas principais formas de especificação de concorrência já propostas.

III.2- COMPARTILHAMENTO DE RECURSOS POR PROCESSOS CONCORRENTES

III.2.1- CONSIDERAÇÕES GERAIS

Esta seção trata do gerenciamento de recursos compartilhados por processos concorrentes. Nos itens que se seguem são apresentadas as principais ferramentas já sugeridas para a exclusão mútua de processos concorrentes durante a execução de regiões críticas sobre um determinado recurso. De um modo ge-

ral , essas propostas buscam criar um conjunto de operações primitivas que alcancem as seguintes metas (DIJKSTRA [8]):

- A solução adotada deve ser simétrica entre os processos, o que significa que não podem ser estabelecidas prioridades estáticas;
- Nada pode ser suposto com relação às velocidades finitas de execução dos processos, nem mesmo que são constantes;
- O bloqueio de algum processo fora de uma região crítica não pode levar ao bloqueio de nenhum outro processo;
- Quando vários processos competem pelo direito de executar regiões críticas sobre um determinado recurso, a decisão sobre qual deve fazê-lo primeiro deve ser tomada em um tempo finito.

Deve ainda ser observado que no desenvolvimento de ferramentas para o compartilhamento de recursos por processos concorrentes tem sido constatada a tendência a incorporar o máximo possível de características de alto nível, com o objetivo de tornar o compartilhamento abstrato, independente da máquina. Apesar disso, existem métodos que são mais adequados a uma ou outra arquitetura em particular. O problema de adequação existe especialmente com relação às arquiteturas a multiprocessadores, nas quais a necessidade de comunicação entre processos traduz-se geralmente na necessidade de comunicação entre processadores. Nos itens a seguir é suposto que os métodos apresentados são mais adequados às arquiteturas fortemente conectadas (seção III.1), a menos que seja dito o contrário.

O apêndice B fornece exemplos da utilização dos métodos apresentados nesta seção.

III.2.2- SEMÁFOROS E REGIÕES CRÍTICAS

O uso de semáforos para sincronização de processos concorrentes foi proposto originalmente em DIJKSTRA [7] . Semáforos são definidos como inteiros não-negativos sobre os quais são realizáveis duas operações indivisíveis: as operações P e V. Se S é um semáforo, então:

- A operação P(S) realiza o decremento de S por uma unidade, se isso não o for tornar negativo. Se o decremento não pode ser realizado, o processo bloqueia-se até que possa fazê-lo;
- A operação V(S) incrementa o valor de S por uma unidade, permitindo a continuação de algum processo que possa estar esperando que S se torne não-nulo.

O fato de as operações P(S) e V(S) sobre o semáforo S terem que ser indivisíveis significa que elas constituem regiões críticas sobre S e que apenas um P ou um V pode ser executado de cada vez sobre um mesmo semáforo. Essa indivisibilidade pode ser implementada através da associação de um flag F ao semáforo S (SHAW [35]). Sobre F define-se uma operação indivisível chamada "test-and-set", como apresentada no algoritmo III.1. A indivisibilidade dessa operação pode ser conseguida

```
function testandset(var F: boolean): boolean;
begin
  testandset := F;
  F := true
end;
```

ALGORITMO III.1 - Operação "test-and-set"

transformando-a em um código de operação do processador e, no caso de sistemas a multiprocessadores, executando esse código com algum mecanismo de trancamento das vias de acesso à memória ("bus-lock"). Com o uso da operação testandset do algoritmo III.1 é construída a operação busywait do algoritmo III.2,

```
procedure busywait(var F: boolean);
begin
  while testandset(F) do (* espera *)
end;
```

ALGORITMO III.2 - Espera Ocupada

utilizada na implementação das operações P(S) e V(S) sobre um semáforo S. Para tanto, são utilizados dois flags F1 e F2 inicializados com false e tem-se:

```
- P(S): busywait(F1);
        S := S - 1;
        if S < 0 then
          begin F1 := false; busywait(F2) end
        else F1 := false;

- V(S): busywait(F1);
        S := S + 1;
        if S <= 0 then F2 := false;
        F1 := false;
```

Uma classe particular de semáforos é a dos semáforos binários, isto é, semáforos que assumem apenas os valores 0 ou 1. Semáforos binários podem ser implementados com um único flag F e as operações P(S) e V(S) sobre um semáforo binário S tornam-se:

```
- P(S): busywait(F);
- V(S): F := false;
```

Implementar semáforos através de formas ocupadas de espera como a do algoritmo III.2 pode degradar o desempenho do sistema, especialmente nos casos em que os recursos de processamento são escassos. Uma solução para esse problema é substituir a operação busywait(F2) por uma forma de bloqueio que utilize filas de processos. Essa técnica consiste em associar a cada processo uma estrutura de dados que o descreva e que possa armazenar o estado do processador quando de um eventual bloqueio, com vistas a uma futura reativação. As estruturas de dados associadas aos processos podem formar listas encadeadas, que podem ser interpretadas como filas de processos. Assim, uma operação P(S) que não possa ser realizada causa o bloqueio do processo numa fila associada ao semáforo S e a operação V(S) que encontre processos bloqueados nessa fila libera algum deles (figura III.4). O uso de filas para bloquear processos é

interessante também porque permite aplicar uma política para a liberação desses processos, o que não era possível com espera ocupada. Com relação a essa política é importante observar que, apesar de até certo ponto arbitrária, ela deve ser tal que garanta o atendimento a todos os processos em algum instante que não possa ser adiado indefinidamente (item III.2.1). Uma política baseada na ordem de chegada (FCFS - "first come, first served"), por exemplo, realiza esse objetivo. Uma vez adotadas filas de processos, a única forma ocupada de espera que resta é a busywait(Fl) , que poderá ser bastante rápida se as operações P e V forem executadas com interrupções inibidas.

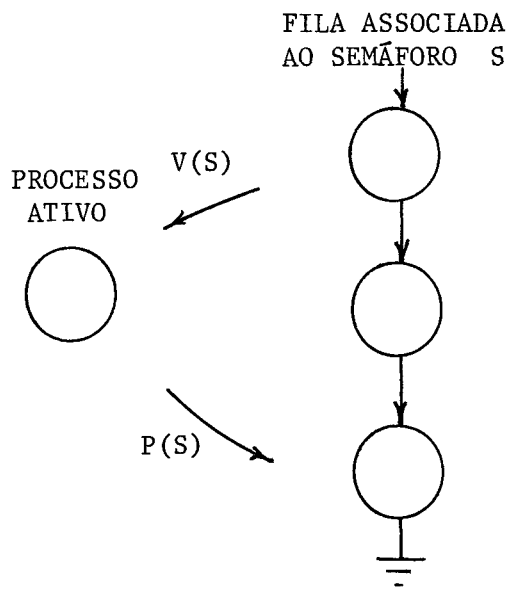


FIGURA III.4 - Implementação de Semáforos com Filas

Um semáforo inicializado com o valor 1 pode ser usado para prover exclusão mútua entre processos que concorrem por um mesmo recurso. Se mutex é esse semáforo, então cada região crítica sobre o recurso deve ser precedida por um P(mutex) e sucedida por um V(mutex). Observe-se que semáforos utilizados para realizar exclusão mútua assumem apenas os valores 0 e 1, o que os qualifica como semáforos binários.

Na seção B.2 encontra-se um exemplo de utilização de semáforos em suas duas formas, geral e binária. Apesar da simplicidade introduzida pelo uso da forma geral de semáforos,

existem em DIJKSTRA [7] indicações de que esta pode ser sempre substituída pela binária, em arranjos mais complexos.

Uma das principais críticas feitas aos semáforos é que eles contêm intrinsecamente pouca informação. Quando um semáforo é utilizado para exclusão mútua, por exemplo, é impossível saber em tempo de compilação a que recurso ele se refere, visto que nada há em sua definição que o relacione ao recurso. A grande vantagem de se poder realizar esses testes em tempo de compilação é evitar uma série de erros dependentes do tempo e de difícil depuração. Além disso, semáforos têm sido também criticados por não permitirem a estruturação do sistema, tornando complicada a programação.

Com o objetivo de dar ao compilador a possibilidade de verificar que o acesso a recursos compartilhados só é realizado dentro de certas regiões protegidas, foi proposta em HOARE [24] (e também em HANSEN [13]) uma forma de especificar regiões críticas através da construção

with R do RC

onde R é um recurso qualquer e RC é uma região crítica sobre R. A tarefa do compilador é verificar que não são realizadas referências a R fora de RC.

A seção B.3 apresenta um exemplo do uso da cláusula with no acesso mutuamente exclusivo a recursos compartilhados.

Uma forma de implementar essa proposta é o compilador associar um semáforo S_R (inicializado com 1) ao recurso R e gerar o código correspondente a

$$P(S_R);\ RC;\ V(S_R)$$

para a execução da região crítica RC sobre R.

III.2.3- REGIÕES CRÍTICAS CONDICIONAIS E FILAS DE EVENTOS

O uso da cláusula with apresentada no item III.2.2 realiza apenas a exclusão mútua no acesso a recursos comparti

lhados, com a possibilidade de testes em tempo de compilação. A necessidade de eventuais bloqueios arbitrários dentro da região crítica por não cumprimento de determinadas condições continua a ser satisfeita pelo uso de semáforos, como no exemplo da seção B.3. Também em HOARE[24] e HANSEN [13] existe uma proposta para incorporar testes de condições na cláusula with, formando a chamada região crítica condicional que em HOARE [24] é denotada por

with R when C do RC

onde R é um recurso qualquer sobre o qual a condição C e a região crítica RC são definidas. Em HANSEN[14] existe um estudo comparativo entre o uso de regiões críticas simples e semáforos e o uso de regiões críticas condicionais. O ponto central dessa comparação é que cada ferramenta expressa de uma forma mais clara um certo tipo particular de interação entre processos, forçando formas obscuras de programação quando utilizada inadequadamente.

A forma de implementação de regiões críticas condicionais sugerida em HOARE[24] é similar à apresentada no algoritmo III.3. São utilizados dois semáforos binários: um para mútua exclusão (mutex), inicializado com 1, e outro (cond), inicializado com 0, destinado a bloquear todos os processos que necessitem de alguma condição, não satisfeita, sobre R. Sempre que algum processo consegue executar sua região crítica, os demais podem reavaliar suas condições, que podem já ter sido satisfeitas. Essa reavaliação periódica de condições é conhecida como forma controlada de espera ocupada, e resulta essencialmente de haver uma única fila para todas as condições sobre o mesmo recurso, quaisquer que sejam. Se o recurso R é muito concorrido e se a capacidade de processamento do sistema é baixa, então a espera ocupada controlada degrada o desempenho deste.

Uma forma de eliminar esse inconveniente no tratamento de condições em regiões críticas foi proposta em HANSEN[13], e consiste em associar uma fila independente a cada condição que possa se referir a um determinado recurso. Essas filas são

```
libera: if a fila de cond não está vazia then
          V(cond)
        else V(mutex); ret;

entraRC: P(mutex);
         while not C do
           begin
             call libera; P(cond)
           end;
         executa RC;
         call libera;
```

ALGORITMO III.3 - Uma Implementação de Regiões Críticas Condicionais

chamadas filas de eventos, declaradas por

```
var E: event R;
```

que associa a fila de eventos (E) ao recurso (R). Sobre uma variável do tipo event definem-se duas operações:

- a operação await(E) para bloquear o processo que a executa na fila de eventos E, liberando o recurso R;
- e a operação cause(E) para reativar algum dos processos que estejam bloqueados na fila de eventos E, liberando o recurso R caso não haja nenhum.

As operações await(E) e cause(E) são utilizadas em conjunto com as regiões críticas simples apresentadas no item III.2.2 e só podem ser chamadas de uma região crítica sobre o recurso R a que E se refere. É importante notar que a operação cause(E) deve ser a última operação da região crítica, uma vez que permite a eventual continuação de outro processo dentro da mesma região.

As seções B.4 e B.5 ilustram a utilização de regiões

críticas condicionais e de filas de eventos, respectivamente.

III.2.4- TROCA DE MENSAGENS

Na utilização de recursos compartilhados por processos concorrentes, manter a integridade desses recursos deve ser um objetivo fundamental. A integridade de recursos compartilhados pode ser mantida através de:

(i) garantir exclusão mútua entre os processos que concorrem por um determinado recurso R durante a execução de suas regiões críticas sobre R;

(ii) assegurar que sobre um determinado recurso R serão realizadas apenas operações pertencentes a um conjunto pré-determinado, compatíveis com a natureza de R.

As técnicas apresentadas nos itens anteriores evoluíram no sentido de assegurar a exclusão mútua na manipulação de recursos compartilhados por processos concorrentes, sem porém nenhuma restrição quanto a que tipo de operações realizar sobre eles.

O emprego de troca de mensagens entre processos pode conduzir a uma melhora nesse sentido. De fato, a forma mais simples de se implementar exclusão mútua por meio de troca de mensagens é através da criação de um processo gerente do recurso, o único a manipulá-lo diretamente. Observe-se que dessa maneira pode-se garantir que apenas determinadas operações são realizáveis sobre o recurso, especificamente aquelas definidas no corpo de seu processo gerente.

Os processos que necessitem utilizar o recurso dirigem-se ao gerente através de mensagens trocadas por meio de operações destinadas a enviar e receber mensagens. Essas operações são usualmente denominadas send e receive e podem funcionar de maneira síncrona ou assíncrona, respectivamente bloqueando ou não um processo que execute um send (receive) até que seja executada por outro processo a operação simétrica receive (send). Em um contexto histórico,o Sistema RC 4000 (HANSEN[12]) representa um exemplo importante de aplicação de tro

ca de mensagens no qual são usadas a forma assíncrona da operação send e a forma síncrona da operação receive (denominada, por isso, wait). Essas operações são semelhantes a:

- send(P,M,E) - Encadeia a mensagem M numa fila de mensagens para o processo P, passando o endereço E para troca de dados;
- wait(P,M,E) - Retira da fila de mensagens do processo que a chamou uma mensagem com atributos P,M e E. P é a identificação do processo que enviou a mensagem, M é a própria mensagem e E é um endereço para troca de dados. Se a fila estiver vazia, o processo é atrasado até chegar uma mensagem.

Na seção B.6 encontra-se um exemplo da utilização das operações send e wait.

III.2.5- MONITORES: AS LINGUAGENS PASCAL CONCORRENTE E MODULA

Foi dito na seção III.2.4 que concentrar todas as operações realizáveis sobre um recurso como atribuição de um processo gerente desse recurso contribui para manter sua integridade. Entretanto, como demonstra o exemplo da seção B.6, a utilização de um processo normal para gerenciar um recurso produz algoritmos muito complicados, essencialmente porque um processo é uma seqüência de comandos que devem ser executados em uma ordem pré-determinada. Em DIJKSTRA[8] e em HANSEN[15] essa dificuldade é reconhecida e é proposta uma entidade de natureza "semi-seqüencial" à qual atribuir a tarefa de gerenciar o recurso. Essa entidade, chamada "secretária" ou monitor, consiste num conjunto de rotinas cuja execução segue uma ordem indeterminada.

A primeira proposta concreta de um monitor encontra-se em HOARE [25], onde monitores são definidos como um conjunto de dados e rotinas que os manipulam, estruturados de acordo com a seguinte notação:

```
nome-do-monitor: monitor;
  begin ... declaração dos dados locais do monitor;
    procedure nome-da-rotina(... parâmetros formais ...);
      begin ... corpo da rotina ... end;
    ... declarações das outras rotinas locais do monitor;
    ... inicialização dos dados locais do monitor ...
  end;
```

Os dados locais do monitor constituem o recurso compartilhado que ele gerencia e as rotinas locais do monitor são as únicas operações realizáveis sobre o recurso. Quando um processo deseja realizar uma dessas operações, ele o solicita através da chamada

```
nome-do-monitor.nome-da-rotina(... parâmetros reais ...)
```

Apenas uma dessas chamadas pode ser atendida de cada vez, o que pode ser implementado associando um semáforo S_M (inicializado com 1) ao monitor e envolvendo a rotina chamada por um par $P(S_M)$ ... $V(S_M)$.

Dentro do monitor, a sincronização por condições é realizada por filas de eventos, como no item III.2.3. Essas filas são declaradas como variáveis do tipo condition; se C é uma dessas variáveis, então as operações de espera e sinalização são denotadas por C.wait e C.signal, respectivamente. Como no no item III.2.3, a operação de espera libera o monitor. Em algumas implementações, a operação de sinalização deve ser a última a ser realizada e o processo que o fez libera o monitor para ser executado pelo processo sinalizado; em outras, a sinalização pode ocorrer em qualquer lugar e o processo que a realizou bloqueia-se em uma fila, de onde sairá para continuar no monitor após o término da região crítica do processo sinalizado.

O conceito de monitor foi implementado em algumas linguagens para programação concorrente, das quais Pascal Concorrente (HANSEN[16 17]) e Modula (WIRTH[43]) são exemplos. Em Pascal Concorrente monitores são declarados como tipos abstratos do

Pascal, o que permite haver vários monitores do mesmo tipo. Condições são variáveis do tipo queue, sobre as quais atuam as operações delay(Q) e continue(Q), se Q é uma dessas variáveis. Em Modula, monitores são declarados como interface modules e condições são variáveis do tipo signal, manipuladas pelas operações wait(S) e send(S), onde S é do tipo signal. Nas seções B.7 e B.8 encontram-se exemplos do uso de Pascal Concorrente e Modula, respectivamente.

Em KESSELS[30] é apresentada uma alternativa ao uso de filas de eventos em monitores e em SEGRE & SANTOS[33] as várias formas e implementações de monitores são discutidas.

O uso de monitores como única ferramenta para escalonamento de processos concorrentes em seu uso de recursos compartilhados tem recebido algumas críticas. Apesar de representarem um método de sincronismo seguro e bem estruturado, sua relativa inflexibilidade quanto a políticas de escalonamento deixa dúvidas sobre a viabilidade de seu uso em sistemas de certo porte. Essa e outras críticas podem ser encontradas em KEEDY[29].

III.2.6- PROGRAMAÇÃO NÃO-DETERMINÍSTICA: COMANDOS GUARDADOS, CSP E DP

A concentração de todas as operações realizáveis sobre um recurso compartilhado em uma única entidade responsável por sua manipulação foi apontada nos itens III.2.4 e III.2.5 como uma forma de assegurar a integridade do recurso em questão. No item III.2.4 essa entidade era um processo como os outros, tendo nas trocas de mensagens o meio para se comunicar com os processos interessados em utilizar seu recurso. O item III.2.5 introduziu o conceito de monitor, responsável também pela gerência de um recurso compartilhado, permitindo porém formas muito mais simples de programação. Esse ganho em simplicidade foi atribuído ao não-determinismo potencial existente na execução do monitor: apesar de programado de maneira determinística, a ordem em que suas diversas partes são executadas depende de políticas de escaloramento propositalmente não especificadas.

Em contraste com o conceito de processo, o de monitor expressa uma entidade passiva, cuja atuação deve ser solicitada pelos processos que têm acesso a ele. A "execução" do monitor deve ser entendida como a execução de um processo dentro do monitor, o que o torna mais adequado a arquiteturas com um único processador ou com muitos processadores fortemente conectados (item III.2.1). Esse é o caso de todas as técnicas de sincronismo discutidas até agora (itens III.2.2 a III.2.5), com exceção da troca de mensagens (item III.2.4).

Com a crescente viabilidade econômica das arquiteturas a multiprocessadores fracamente conectados para algumas aplicações (WEITZMAN[39]), tem havido a preocupação de incorporar características de programação não-determinística às linguagens. O objetivo é tornar o uso de processos gerentes (adequado às arquiteturas fracamente conectadas) mais simples, a exemplo dos monitores, não-determinísticos em alguns aspectos.

Em DIJKSTRA [10] podem ser encontradas considerações sobre programação não-determinística, sendo enfatizado o aspecto de simplificação do desenvolvimento de certas classes de programas. A primeira proposta para programação não-determinística está em DIJKSTRA[9], onde são sugeridos os chamados comandos guardados ou GC's.

Um GC tem a forma geral

expressão lógica (B) → lista de comandos (SL)

ou

$$B \rightarrow SL$$

onde SL só pode ser executada quando B é verdadeira. Um conjunto de n GC's é simbolizado por

$$GC_1 \ \underline{or} \ GC_2 \ \underline{or} \ \ldots \ \underline{or} \ GC_n$$

ou por

$$B_1 \rightarrow SL_1 \ \underline{or} \ B_2 \rightarrow SL_2 \ \underline{or} \ \ldots \ \underline{or} \ B_n \rightarrow SL_n$$

Essa proposta de programação não-determinística utiliza GC's sob a forma de dois comandos:

(i) Comando alternativo:

if conjunto de GC's fi

Funcionamento:

if algum dos guardas (B) do conjunto de GC's é verdadeiro then
seleciona arbitrariamente uma lista de comandos (SL) com guarda (B) verdadeiro para execução
else
aborta a execução;

(ii) Comando repetitivo:

do conjunto de GC's od

Funcionamento:

repeat
seleciona arbitrariamente uma lista de comandos (SL) com guarda (B) verdadeiro para execução
until todos os guardas (B) serem falsos;

Uma notação para programação não-determinística encontra-se em HOARE [26], onde são apresentados os chamados Processos Seqüenciais em Comunicação (CSP) para arquiteturas distribuídas fracamente conectadas. O ponto central dessa proposta é que GC's devem ser utilizados como estrutura de controle não-determinístico e operações de e/s como primitivas de programação. Essas primitivas seriam da forma:

receive(P,m) - receber uma mensagem do processo P através de m;

send(P,m) - enviar ao processo P a mensagem contida em m.

O processo que executa uma dessas operações é atrasado até que o processo especificado na operação execute a operação simétrica (a menos que este já o tenha feito). Uma implementação dos CSP encontra-se descrita em SHRIRA & FRANCEZ[37] e a seção B.9 fornece um exemplo de seu uso.

O conceito de GC's foi estendido em HANSEN & STAUNSTRUP[18] para as chamadas regiões guardadas, denotadas por GR ou por

when conjunto de GC's end

O funcionamento de uma GR pode ser descrito por

repeat (* espera *)
until algum dos guardas (B) do conjunto de GC's ser verdadeiro;
seleciona arbitrariamente uma lista de comandos (SL) com guarda (B) verdadeiro para execução;

Em HANSEN[19] é proposta uma nova forma de GR, denotada por

cycle conjunto de GC's end

cujo funcionamento equivale à repetição por tempo indefinido do comando when. Também em HANSEN[19] é proposta uma outra notação para programação não-determinística, através dos chamados Processos Distribuídos (DP) para arquiteturas distribuídas fracamente conectadas. A comunicação entre DP realiza-se através da chamada de rotinas definidas em outro DP. Cada DP possui a estrutura geral

nome do processo
variáveis exclusivas
rotinas comuns
comando inicial

e sua execução alterna-se entre o comando inicial e as rotinas comuns chamadas por outro DP. Observe-se que há comportamento não-determinístico não só nas GR's como também na ordem em que as chamadas às rotinas comuns são atendidas.

Na seção B.10 pode ser encontrado um exemplo do uso de DP e em WELSH et al[40] é feito um estudo comparativo entre CSP e DP.

III.2.7- AS LINGUAGENS ADA E EDISON

Ada é uma linguagem recente que incorpora características de programação não-determinística. Uma descrição a nível introdutório da linguagem Ada encontra-se em BARNES [2] e em LEDGARD [31]. Ada incorpora duas características que permitem projetar processos gerentes de recursos compartilhados: packages e tasks. Packages são construções destinadas a encapsular o recurso compartilhado e as operações realizáveis sobre ele (não como o monitor do item III.2.5, pois essas operações não se excluem no tempo). Tasks são as unidades ativas (processos) de processamento paralelo e comunicam-se através de chamadas de rotinas. Por exemplo, se uma task T define uma entry E, então essa entry pode ser chamada por outra task através de T.E e essa chamada só será atendida quando T executar um

accept E do S end

Ao término da execução de S, a task que originou a chamada poderá continuar sua execução. O sincronismo entre essas duas tasks é tal que aquela que chegar primeiro ao ponto de comunicação deverá esperar a outra. Gerentes de recursos compartilhados podem ser escritos com o uso combinado de packages e tasks. A finalidade dessas tasks é prover exclusão mútua, através de comandos select que introduzem não-determinismo. A forma de um comando select é

select S_1 or S_2 or...or S_n end select

que seleciona para execução um comando qualquer dos que possam ser executados, de S_1 a S_n (um comando poderá não ser executável se for um comando guardado por uma cláusula when com guarda falso). A seção B.11 apresenta um exemplo do uso da linguagem Ada. Nesse exemplo são introduzidos guardas e apenas tasks são utilizadas, uma vez que suas entries já correspondem a to-

das as operações realizáveis sobre o recurso. Em WELSH & LISTER[41] encontra-se um estudo comparativo entre Ada, CSP e DP (item III.2.6).

A linguagem Edison encontra-se descrita em HANSEN[20 21 22]. Em Edison, recursos e operações realizáveis sobre eles são encapsulados em modules, que possuem entidades exportáveis. As rotinas exportadas podem ser chamadas por processos e, como nos packages de Ada, não se excluem mutuamente no tempo. A exclusão mútua é dada por uma forma simplificada de região crítica condicional (item III.2.3)

when C do RC

que introduz não-determinismo na ordem em que as chamadas às rotinas do module são atendidas. Por não especificar o recurso em questão, essa forma de sincronismo provê exclusão mútua global no acesso aos recursos. Em conseqüência, intensifica a espera ocupada controlada das regiões críticas condicionais do item III.2.3 e elimina o paralelismo que poderia haver no acesso a recursos compartilhados diferentes. Na opinião de seu autor, esses problemas não são significativos em sistemas a multimicroprocessadores, em função dos quais a linguagem foi projetada. Na seção B.12 existe um exemplo do uso da linguagem Edison.

III.3- ESPECIFICAÇÃO DA CONCORRÊNCIA ENTRE PROCESSOS

As formas de especificar a concorrência entre processos seguem uma classificação geral que as divide no que se pode chamar de formas estáticas e dinâmicas.

Especificar a concorrência entre processos de maneira estática é fazê-lo independentemente de sua execução, por exemplo através da ativação simultânea de todos eles, fazendo-os existir indefinidamente como entidades ativas. Uma outra forma de especificar concorrência estaticamente é de algum modo fornecer ao Sistema um grafo de precedência entre os processos, que são então ativados de acordo com esse grafo. Esses esquemas estáticos de especificação de concorrência podem apresen-

tar desvantagens em alguns casos em que a geração de processos concorrentes dependa de eventos de natureza não-determinística (como em sistemas para atuação em tempo real, por exemplo).

As formas dinâmicas de especificação de concorrência são realizadas através de comandos de uma linguagem de programação. Dado um programa, a execução concorrente de alguns de seus trechos depende naturalmente da identificação desses trechos como processos concorrentes. Vários são os níveis em que essa identificação pode ser realizada. Por exemplo, a avaliação de uma expressão aritmética de certa complexidade pode ser realizada por meio da computação simultânea de algumas de suas partes. A figura III.5 apresenta um grafo representativo de como pode ser avaliada a expressão X = (2*A) + ((C-D) /3). Os nodos desse grafo podem ser associados a processos concorrentes. Também podem ser identificadas possibilidades de execução

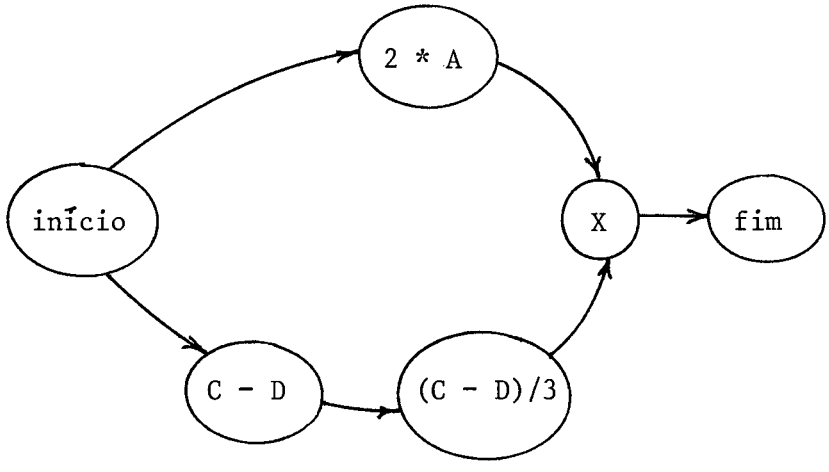


FIGURA III.5 - Paralelismo na Avaliação de uma Expressão Aritmética

concorrente em um nível um pouco mais alto, que é o dos comandos de uma linguagem. Para um programa escrito na linguagem Pascal, por exemplo, pode-se especificar a execução de um comando case ao mesmo tempo em que um comando while e um de atribuição são executados, por exemplo. Em um nível ainda mais alto, processos concorrentes podem ser associados a grupos de comandos, como rotinas e outras construções disponíveis para estruturar um programa e torná-lo modular.

Uma vez identificados os trechos de um programa a serem associados a processos concorrentes, a especificação dinâmica da concorrência entre eles é realizada em tempo de execução, o que lhe confere um maior grau de aplicabilidade. Uma forma de realizar essa especificação dinâmica é por meio dos comandos fork/join/quit sugeridos em CONWAY [6]. Imaginem-se, por exemplo, dois processos A e B, estando A em execução. Quando A executa um comando fork B, o processo B é ativado e A continua sua execução. A execução de B continua até que apareça um comando quit e a de A até que execute um comando join B, que pode atrasá-lo até B terminar (figura III.6).

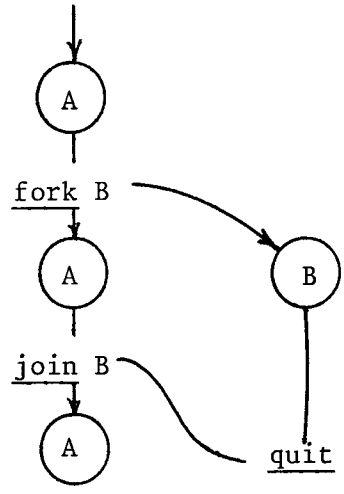


FIGURA III.6 - Utilização dos Comandos fork/join/quit

A concorrência entre processos também pode ser especificada dinamicamente através do par de comandos parbegin/parend ou cobegin/coend sugerido em DIJKSTRA[7]. Seu uso causa a ativação simultânea de um certo número de processos e o bloqueio do processo que o executou até que todos os processos ativados terminem. A figura III.7 ilustra a ativação dos processos B, C e D pelo processo A e a de E e F por D; o grafo da figura mostra o bloqueio de A até o término de B, C e D e o de D até que terminem E e F. O grafo realizado por meio de comandos cobegin/coend é tal que o processo que executa um desses comandos aparece duas vezes, em situações simétricas, como A e D no grafo da figura III.7. Essa propriedade intrínseca de simetria restringe a classe de grafos realizáveis pelos comandos cobegin/coend.

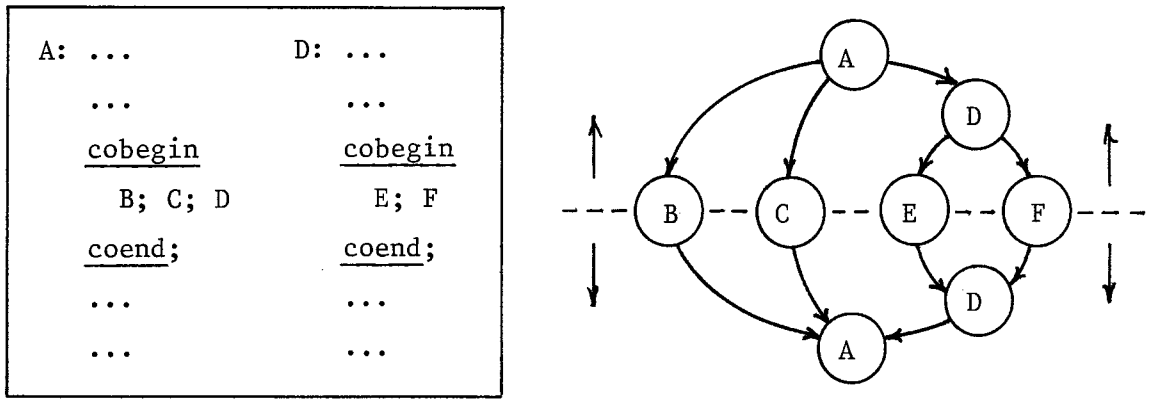


FIGURA III.7 - Utilização dos Comandos cobegin/coend

Os comandos fork/join/quit e cobegin/coend foram apresentados acima como ferramentas para especificar dinamicamente a execução concorrente de processos, através de sua ativação e desativação. Um caso particular de especificação de concorrência é o que se pode chamar de sinalização entre processos, realizada por meio da sincronização de sua execução em determinados pontos. Nesse caso a concorrência que já havia sido especificada quando da ativação dos processos é momentaneamente alterada, devido a uma necessidade local de sincronismo. Suponham-se, por exemplo, dois processos concorrentes A e B, um ponto x um A e um ponto y em B. Suponha-se também que A deve ser executado até o ponto x e então esperar que B atinja o ponto y a fim de continuar. Uma forma de resolver esse problema é utilizar semáforos ou troca de mensagens, apresentados nos itens III.2.2 e III.2.4, respectivamente. Ao chegar ao ponto x, A executa um P(S) , sendo S um semáforo inicializado com 0, ou espera por uma mensagem de B, através da operação wait. B, por sua vez, sinaliza a passagem pelo ponto y através de um V(S) sobre o mesmo semáforo S ou do envio de uma mensagem a A por meio da operação send. Se a sincronização que se deseja é bilateral, ou seja, se também B deve bloquear-se em y à espera da passagem de A por x, então além de semáforos e troca de mensagens podem ser utilizados os comandos apresentados nesta se

ção para a especificação dinâmica da concorrência entre processos. Os comandos fork/join/quit, por exemplo, podem ser utilizados através da divisão de A em A1 e A2 no ponto x ou de B em B1 e B2 no ponto y (figura III.8). Observe-se que esses comandos aplicam-se se as ativações de A e B não são independentes entre si. Também os comandos cobegin/coend podem ser utilizados, se A e B são gerados pelo mesmo processo ao mesmo tempo. Para tanto os pontos x e y são utilizados respectivamente para dividir A em A1 e A2 e B em B1 e B2. A sincronização é então obtida segundo o grafo da figura III.9, onde P é o processo gerador de A e B, executando dois pares de comandos cobegin/coend em seqüência.

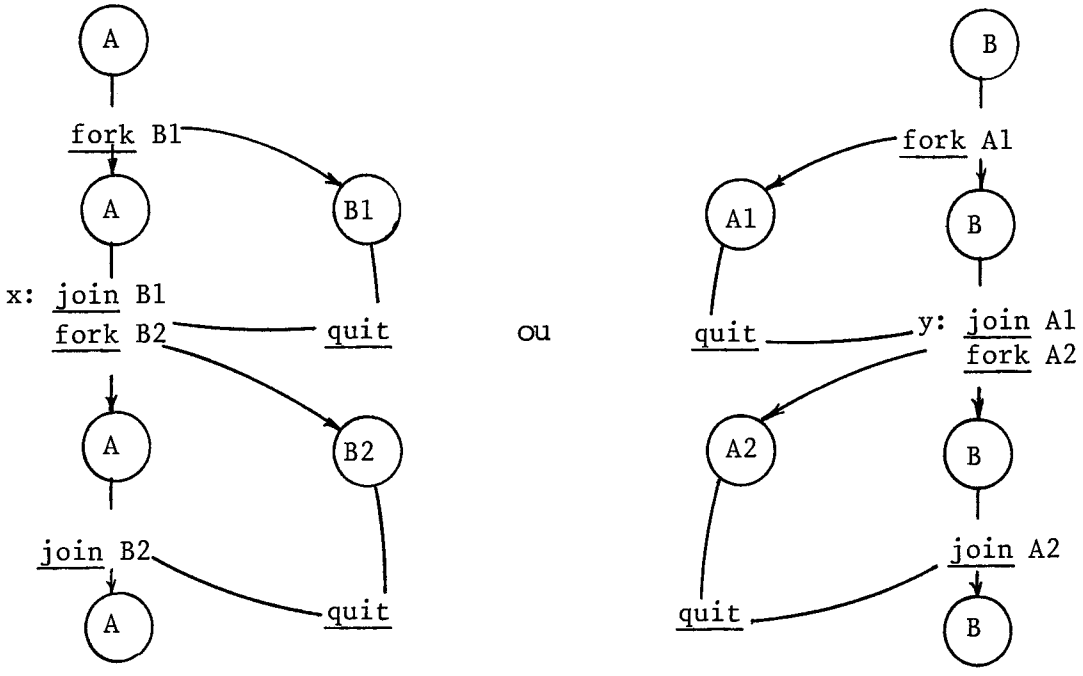


FIGURA III.8 - Uso dos Comandos fork/join/quit para Sincronização de Processos

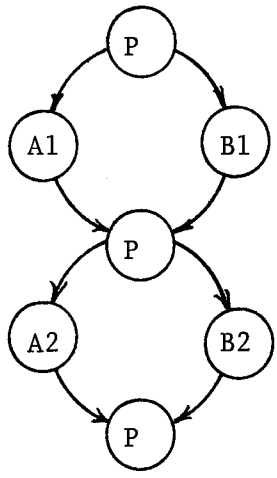


FIGURA III.9 - Uso dos Comandos cobegin/coend para Sincronização de Processos

CAPÍTULO IV

PROCESSOS CONCORRENTES NO SISTEMA P

IV.1- INTRODUÇÃO

Este capítulo trata da extensão para processamento concorrente do Sistema P apresentado no capítulo II. O objetivo é incorporar à linguagem Pascal UCSD algumas ferramentas que permitam escrever programas concorrentes nessa linguagem e especificar um núcleo para a linguagem resultante dessas extensões. As funções do núcleo incluem realização da concorrência entre processos e gerenciamento de recursos compartilhados através da sincronização dos processos concorrentes. Os recursos compartilhados pelos processos concorrentes incluem processadores virtuais, memória, unidades de entrada e saída de dados e recursos do próprio programa de que fazem parte os processos. Conforme mencionado no capítulo I, apenas parte dessa adaptação encontra-se descrita neste trabalho. Essa parte inclui os problemas de escalonamento dos processos concorrentes em geral. Problemas de gerência de memória, construção de um interpretador para processadores reais mais potentes e demais problemas mais intimamente ligados à arquitetura física utilizada encontram-se descritos em MOTTA [32].

Na extensão do Sistema P para processamento concorrente procurou-se manter o novo sistema compatível com os programas seqüenciais já existentes. Essa decisão apoia-se no fato de que existe uma grande quantidade de software desenvolvido para o Sistema P e que esse software é composto por programas de execução seqüencial. Como de parte desse software dispõe-se apenas do código P resultante de sua compilação, é interessante que essa compatibilidade mantenha-se mesmo a esse nível. Como conseqüência, o compilador Pascal deve ser capaz de realizar a distinção entre programas seqüenciais e programas concorrentes. Essa distinção é feita por meio de uma nova palavra reservada,

concurrent, utilizada para compor o cabeçalho de um programa concorrente em Pascal por meio de concurrent program "nome-do-programa". Um programa identificado por esse cabeçalho, e apenas ele, tem acesso às ferramentas incorporadas ao Pascal UCSD para processamento concorrente.

As próximas seções tratam das principais catacterísticas introduzidas no Sistema P em sua extensão para processamento concorrente. A seção IV.2 é dedicada aos problemas de identificação de processos concorrentes em um programa e especificação da concorrência entre eles. Na seção IV.3 são apresentadas as principais formas de arquiteturas adequadas ao tipo de processamento concorrente adotado para o Sistema P. A seção IV.4, finalmente, discute as operações primitivas para compartilhamento de recursos pelos processos concorrentes no Sistema P.

IV.2- ESPECIFICAÇÃO DA CONCORRÊNCIA ENTRE OS PROCESSOS DO SISTEMA P

Para especificar a concorrência entre os processos do Sistema P foram adotados os comandos cobegin/coend apresentados na seção III.3. Essa escolha deve-se a dois fatores. O primeiro é que esses comandos permitem a especificação da concorrência entre processos de uma forma que foi chamada dinâmica na seção III.3. Essa especificação dinâmica estende o uso do Sistema P a uma classe mais ampla de programas concorrentes. O segundo fator para a escolha dos comandos cobegin/coend é o fato de serem mais estruturados que os comandos da outra alternativa proposta para especificação dinâmica de concorrência (fork/join/quit). Assim, na extensão do Sistema P para processamento concorrente, o compilador Pascal deve reconhecer os comandos cobegin/coend e gerar a ativação simultânea dos processos compreendidos entre esses dois comandos.

Foi mencionado na seção III.3 que a identificação de um processo concorrente em um programa pode ser feita em diversos níveis, que vão de partes de expressões a rotinas. Para o

Sistema P foram adotadas rotinas, mais especificamente procedures e segment procedures do Pascal UCSD, na identificação de processos concorrentes. A razão dessa escolha está ligada à já existência no Sistema P de mecanismos de chamada e retorno de rotinas que podem ser aproveitados na ativação e desativação de processos. Observe-se que rotinas do tipo function ou segment function não podem ser relacionadas a processos concorrentes. Essa restrição existe em função do mecanismo de ativação dos processos concorrentes adotado (ver seção V.5). Com vistas a uma maior flexibilidade do Sistema P, nem todas as procedures e segment procedures são processos concorrentes. Para tanto, a palavra reservada concurrent mencionada na seção IV.1 é também utilizada para identificar processos concorrentes, que passam então a ser denotados por concurrent procedure ou concurrent segment procedure. Uma restrição que deve ser observada é que apenas rotinas do tipo concurrent podem ser chamadas de um par cobegin/coend, e só daí podem ser chamadas.

O programa Pascal do algoritmo IV.1 ilustra a introdução de concorrência no Sistema P. Esse programa avalia o mínimo de n inteiros (onde n > 2 é potência de 2) através de chamadas recursivas à rotina minint.

A execução do programa do algoritmo IV.1 para n = 8 realiza um grafo como o da figura IV.1; até a linha de simetria o grafo progride por chamadas recursivas a minint e a partir dela pelo retorno de cada uma dessas chamadas, com a avaliação dos mínimos dos elementos dois a dois. A simetria inerente a essa classe de grafos sugere a representação dos processos concorrentes como elementos de uma estrutura dinâmica em forma de árvore (SHAW[35]), na qual cada nodo representa um processo já ativado. Quando um processo executa um cobegin/coend, a árvore cresce através da criação de nodos filhos desse processo. Analogamente, quando um processo termina, a árvore decresce através da eliminação do nodo correspondente a esse processo.

Com relação a uma árvore de processos em um estado qualquer de sua evolução, pode-se afirmar que:

```
concurrent program mínimo;
const maxn = 64;
var inx, n, a1, a2: integer;
    a: array[1..maxn] of integer;
function min(a1, a2: integer): integer;
begin if a1 < a2 then min := a1 else min := a2 end;
concurrent procedure minint(i, j: integer; var amin: integer);
var a1, a2, m: integer;
begin
  if j = i + 1 then amin := min(a[i], a[j])
  else
    begin m := (j - i + 1) div 2;
      cobegin
        minint(i, m, a1);
        minint(m+1, j, a2)
      coend;
      amin := min(a1, a2)
    end
end;
begin
  read(n);
  for inx := 1 to n do read(a[inx]);
  inx := n div 2;
  cobegin
    minint(1, inx, a1);
    minint(inx+1, n, a2)
  coend;
  writeln(min(a1, a2))
end.
```

ALGORITMO IV.1 - Exemplo da Concorrência Introduzida no Sistema P

- os nodos não-folhas representam processos bloqueados à espera do término de seus filhos;
- os nodos folhas representam processos ativos ou bloqueados por alguma outra razão.

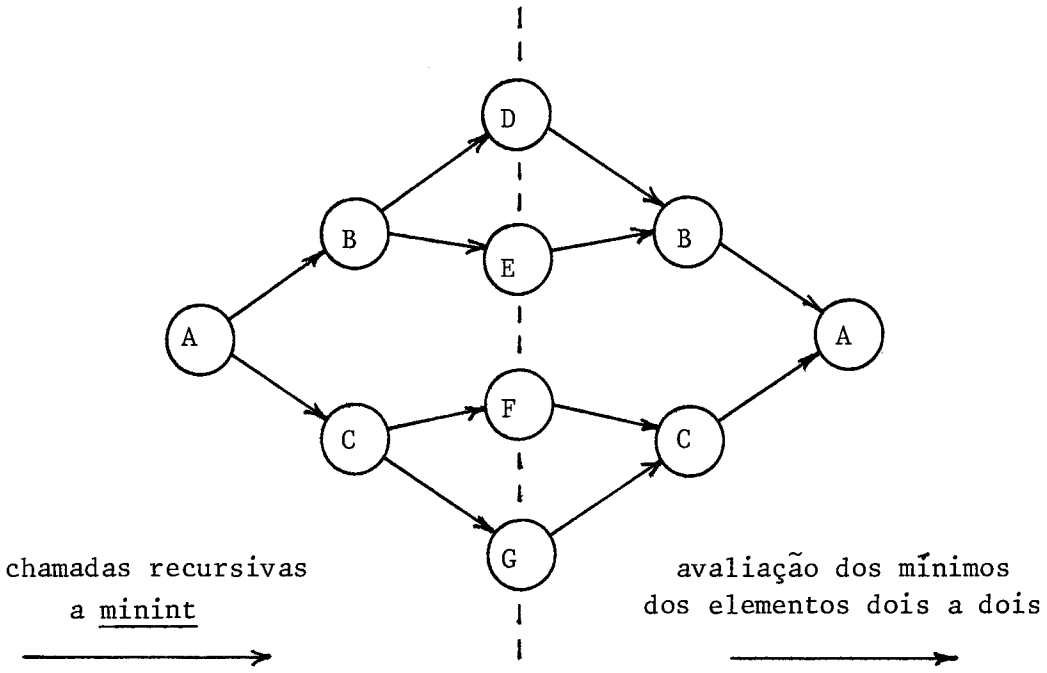


FIGURA IV.1 - Grafo de Precedência entre Processos Concorrentes do Sistema P

A árvore da figura IV.2 representa um dos possíveis estados da execução dos processos do grafo da figura IV.1.

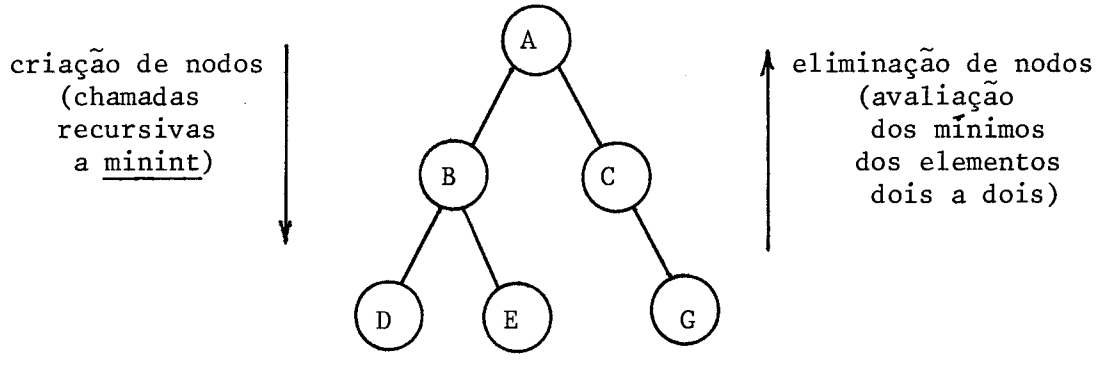


FIGURA IV.2 - Árvore de Processos no Sistema P

Deve ser observado que não existe em princípio nenhuma restrição quanto ao número de filhos de um determinado nodo da árvore. A única restrição que existe é quanto à formação da árvore: por ser construída através de comandos cobegin/coend, todos os filhos de um determinado nodo são gerados simultaneamente e o nodo pai só se torna ativo novamente ao voltar a ser folha, isto é, após o término da execução de todos os seus filhos.

Como foi observado na seção III.3, os comandos cobegin /coend prestam-se à realização de alguns casos de sinalização entre processos concorrentes, mais especificamente os casos em que os processos envolvidos são filhos do mesmo pai e a sinalização é bilateral. Com a adoção desses comandos para especificação da concorrência entre os processos do Sistema P, os casos de sinalização que não cumprem essas restrições devem ser realizados por outros meios. Na seção IV.4 a sinalização entre processos também é investigada.

IV.3- ARQUITETURA DA MÁQUINA P PARA PROCESSAMENTO CONCORRENTE

IV.3.1- PROCESSAMENTO EM CACTUS-STACK

Esta seção apresenta as principais alternativas para a arquitetura da máquina P para processamento concorrente. O material apresentados nos itens IV.3.1 a IV.3.4 está descrito em maior detalhe em MOTTA[32], sendo aqui analisado com a finalidade de possibilitar a compreensão dos próximos capítulos.

A representação dos processos concorrentes do Sistema P em árvore (seção IV.2) e a estrutura do stack da máquina P apresentada no item II.3.3 conduzem naturalmente a um processamento concorrente no Sistema P baseado na substituição do stack linear por um stack em árvore, ou cactus-stack (HOARE[24]), como ilustrado na figura IV.4. Nessa figura, cada nodo da árvore representa uma região de memória relativa ao processamento posterior à chamada de uma rotina do tipo concurrent (seção IV.2). De acordo com a organização do stack descrita no item II.3.3, cada um desses trechos de stack contém segmentos de dados e pilhas

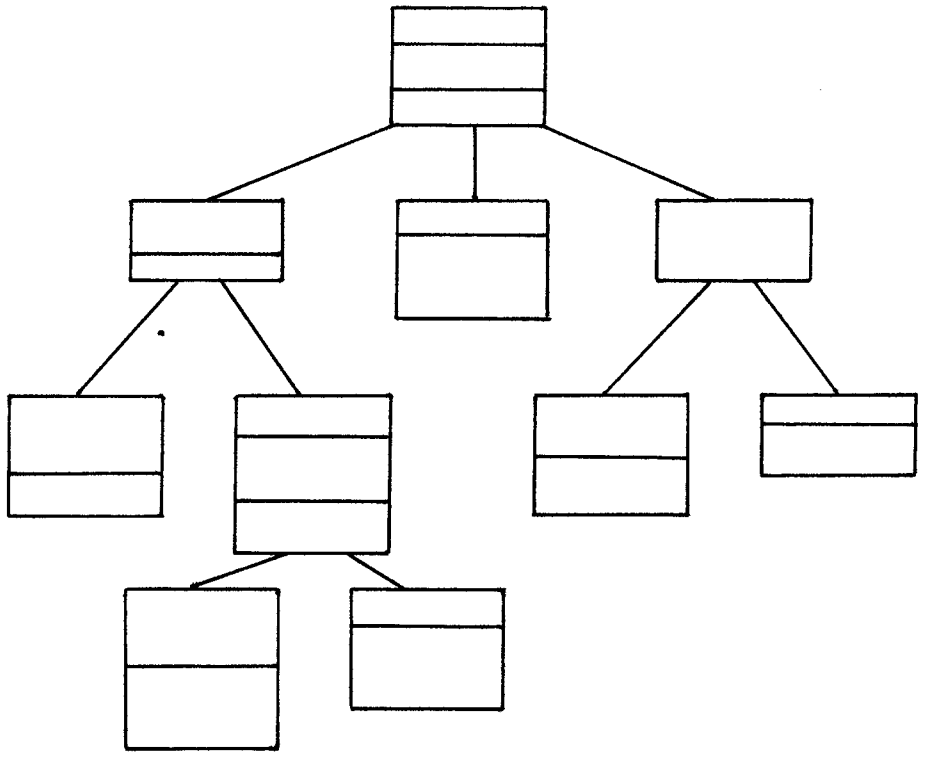


FIGURA IV.3 - Árvore de Stacks (Cactus-Stack)
para Processamento Concorrente no Sistema P

de avaliação, podendo também conter segmentos de código relativos a chamadas de rotinas do tipo segment (seção II.2). Na árvore de stacks da figura IV.3, os segmentos de dados, segmentos de código e pilhas de avaliação existentes no caminho de cada folha à raiz formam um stack linear como descrito no item II.3.3.

IV.3.2- ARQUITETURAS A MULTIPROCESSADORES

Na árvore de stacks da figura IV.3, as folhas representam processos ativos ou bloqueados por alguma razão que não seja o bloqueio inerente à execução de um par de comandos cobegin /coend (seção IV.2). A figura IV.4 mostra uma arquitetura distribuída para o processamento dessa árvore de stacks. De acordo com essa arquitetura, a máquina P é composta por um número qualquer de processadores virtuais que compartilham um módulo de me

mória por meio de uma via virtual de acesso à memória. É uma arquitetura fortemente conectada (seção III.1), em função do fato de que cada processo em execução pode ter a parte superior de seu stack linear em comum com outros processos também em execução. O termo "virtual" é utilizado na caracterização da arquitetura com o objetivo de realçar dois fatos: o primeiro é que cada processador é um interpretador de código P; o segundo é que a via de acessos à memória é utilizada para a busca de códigos P e dados manipulados por eles.

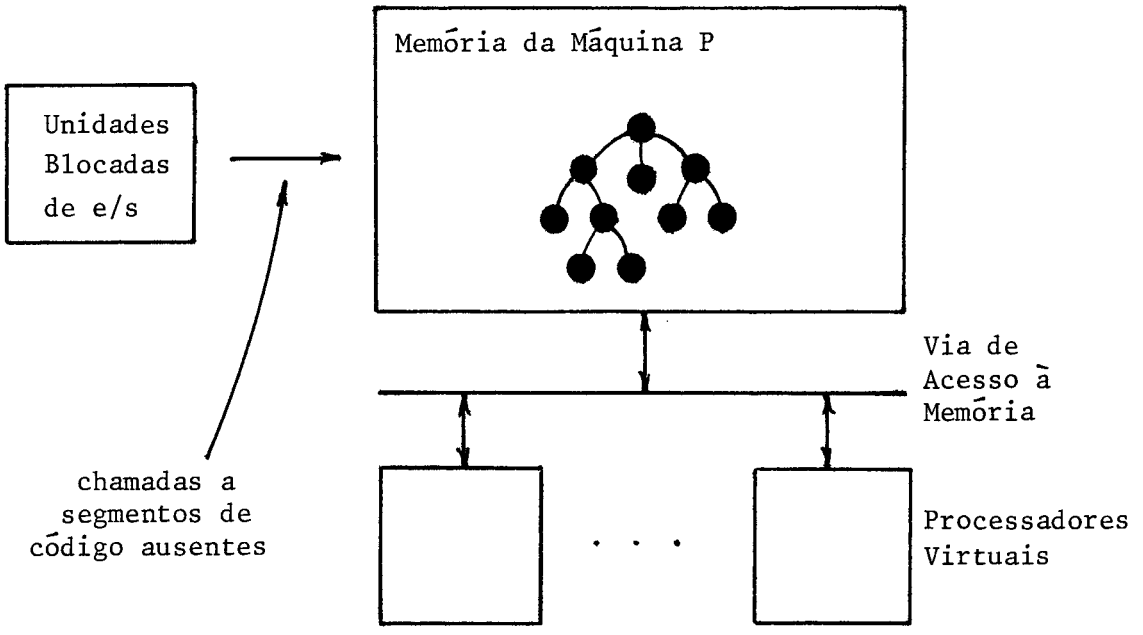


FIGURA IV.4 - Arquitetura a Multiprocessadores Virtuais com Toda a Memória Centralizada

Na arquitetura da figura IV.4 os processadores compartilham uma via de acesso à memória e competem por ela durante todo o processamento, uma vez que todo o código P a ser executado

e todos os dados necessários encontram-se no módulo compartilhado de memória. Dependendo das características de velocidade da via de acesso à memória ("bandwidth") e da taxa média de acessos efetuados pelos processadores, pode ocorrer por parte destes uma perda consideraável de eficiência causada pelo tempo dispendido à espera da liberação da via. Nos casos em que realmente possa ocorrer esse tipo de degradação do desempenho do sistema, existe a alternativa de fornecer a cada processador um módulo exclusivo de memória destinado a conter apenas a folha da árvore de stacks correspondente ao processo em execução naquele processador (figura IV.5). Observe-se que os dados existentes em uma

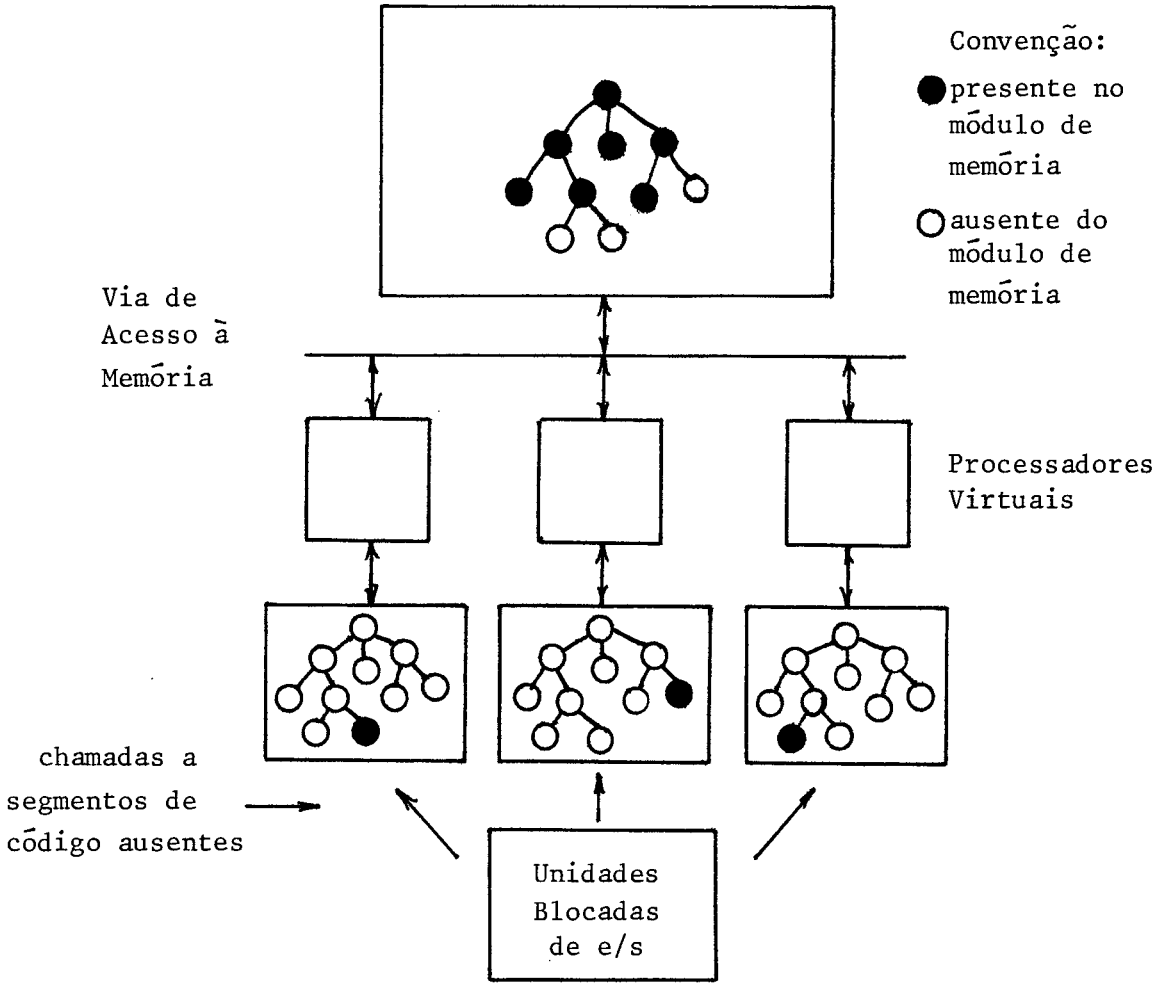


FIGURA IV.5 - Arquitetura a Multiprocessadores Virtuais com Memórias Distribuídas e Centralizada Organizadas Estaticamente

folha são exclusivos a ela, o que permite retirá-la da memória compartilhada. Dessa maneira, a freqüência de utilização da via compartilhada de acesso à memória pode cair bastante, desde que a maior parte das referências à memória dirijam-se aos módulos exclusivos. Durante a execução da árvore de stacks na arquitetura da figura IV.5 o código P do processo em execução em cada processador pode localizar-se na memória compartilhada ou na exclusiva. Esse último caso ocorre quando o processo é uma concurrent segment procedure chamada de forma não-recursiva, isto é, o código do processo não se encontra em seu stack linear , sendo então carregado. Os segmentos de dados utilizados na execução de um determinado processo também podem localizar-se na memória exclusiva do processador que o está executando (dados locais ao processo) ou na memória compartilhada (dados globais ao processo).

A forma alternativa de multiprocessamento em que cada processador possui seu módulo exclusivo de memória apresenta também algumas desvantagens. Uma delas é que sempre que um novo processo é selecionado para execução em um processador, a folha correspondente da árvore de stacks deve ser transferida da memória compartilhada para a memória exclusiva daquele processador. O processo que estava sendo ali executado, se não terminou, também deve ter seu stack transportado da memória local para a global. Esse processo de transferência de stacks entre as memórias pode tornar o escalonamento de processos para execução ineficiente. Uma outra desvantagem é quanto à freqüência com que um mesmo segmento de código pode ser carregado na memória exclusiva de processadores diferentes, a partir das unidades blocadas de e/s. Essa carga ocorre sempre que uma concurrent segment procedure é chamada de maneira não-recursiva, o que não seria necessário na arquitetura da figura IV.4, exceto da primeira vez. Dependendo dos dispositivos físicos em que são mapeadas as unidades blocadas de e/s, o tempo perdido nesses processos de carga pode prejudicar o desempenho.

O principal problema com essa segunda arquitetura, em que a memória é "semi-distribuída", é o fato de o escalonamen-

to dos processos para execução depender da transferência de stacks entre as memórias exclusiva e compartilhada. Essa necessidade existe em função da configuração estática da memória do Sistema; se fosse adotada uma configuração mais flexível em que a memória do Sistema se compusesse de módulos dinamicamente alocáveis por cada processador, então a necessidade de transportar stacks entre regiões de memória seria substituída pela modificação dinâmica da configuração da memória. A figura IV.6 ilustra

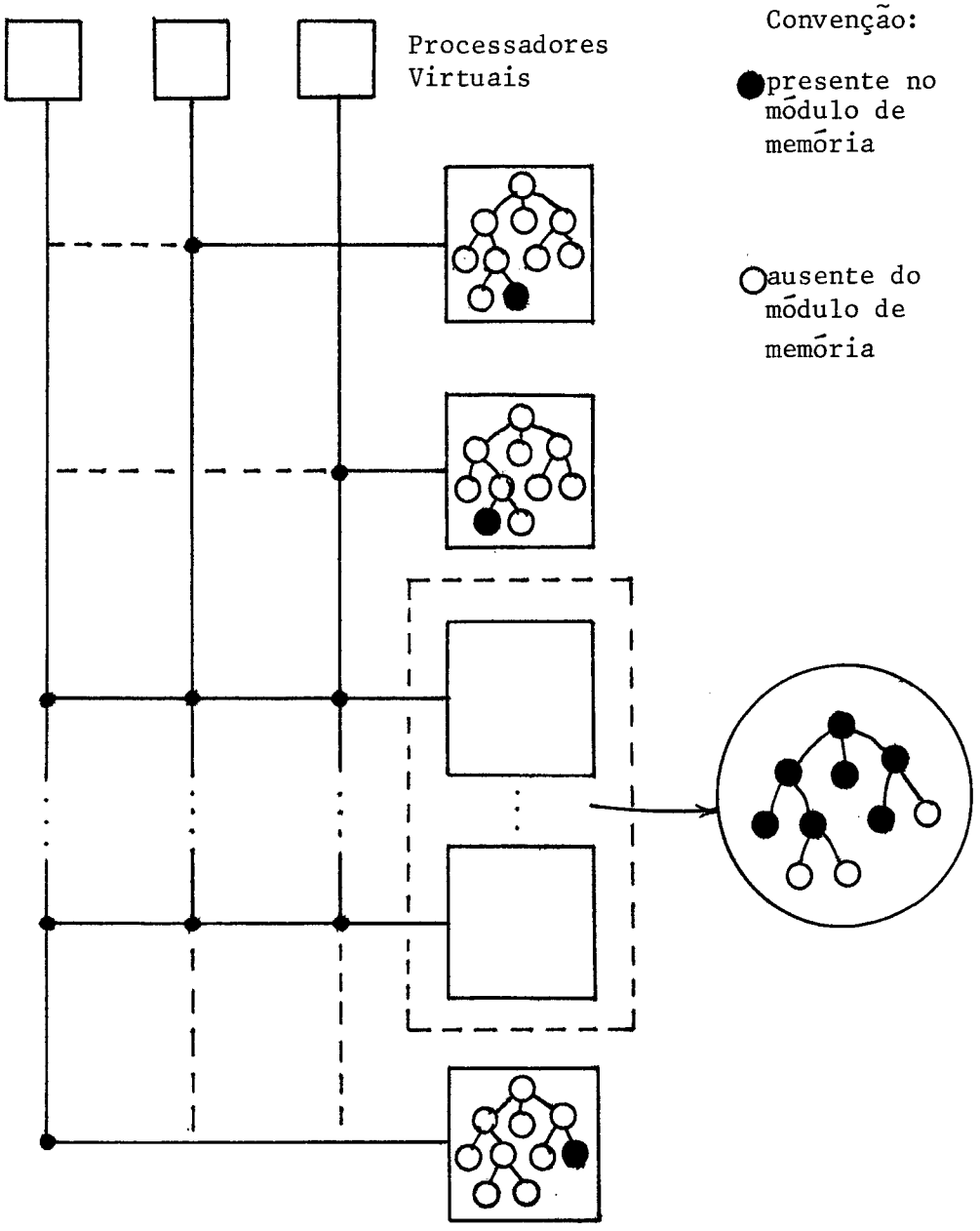


FIGURA IV.6 - Arquitetura a Multiprocessadores Virtuais com Memórias Distribuídas e Centralizada Organizáveis Dinamicamente

essa possibilidade: nela é mostrado um sistema de memória que englobaria vários módulos, todos em princípio endereçáveis por qualquer processador. Através de um esquema especial de mapeamento, um módulo de memória poderia tornar-se exclusivo a um determinado processador, assim como poderia ser compartilhado por mais de um. No caso da figura, cada um dos três processadores virtuais possuiria um módulo exclusivo de memória e teria acesso aos módulos que não fossem exclusivos aos outros processadores. Esses módulos compartilhados formariam a memória global do Sistema e problemas de contenção nas vias poderiam ocorrer apenas durante o endereçamento dessa memória. Em relação à arquitetura da figura IV.5, essa alternativa representaria uma evolução, pois permitiria a reconfiguração dinâmica do endereçamento aos módulos exclusivos e compartilhados de memória. Ela teria entretanto a necessidade de um hardware adicional, complexo e dispendioso em termos do estado atual da tecnologia, e que criaria também problemas de confiabilidade.

Com relação às arquiteturas a multiprocessadores sugeridas neste item, dois fatos devem ser observados: o primeiro é que cada processador virtual é potencialmente multiprogramável, porque o número de processos a serem executados (menor ou igual ao número de folhas da árvore de stacks) pode ser superior ao de processadores; o segundo fato é que todos os processadores do sistema são idênticos, no sentido em que qualquer um deles pode executar qualquer processo.

IV.3.3- PROBLEMAS DE ENDEREÇAMENTO NO CACTUS-STACK

Uma limitação importante da definição original da máquina P é o fato de o endereçamento da memória ser restrito a 64K bytes (item II.3.2). Para as arquiteturas a multiprocessadores propostas no item IV.3.2, esse limite de endereçamento seria aplicável a cada processador. No caso da arquitetura da figura IV.4 a memória é toda compartilhada pelos processadores, e fica limitada a esses 64K bytes. Nas demais arquiteturas em que cada processador possui também um módulo exclusivo de memória (figuras IV.5 e IV.6), a soma do tamanho de qualquer dessas me-

mórias exclusivas ao da memória compartilhada teria que ser menor ou igual a 64K bytes. Em qualquer dos casos, porém, o limite de 64K bytes resultaria insatisfatório, especialmente porque a parte compartilhada da memória conteria pelo menos todos os nodos não-folhas da árvore de stacks.

Para expandir o espaço de endereçamento de cada processador virtual poder-se-ia adotar uma forma de endereçamento relativo dentro de cada nodo da árvore de stacks. Esse esquema de endereçamento consistiria em associar um registrador de base a cada um desses nodos, de modo que endereços pudessem então ser calculados em relação a ele. Dessa forma, o limite de 64K bytes passaria a aplicar-se apenas dentro de cada nodo da árvore de stacks, constituindo o tamanho máximo da área ocupável por um processo (dados e pilhas de avaliação, no caso de concurrent procedures, além de código, no caso de concurrent segment procedures). Ainda nessa mesma linha de endereçamento relativo poder-se-ia separar os dados do código eventualmente contido em um nodo da árvore de stacks, e associar um registrador de base independente a cada uma dessas áreas. Como resultado, o máximo espaço de memória que os dados e o código de um processo poderiam ocupar passaria a limitar-se em 64K bytes cada.

Em qualquer dos casos, a fim de tornar viável essa nova forma de endereçamento, o interpretador do código P deveria ser capaz de realizar o endereçamento de código e dados através de deslocamentos relativos aos registradores de base. Para as áreas de dados, em particular, também o compilador teria que ser alterado, de modo a poder gerar instruções adequadas ao manuseio de alguns endereços absolutos, que incluiriam a especificação do registrador de base, além de um deslocamento dentro de 64K bytes. Observe-se que dessa forma tornar-se-iam relocáveis as áreas às quais se associou um registrador de base, através da simples modificação do conteúdo desse registrador.

IV.3.4- INTERRUPÇÃO DOS PROCESSADORES VIRTUAIS

O processador virtual do Sistema P, como descrito no capítulo II, não precisa utilizar interrupções em seu funciona-

mento, devido à sua caracterização como sistema mono-usuário para processamento seqüencial. Todavia, ao ser considerado o problema de estendê-lo ao processamento de programas concorrentes, as interrupções do processador virtual passam a ser de grande importância, pois constituem, como será visto no capítulo V, uma ferramenta fundamental para a reativação de processos bloqueados em certas circunstâncias. A figura IV.7 ilustra as principais características do sistema de interrupções necessário ao processamento concorrente aqui proposto para o Sistema P. Observe-se que o sinal de interrupção originado de uma unidade de e/s ou de um processador virtual propaga-se pelas vias de comunicação do Sistema e deve poder atingir todos os processadores virtuais.

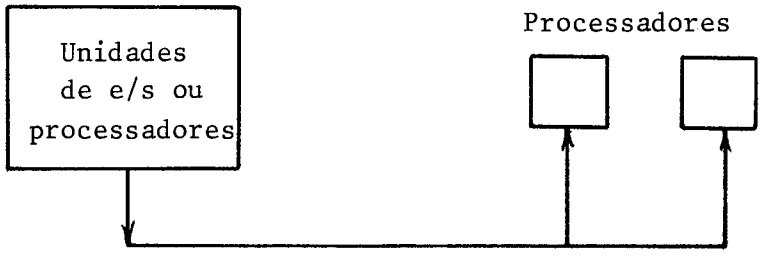


FIGURA IV.7 - Interrupção dos Processadores Virtuais

IV.4- COMPARTILHAMENTO DE RECURSOS E ESCALONAMENTO

O principal problema tratado nesta seção é o do gerenciamento da concorrência entre os processos do Sistema P. Essa concorrência existe entre as procedures do tipo concurrent em relação aos recursos da máquina P e a recursos do próprio programa concorrente.

Os recursos da máquina P compartilhados pelos processos concorrentes são o processador (ou processadores), a memória e as unidades de e/s. O gerenciamento do acesso a esses recursos

deve ser transparente aos processos.

Os recursos do programa concorrente compartilhados pelos processos são variáveis do programa. Essa concorrência existe como conseqüência de os stacks lineares dos processos possuírem trechos em comum. Um segmento de dados em um determinado ponto da árvore de stacks é acessível a toda a sub-árvore de que ele é raiz, dependendo de seu escopo. É responsabilidade do programador fazer com que os processos se sincronizem para o acesso a esses recursos, através de procedimentos manipulados no próprio texto do programa.

O problema da troca de sinais entre processos concorrentes também é tratado nesta seção. Como foi dito na seção III.3, os comandos cobegin/coend resolvem o caso particular de sinalização em que os processos envolvidos têm o mesmo pai e a troca de sinais é bilateral. Nos casos em que essas condições não se cumprem, a sinalização deve realizar-se de outra forma.

As ferramentas apresentadas nos itens III.3.2 a III.3.7 evoluíram no sentido de manter íntegros os recursos compartilhados através da garantia de exclusão mútua entre os processos e da delimitação de um conjunto de operações realizáveis. Ficou claro durante essa evolução que a possibilidade de realizar testes em tempo de compilação poderia evitar muitos erros de execução dependentes do tempo. Por exemplo, a construção apresentada no item III.3.2 para delimitar regiões críticas (cláusula with) possibilitou ao compilador verificar que não fossem realizadas referências a um recurso fora da região que seria executada com exclusão. Da mesma forma, o uso de monitores permitiu garantir em tempo de compilação que apenas certas operações poderiam ser realizadas sobre um recurso, pois este não seria acessível de fora de seu monitor, a não ser através de operações desse monitor.

Em termos de segurança do recurso compartilhado em arquiteturas fortemente conectadas, o monitor representa uma ferramenta satisfatória, sobre a qual foram propostos alguns aprimoramentos (como em KESSELS [30]) e extensões (como as "path expre

ssions" apresentadas em CAMPBELL & HABERMANN[4]). Para algumas aplicações, porém, o monitor revela-se inadequado, ora apresentando características excessivamente elaboradas (como para a simples troca de sinais entre processos), ora mostrando-se inflexível quanto ao escalonamento dos processos que competem por seu recurso.

As propostas apresentadas no item III.3.7 (linguagens Ada e Edison) revelam uma tendência a tornar menos inflexíveis as ferramentas para sincronização de processos concorrentes, evoluindo no sentido de definir um conjunto básico de operações (chamadas primitivas de sincronização) que possam ser utilizadas de diferentes maneiras, adaptando-se a uma classe ampla de problemas. A partir dessas operações, até mesmo monitores podem ser construídos, porém apenas em situações às quais o programador julgue que o conceito de monitor é aplicável. Como apontado em HANSEN[21] com relação à linguagem Edison, esse ganho em flexibilidade (e insegurança) pode eventualmente dar a impressão de um retrocesso, assim como pode sugerir novas perspectivas em metodologia de programação.

Para selecionar um conjunto de operações primitivas de sincronismo para estender o Sistema P a programação concorrente, procurou-se métodos que levassem em consideração os seguintes problemas:

- dois ou mais processos que concorrem pelo uso de um determinado recurso devem excluir-se mutuamente no tempo para o uso desse recurso, através da execução disciplinada das diversas regiões críticas sobre ele;
- durante a execução de sua região crítica sobre um determinado recurso, um processo deve possuir meios de se bloquear à espera do cumprimento de uma condição ainda não satisfeita sobre esse recurso, liberando seu direito de acesso exclusivo;
- ao terminar a execução de sua região crítica sobre um determinado recurso, um processo deve possuir meios de sinalizar o cumprimento de alguma condição sobre esse re -

curso pela qual algum outro processo possa estar esperando;
- dois processos em execução paralela devem poder se comunicar através da troca de sinais;
- formas ocupadas de espera devem ser evitadas, pois os recursos de processamento do sistema, em função do número indeterminado de processos, são potencialmente escassos;
- como a classe de programas que podem fazer uso de processamento concorrente é muito diversificada, as políticas adotadas para escalonamento de processos devem ser tão flexíveis quanto possível.

O problema da delimitação de regiões críticas foi solucionado através do uso de semáforos binários implementados com filas para bloquear processos. Assim, as operações P(S) e V(S), onde S é um semáforo binário, são usadas para demarcar as operações realizadas por um processo sobre um recurso. Observe-se que torna-se responsabilidade do programador assegurar que não sejam feitas referências a um recurso compartilhado fora da região delimitada pelas operações P(S) e V(S), sendo S o semáforo binário associado ao recurso. A incorporação de semáforos ao Sistema P acessíveis diretamente ao programador é também útil como mecanismo para processos concorrentes trocarem sinais entre si. Como foi ilustrado no item III.3.2, semáforos binários podem ser utilizados para bloquear um processo num determinado ponto até que outro processo atinja um outro certo ponto.

Um semáforo binário S é declarado no Pascal UCSD através de

```
var S: semaphore;
```

Sobre variáveis assim declaradas (e só sobre elas) podem ser executadas as operações P(S) e V(S). Sobre variáveis do tipo semaphore também pode ser aplicada a operação initsem(S,v), que atribui o valor v ao semáforo S. Pela natureza binária desse semáforo S, o valor v deve ser true ou false, sendo que a condição potencial de bloqueio está associada ao valor false.

Para possibilitar o sincronismo condicional dentro de regiões críticas, foram adotadas filas de eventos como as apresentadas no item III.3.3. Essas filas de eventos possuem uma certa redundância com os semáforos binários apresentados acima, uma vez que apenas estes poderiam ser suficientes para resolver os problemas de sincronização dentro de regiões críticas (DIJKSTRA[7]). Todavia, a solução através do uso exclusivo de semáforos binários tende a ser mais complexa e portanto menos legível do que a solução por filas de eventos, o que levou à sua incorporação.

Uma fila de eventos é declarada no Pascal UCSD por meio de

```
var Q: queue;
```

Sobre variáveis do tipo queue (e só sobre elas) aplicam-se as operações wait(Q,S) e signal(Q,S), onde Q é uma fila de eventos e S é o semáforo binário associado ao recurso manipulado pela região crítica dentro da qual a operação wait ou a operação signal é executada. Também é responsabilidade do programador garantir que essas operações sejam chamadas apenas das regiões críticas adequadas.

As operações wait e signal correspondem às operações await e cause definidas sobre filas de eventos no item III.3.3. Ao executar um wait(Q,S), um processo bloqueia-se na fila Q e libera o recurso ao qual o semáforo S está associado. A operação signal(Q,S) verifica se há processos bloqueados na fila Q: em caso afirmativo, um desses processos é reativado para continuar a execução de sua região crítica; em caso contrário, o recurso é liberado. Observe-se que, por poder liberar o recurso, a operação signal(Q,S) substitui a operação V(S) de fim da região crítica.

A figura IV.8 ilustra as possíveis formas de uma região crítica sincronizada por meio das operações P, V, wait e signal.

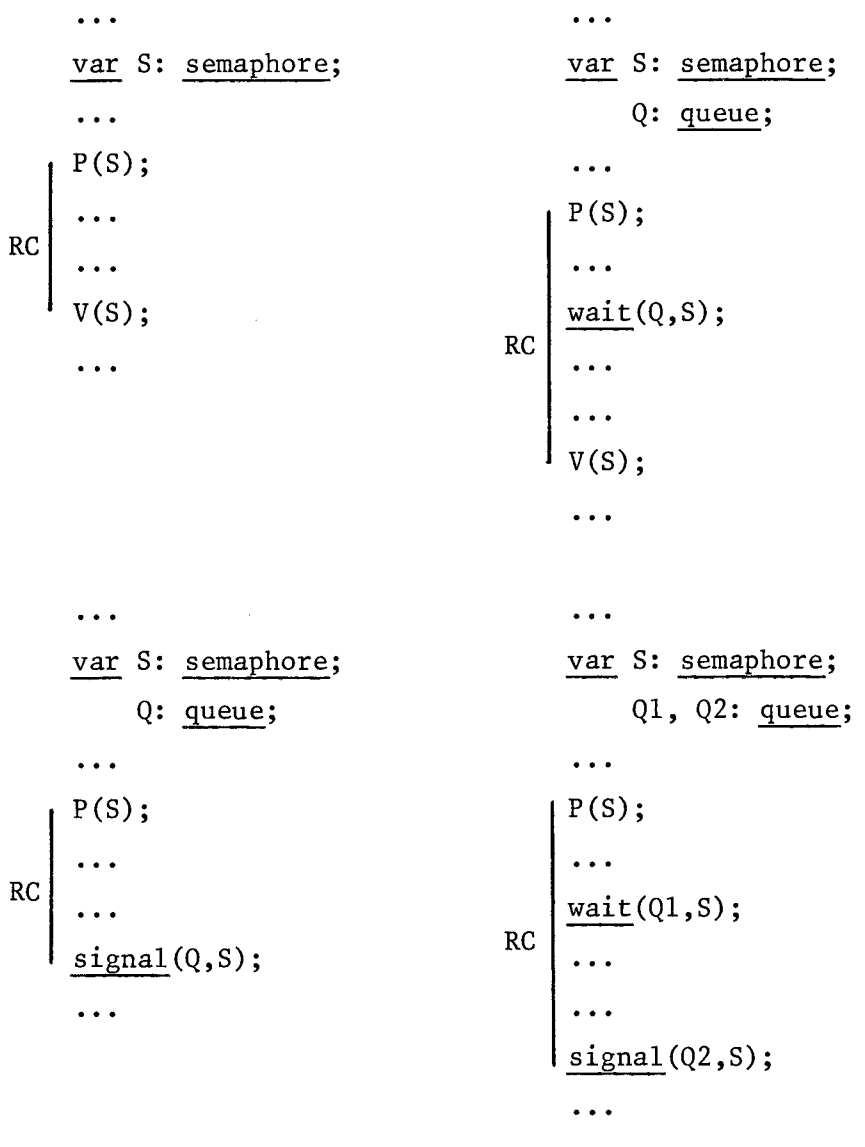


FIGURA IV.8 - Formas Possíveis de Regiões Críticas na Extensão Proposta sobre o Pascal UCSD

Foi também incorporada ao Pascal UCSD a função empty, importante em alguns problemas de sincronização. A função empty é uma função lógica que pode ser aplicada a variáveis do tipo queue ou do tipo semaphore. Quando aplicada a uma fila de eventos (empty(Q), Q fila de eventos), a função empty retorna o valor true se não há processos bloqueados na fila. Aplicada a um semáforo (empty(S), S semáforo), retorna true se não há processos bloqueados na fila do semáforo.

Com a utilização das ferramentas de sincronismo des -

critas acima, o usuário do Sistema P pode construir monitores para gerenciar recursos. Um monitor em Pascal UCSD é simplesmente um conjunto de rotinas que se excluem mutuamente no tempo através do uso de um semáforo binário associado ao recurso que elas gerenciam. Observe-se que utilizar essas rotinas adequadamente é uma disciplina de programação, uma vez que nada obriga o usuário a não fazer acessos ao recurso compartilhado sem passar por elas, porque não são feitas verificações em tempo de compilação. Dessa forma, a programação de monitores é apenas uma maneira de agrupar diversas regiões críticas sobre o mesmo recurso, ao invés de deixá-las espalhadas. Deve-se também notar que, pelo fato de o monitor não estar associado a um tipo abstrato (como em Pascal Concorrente, item III.2.5), a construção de monitores para gerenciar diversos recursos iguais deve utilizar parâmetros que especifiquem o recurso em questão, com a finalidade de evitar que monitores idênticos tenham que ser programados explicitamente.

O algoritmo IV.2 ilustra a solução do problema apresentado na seção B.1 através do uso de um monitor no Sistema P. Esse problema consiste em disciplinar o acesso de vários processos produtores e consumidores a um buffer circular. A solução apresentada no algoritmo IV.2 utiliza dois processos produtores e dois consumidores.

Resta ainda um problema a ser abordado, que é o das políticas de escalonamento dos processos concorrentes. As operações V(S) e signal(Q,S) apresentadas acima possuem a atribuição de reativar processos que possam estar bloqueados na fila do semáforo S ou na fila de eventos Q, respectivamente. Em princípio, porém, nada é conhecido sobre qual dos processos será reativado, no caso de haver mais de um. Todavia, como foi dito na seção III.3.2, uma política justa é liberar os processos na mesma ordem em que são bloqueados (FCFS - "first come, first served" ou FIFO - "first in, first out"), constituindo uma maneira de garantir que todos os processos são reativados em um tempo finito, a menos de problemas de bloqueio perpétuo ("deadlock"). Essa política foi adotada na extensão do Sistema P: as operações P e wait

```
concurrent program prodcons;
const maxbuf = 16; maxbuf1 = 15;
type item = record ... end;
(* início do monitor *)
var B: array[0..maxbuf1] of item;
    p, c: (0..maxbuf1);
    cont: (0..maxbuf);
    Sbuf: semaphore; algumcheio, algumvazio: queue;
procedure produz(x: item);
begin
  P(Sbuf);
  if cont = maxbuf then wait(algumvazio, Sbuf);
  B[p] := x; p := (p + 1) mod maxbuf; cont := cont + 1;
  signal(algumcheio, Sbuf)
end;
procedure consome(var x: item);
begin
  P(Sbuf);
  if cont = 0 then wait(algumcheio, Sbuf);
  x := B[c]; c := (c + 1) mod maxbuf; cont := cont - 1;
  signal(algumvazio, Sbuf)
end;
(* fim do monitor *)
concurrent procedure produtor;
var i: item;
begin
  while true do begin i := ...; produz(i) end
end;
concurrent procedure consumidor;
var i: item;
begin
  while true do begin consome(i); ... end
end;
begin p := 0; c := 0; cont := 0; initsem(Sbuf, true);
  cobegin produtor; produtor; consumidor; consumidor coend
end.
```

ALGORITMO IV.2 - Exemplo de Construção de Monitores na Extensão Proposta sobre o Pascal UCSD

encadeiam processos na ordem de chegada e as operações V e signal liberam o primeiro processo da fila. Mas há casos em que o risco de adiar indefinidamente a reativação de um processo pode ser aceitável. Em Sistemas de Controle em Tempo Real, por exemplo, é imprescindível que certas tarefas possam ser escalonadas com prioridades especiais. Por essa razão, foi adotado na extensão do Sistema P para programação concorrente um mecanismo de escalonamento por prioridades, do qual a política FCFS descrita anteriormente constitui o caso particular em que todos os processos têm a mesma prioridade.

Prioridades na extensão aqui proposta para o Sistema P são definidas como inteiros (não-negativos no intervalo de 0 a 100, no caso de prioridade para execução) e quanto menor esse inteiro maior é a prioridade. Sempre que um processo é criado (através dos comandos cobegin/coend), a ele é associada uma prioridade "default", cujo valor foi arbitrado em 50. A prioridade associada a um processo é utilizada em seu escalonamento para execução e pode ser modificada através da operação

setpriority(p)

que dá ao processo que a executa uma nova prioridade p. A política de escalonamento de processos para execução é tal que sempre estão em execução n dos processos mais prioritários, onde n é o número de processadores. Prioridades também podem ser utilizadas nas operações P e wait para especificar a ordem relativa de bloqueio dos diversos processos que competem pelo uso de um recurso. Assim, a operação P sobre um semáforo S pode ser utilizada em duas formas:

P(S) ou P(S,p)

valendo o mesmo para a operação wait sobre uma fila de eventos Q e um semáforo S:

wait(Q,S) ou wait(Q,S,p)

onde p é a prioridade utilizada na ordenação dos processos nas

filas respectivas. As operações V(S) e signal(Q,S) continuam liberando o primeiro processo da fila, que então corresponde ao mais prioritário. Se as operações P e wait forem utilizadas sem especificação de prioridade, será utilizado o valor de prioridade para execução associado ao processo em sua criação ou pela operação setpriority.

Deve ser finalmente observado que as ferramentas descritas nesta seção são acessíveis apenas a programas do tipo concurrent.

O apêndice C desta documentação fornece um resumo dessas novas características disponíveis na linguagem Pascal UCSD.

CAPÍTULO V

O NÚCLEO PARA PROCESSAMENTO CONCORRENTE NO SISTEMA P

V.1- INTRODUÇÃO

Neste capítulo são descritos os procedimentos destinados a dar suporte às características de programação concorrente introduzidas no Sistema P através da extensão proposta na seção IV.4 para o Pascal UCSD. A implementação aqui proposta destina-se às arquiteturas a multiprocessadores discutidas no item IV.3.2. Como foi dito nesse item, cada forma particular de arquitetura a multiprocessadores fortemente conectados possui determinadas desvantagens que afetam o desempenho ou a viabilidade do sistema: problemas de contenção em vias compartilhadas, dificuldades em realizar o escalonamento de processos para execução, problemas de caráter econômico e de confiabilidade, etc., atuam como fatores a serem considerados na escolha do tipo particular de arquitetura. Em MOTTA[32] podem ser encontradas considerações sobre esses problemas. De qualquer forma, o projeto do núcleo para dar suporte a processamento concorrente é até certo ponto dependente do tipo de arquitetura adotado: o método de escalonamento de processos para execução, por exemplo, pode ser influenciado pelos problemas mencionados acima. Na discussão das próximas seções, portanto, alguns pontos são abordados de uma maneira mais genérica, em função de sua dependência sobre detalhes da arquitetura utilizada.

Um ponto importante que deve ser observado é a situação do núcleo proposto neste capítulo com relação aos encontrados na literatura. Em HOLT et al [27] existe uma proposta particularmente interessante para construção de um núcleo para dar suporte à linguagem CSP/k em arquiteturas a multiprocessadores. Esse núcleo é muito simples e compõe-se de um conjunto de procedimentos que manipulam dados globais do sistema. A proposta baseia-se na existência de um processador virtual dedicado à

execução do núcleo, e que é ativado quando algum processo ou dispositivo periférico o solicita. Pelo fato de que esse processador virtual não pode ser interrompido, a integridade dos dados que o núcleo manipula fica automaticamente garantida. Nesse esquema, cada processador divide seu tempo entre executar um processo qualquer e simular o processador virtual que executa o núcleo, o que só pode ser feito por um processador de cada vez. A proposta deste capítulo para um núcleo do Pascal UCSD estendido difere da proposta encontrada em HOLT et al [27] essencialmente no sentido em que nem todos os procedimentos do núcleo são regiões críticas sobre dados globais compartilhados, como os procedimentos para inicializar processos dinamicamente, por exemplo. A menos de certos trechos que constituem realmente regiões críticas, o núcleo pode estar sendo executado simultaneamente por mais de um processador. A execução do núcleo deve, então, ser entendida como uma continuação da execução do processo que o chamou; as seções críticas do núcleo, em particular , devem ser executadas com exclusão mútua pelos diversos processos, de uma forma comparável à execução de processos dentro de monitores (item III.2.5). Assim, os processadores do Sistema devem ser estendidos a fim de que possam dar suporte a certas peculiaridades do núcleo como, por exemplo, salvar e restaurar status (ver apêndice D e MOTTA[32] para descrição dessas extensões). O procedimento dispose para remover variáveis alocadas dinamicamente faz parte dessas extensões e elimina os problemas apresentados no item II.3.4.

O núcleo do Pascal UCSD estendido para processamento concorrente é um conjunto de rotinas para as quais são geradas chamadas apenas durante a compilação de programas do tipo concurrent, como descritos na seção IV.1. Para fins de especificação foi adotada a linguagem Pascal, e é nela que se encontram escritos os algoritmos apresentados neste capítulo. As rotinas do núcleo desempenham as funções de escalonamento dos processos concorrentes do Sistema P. Além do escalonamento para execução, o núcleo também escalona os processos para o acesso a recursos compartilhados, através da realização das operações sobre semáforos e filas de eventos (seção IV.4). Também é atribuição do núcleo o gerenciamento transparente de recursos do Sistema como unidades de e/s e, até certo ponto, processadores

e memória. A seção V.2 descreve a representação de cada processo dentro do núcleo e as estruturas de filas e árvore de processos. São também apresentadas algumas filas especiais: a fila ready de processos prontos para execução à espera de um processador disponível, e uma fila exclusiva de cada processador, apontada por running, que identifica o processo em execução naquele processador. Filas e árvore de processos são exemplos de algumas variáveis do núcleo compartilhadas pelos diversos processos e que devem, por essa razão, ser manipuladas com exclusão mútua. A exclusão mútua para execução de certos trechos do núcleo é assegurada pela introdução no Sistema P dos conceitos de interrupção do processador virtual e de espera ocupada sobre flags (item III.3.2) descrita na seção V.3. A seção V.4 descreve as operações básicas para escalonamento dos processos concorrentes para execução e propõe uma forma de implementar o escalonamento por prioridades descrito na seção IV.4 com base apenas em interrupções periódicas de "time-slice". São descritas essas interrupções e a realização da operação setpriority. Na seção V.5 são descritos a inicialização e o término de programas e processos concorrentes, bem como a implementação dos comandos cobegin/coend. Conforme foi dito na seção IV.4, foram introduzidos semáforos e filas de eventos no Sistema P para disciplinar o compartilhamento de recursos pelos processos concorrentes. A implementação de semáforos binários e filas de eventos no núcleo do Sistema P e das operações realizáveis sobre eles encontram-se descritas nas seções V.6 (semáforos binários) e V.7 (filas de eventos). A seção V.8 descreve como são gerenciadas as operações de entrada e saída descritas no item II.4.7 no caso de programas concorrentes.

V.2- REPRESENTAÇÃO DOS PROCESSOS, ÁRVORE E FILAS

Cada processo no Sistema P estendido para processamento concorrente é representado por uma estrutura definida pelo seguinte tipo abstrato em Pascal:

```
type nodoptr = ↑nodo;
     nodo = record pai, prox: nodoptr; ... end;
```

Nessa representação aparecem apenas os campos que são utilizados na formação da árvore de processos e das filas de processos. Os demais campos que compõem a estrutura representativa de um processo serão apresentados ao longo do texto, à medida que se fizerem necessários. Como foi dito na seção IV.2 , os nodos de uma árvore de processos pertencem a duas categorias: processos bloqueados à espera do término de seus filhos (nodos não-folhas) e processos ativos ou bloqueados por alguma outra razão (nodos folhas). Sobre essa última classe de processos, pode-se afirmar que encontram-se encadeados em alguma fila, como será visto na seção V.4. Assim, se p tem o tipo nodoptr definido acima, então (ver figura V.1):

- p↑.pai é um apontador para o nodo que representa o processo bloqueado à espera que o processo representado por p↑ termine. Se p↑ é a raiz da árvore, então p↑.pai = nil;
- se p↑ é uma folha da árvore, então p↑.prox aponta para o nodo folha que representa o próximo processo na fila em que p↑ se encontra. Se p↑ é o último processo da fila ou não é uma folha, então o valor de p↑.prox é indeterminado.

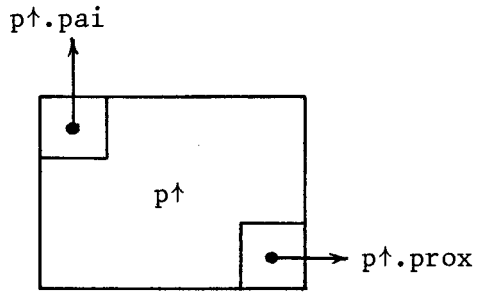


FIGURA V.1 - Nodo Representativo de um Processo

O núcleo para processamento concorrente manipula uma variável especial do tipo nodoptr: a variável raiz, que aponta para a raiz da árvore de processos durante a execução de um pro

grama concorrente.

As filas utilizadas pelo núcleo são variáveis do tipo:

```
type fila = record
              vazia: boolean;
              primeiro,
              último: nodoptr
            end;
```

Se f é uma variável do tipo fila, então f.vazia tem valor true se não há nenhum processo encadeado em f e valor false em caso contrário. Se f.vazia = false, então f.primeiro e f.último apontam respectivamente para o primeiro e para o último processos da fila. Se f.vazia = true, então f.primeiro e f.último possuem valores indeterminados. A operação básica para manipulação de variáveis do tipo fila é a operação transf definida como

```
procedure transf(var fonte: fila; var dest: fila; p: integer);
```

Essa operação transfere o processo representado por fonte.primeiro↑ para a fila dest, encadeando-o nessa fila em uma ordem crescente dada por p. Ao mesmo tempo, registra o valor de p em um outro campo da estrutura de tipo nodo, chamado prioridade, a fim de possibilitar o posicionamento adequado de processos a serem futuramente inseridos na mesma fila.

O núcleo manipula algumas variáveis especiais do tipo fila: uma fila chamada ready, que contém processos prontos para serem executados à espera de algum processador livre, e uma fila apontada por running, exclusiva de cada processador, que identifica o processo que nele está sendo executado.

V.3- O NÚCLEO COMO REGIÃO CRÍTICA

Durante a execução de um programa concorrente, as rotinas do núcleo podem ser chamadas por qualquer processo que esteja ativo e seja escalonado para execução. Dessa forma, al-

guns dados manipulados por essas rotinas são também dados compartilhados pelos processos, sendo as rotinas regiões críticas sobre esses dados. Alguns exemplos de dados indiretamente compartilhados pelos processos são filas, semáforos, etc., e as rotinas que os manipulam devem ser executadas com exclusão mútua pelos diversos processos que podem chamá-las. Exclusão mútua a esse nível é estabelecida através da operação busywait sobre flags mencionada no item III.2.2 (algoritmo III.2). Com um flag f (inicializado com false), por exemplo, uma região crítica RC pode ser protegida por

```
busywait(f); RC; f := false;
```

Também foi dito no item III.2.2 que, com o objetivo de minimizar a perda de eficiência devida à espera ocupada, pode-se executar o trecho acima com interrupções inibidas, garantindo que o flag é liberado o mais rápido possível:

```
inibe interrupções;
busywait(f); RC; f := false;
habilita interrupções;
```

Além disso, deve ser observado que a inibição de interrupções para executar as regiões críticas do núcleo é fundamental, pois a ocorrência de uma interrupção durante a execução do núcleo causaria uma tentativa de executá-lo de forma aninhada, provocando uma situação de "deadlock" (ver capítulo VI).

Ao estender o Sistema P para processamento concorrente foram adotadas as operações de espera ocupada sobre flags , habilitação e inibição de interrupções como forma de garantir acesso exclusivo dos processos concorrentes a alguns dados do núcleo. Uma região crítica sobre esses dados é então precedida pela rotina entraRC e sucedida pela rotina deixaRC, ambas apresentadas no algoritmo V.1. As operações disable, enable e busywait são implementadas como rotinas standard (item II.3.2) e têm respectivamente as funções de inibir as interrupções da má

```
procedure entraRC;
begin
  disable;
  busywait(fmutex)
end;

procedure deixaRC;
begin
  fmutex := false;
  enable
end;
```

ALGORITMO V.1 - Entrada e Saída das Regiões Críticas do Núcleo

quina virtual, habilitá-las e realizar espera ocupada sobre flags. A variável fmutex tem o tipo boolean e é utilizada como flag para controlar a exclusividade durante a execução das seções críticas do núcleo.

V.4- ESCALONAMENTO PARA EXECUÇÃO

Os processos concorrentes do Sistema P dividem-se, como já foi dito, no que se pode chamar de processos representados por nodos não-folhas e processos representados por nodos folhas na árvore de processos. Estes últimos, em particular, encontram-se em um dos três seguintes estados (figura V.2): prontos para execução, bloqueados em alguma fila de semáforo ou de eventos, ou sendo executados. Os processos prontos para execução encontram-se encadeados na fila ready mencionada na seção V.2 e estão à espera de processadores livres para executá-los. Cada processo em execução encontra-se em uma fila exclusiva ao processador que o está executando. Como foi dito na seção V.2 , essa fila é apontada por uma variável global definida por

```
var running: ↑fila;
```

Apesar de existir um único apontador running no Siste

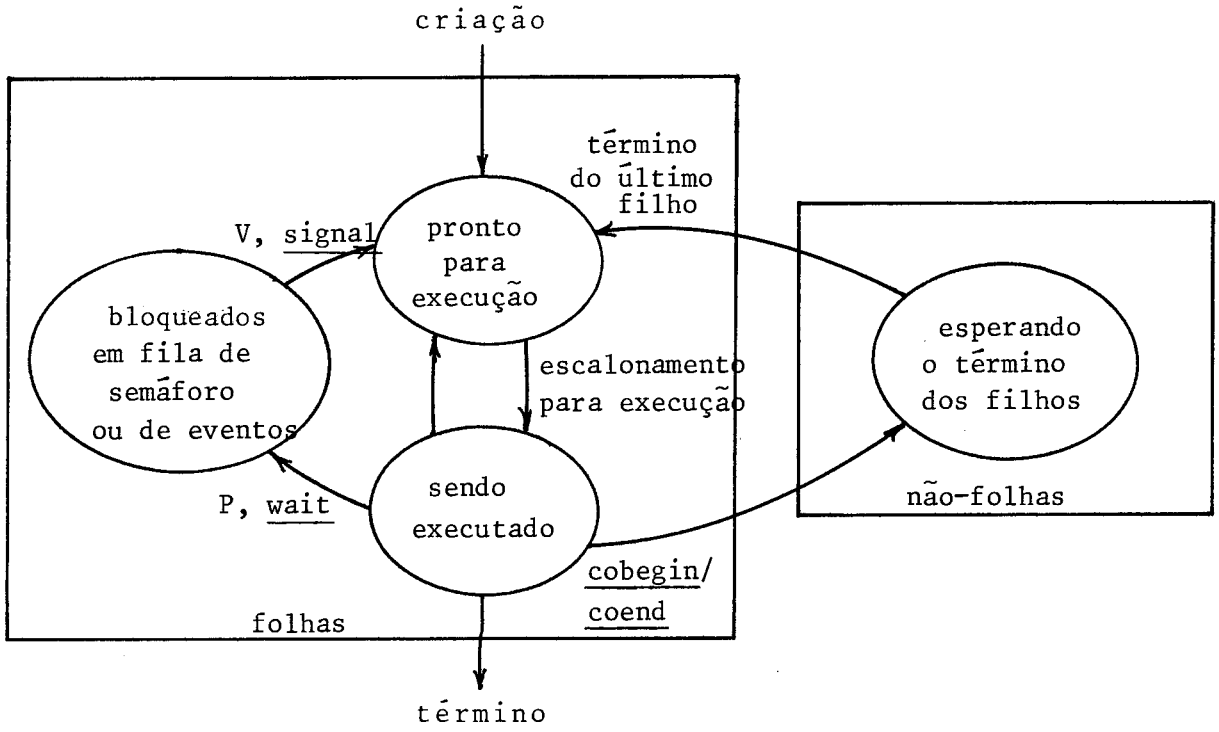


FIGURA V.2 - Estados dos Processos Concorrentes e Transições entre Eles

ma, a fila running↑ é exclusiva de cada processador. Isto é conseguido fazendo-se com que, ao ser ativado o Sistema, running aponte para um endereço de memória local a cada processador virtual (interpretador). Assim, uma referência a running↑ feita por um determinado processador diz respeito à área de endereço running daquele processador, exclusiva portanto.

Deve ser observado com relação às filas running↑ que elas sempre possuem um processo (running↑.vazia = false), ainda que não esteja sendo executado um programa concorrente ou que o programa concorrente em execução não esteja utilizando todos os processadores. Para tanto, cada processador cria, quando de sua ativação, um processo com prioridade para execução superior a 100 (o que o classifica como menos prioritário que todos os processos) destinado a ocupar o tempo de algum processador quando este não possuir nada mais prioritário (processo de um programa concorrente) para executar. Observe-se que a incorporação desses processos "ociosos" nas filas do Sistema permite tratar todas as situações de escalonamento de uma maneira uniforme, ainda que não haja processos de programas do usuário ativos. Esses proces

sos não pertencem à árvore de processos concorrentes e estão sempre encadeados na fila ready ou em alguma das filas running↑.Isso pode ser observado na figura V.3, que ilustra a estrutura da árvore e filas adotadas para processamento concorrente no Sistema P. O exemplo apresentado é de uma arquitetura com apenas dois processadores, o que implica na existência de duas filas running↑ e de dois processos isolados da árvore (no caso encadeados no final da fila ready, pois os processadores encontram-se ocupa -

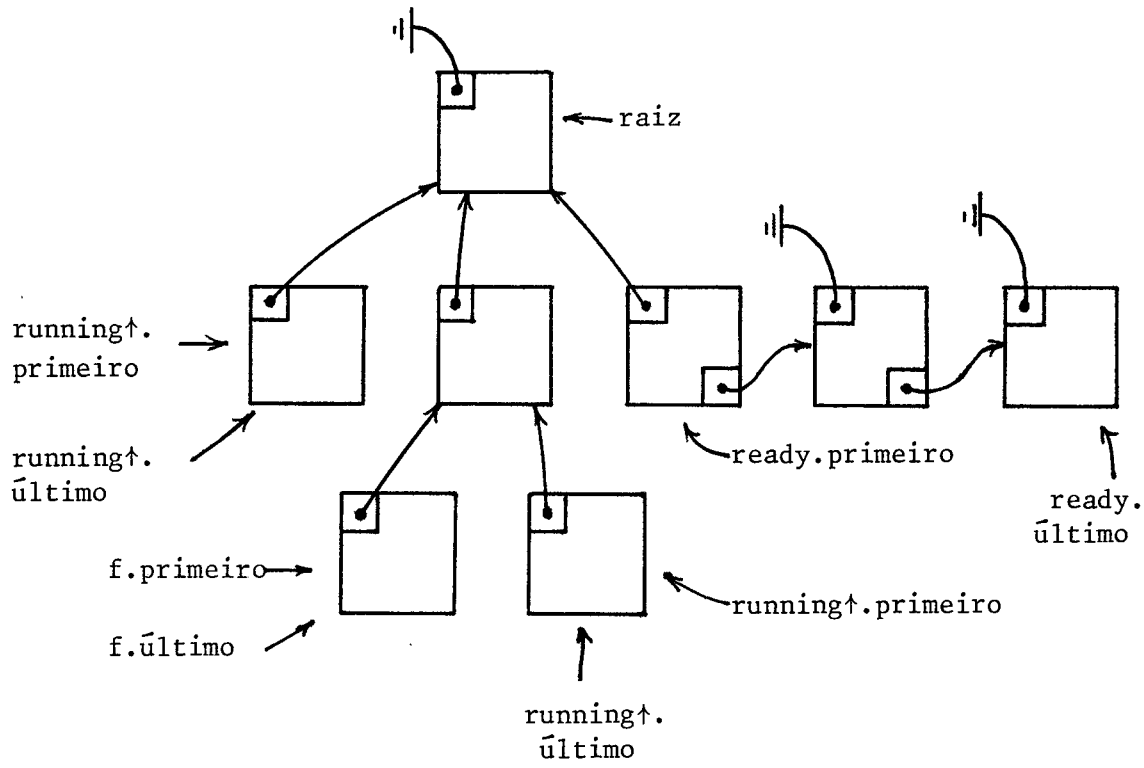


FIGURA V.3 - Exemplo de uma Árvore de Processos

dos com o processamento de tarefas mais prioritárias). A folha da árvore que não se encontra na fila ready ou nas filas running↑ está encadeada em uma outra fila qualquer f (fila de semáforo ou de eventos).

A operação básica de escalonamento de processos concorrentes para execução é a rotina suspende apresentada no algoritmo V.2. Conforme será visto nas próximas seções, essa operação é

```
procedure suspende(var f: fila; p: integer);
var ptr: nodoptr;
begin
  with running↑ do
    begin ptr := primeiro;
      if ptr <> nil then
        transf(running↑, f, p);
      transf(ready, running↑, 50);
      raizexec := primeiro = raiz;
      schedule(ptr, primeiro)
    end
end;
```

ALGORITMO V.2 - Operação Básica de Escalonamento

chamada sempre que um processo precisa bloquear-se por alguma razão. A rotina suspende realiza o bloqueio colocando o processo (running↑.primeiro↑) na fila f em uma ordem dada por p, e passa então à escolha de um novo processo a ser executado naquele processador. Um caso particular que deve ser observado é o caso em que running↑.primeiro = nil. Esse caso corresponde ao pedido de suspensão de um processo que terminou sua execução e portanto não deve ser transferido para nenhuma fila.

A variável raizexec utilizada pela operação suspende é uma de duas variáveis do tipo boolean manipuladas pelo núcleo: a outra é a variável concexec, que tem o valor true apenas quando está sendo executado um programa concorrente. Quando concexec = true, a variável raizexec tem o valor true se a raiz da árvore de processos está sendo executada em algum processador (running↑.primeiro = raiz). Pode-se afirmar que só existe concorrência quando concexec = true e raizexec = false. Esse fato será utilizado para o gerenciamento de e/s concorrente (ver seção V.8).

Como pode ser observado no algoritmo V.2, o procedimento suspende termina por executar uma chamada a schedule, passando como parâmetros respectivamente o endereço do descri-

tor do processo a ser suspenso e o do processo a ser ativado para execução. A função da rotina schedule, implementada como rotina standard no interpretador, é salvar em ptr↑ o contexto do processo que estava sendo executado e restabelecer o do processo que foi reativado, a partir de running↑.primeiro↑. No caso particular em que ptr = nil, ou seja, o processo ptr↑ deve ser suspenso por ter terminado, a rotina schedule libera a região de memória ocupada por esse processo. Uma característica de schedule é que ela sempre executa um deixaRC ao fim de sua operação.

Como pode ser visto na figura V.2, o escalonamento de processos para execução inclui também a passagem de processos de uma fila running↑ para a fila ready, isto é, a suspensão da execução de um determinado processo sem razão aparente. Essas suspensões são necessárias como forma de realizar o escalonamento para execução por prioridades proposto na seção IV.4. Conforme foi dito na seção V.1, o núcleo proposto neste capítulo realiza esse escalonamento através de interrupções de "time-slice" somente. Essas interrupções são locais a cada processador, o que faz com que os diversos processadores do Sistema sejam interrompidos de maneira assíncrona, evitando competições muito intensas pela via de acesso à memória. Cada processador, ao ser interrompido, executa o procedimento do algoritmo V.3, que suspende o processo running↑.primeiro↑ no caso de o primeiro processo da fila ready ser mais prioritário (ou ter mesma prioridade) que ele. O campo schedprior que aparece nesse algoritmo pertence ao nodo representativo de um processo e armazena sua prioridade para execução. Observe-se que não se está garantindo que a cada instante estejam em execução n dos processos mais prioritários (sendo n o número de processadores); o que se garante é que um processo inserido na fila ready esperará no máximo um período de "time-slice" para ser executado por algum processador (supondo que ele é mais prioritário, ou de mesma prioridade, que algum dos processos em execução e que todos os períodos de "time-slice" são iguais) . Dessa forma, pode haver aplicações particulares, especialmente para operação em tempo real, que não sejam resolvidas apenas com interrupções de "time-slice" e que exijam a possibilidade de se escalonar processos para execução com mais urgência. No capítulo VI são fornecidas algumas sugestões de como isso poderia ser conseguido através da utilização de interrupções originadas

dos próprios processadores (item IV.3.4). Observe-se que o efeito das interrupções de "time-slice" propostas é não só o de assegurar a execução de processos mais prioritários, como também de fazer com que processos de mesma prioridade compartilhem os processadores, revezando-se em seu uso de forma cíclica.

```
procedure sliceinterrupt;
begin entraRC;
  with running↑.primeiro↑ do
    if schedprior >= ready.primeiro↑.schedprior then
      suspende(ready, schedprior)
    else deixaRC
end;
```

ALGORITMO V.3 - Tratamento de Interrupções de "Time-Slice"

Como foi dito na seção IV.4, cada processo pode alterar sua própria prioridade através da execução do procedimento setpriority. Esse procedimento encontra-se descrito no algoritmo V.4. Observe-se que se a nova prioridade atribuída a runnin-

```
procedure setpriority(p: integer);
begin
  with running↑.primeiro↑ do
    if (p <> schedprior) and (p in [0..100]) then
      begin entraRC;
        schedprior := p;
        if p <= ready.primeiro↑.schedprior then
          deixaRC
        else
          suspende(ready, p)
      end
end;
```

ALGORITMO V.4 - Alteração de Prioridade para Execução

g↑.primeiro↑ torna-o menos prioritário que ready.primeiro↑, então ele deve ser suspenso.

V.5- EXECUÇÃO DE PROGRAMAS CONCORRENTES

O código P gerado na compilação de um programa concorrente inicia com uma chamada à rotina initconcpgm e termina com uma chamada à rotina endconcpgm. Essas rotinas encontram-se implementadas no núcleo e funcionam conforme os algoritmos V.5 e V.6, respectivamente.

```
procedure initconcpgm;
begin
  inicializa unitable;
  concexec := true;
  raizexec := true;
  new(raiz);
  with raiz↑ do inicializa campos de raiz↑;
  with running↑ do
    begin vazia := false;
      primeiro := raiz; último := raiz
    end
end;
```

ALGORITMO V.5 - Inicialização de um Programa Concorrente

A inicialização de unitable no algoritmo V.5 compreende a inicialização com valores apropriados dos semáforos, filas e demais variáveis utilizados no gerenciamento do acesso concorrente às unidades de e/s e estruturas de diretórios (ver seção V.8, onde a variável unitable é apresentada).

Dentro de um programa concorrente, o código P gerado na compilação de um processo concorrente (rotina do tipo concurrent procedure ou concurrent segment procedure) é também precedido e sucedido por chamadas a duas rotinas especiais do núcleo, as ro

```
procedure endconcpgm;
begin
  dispose(raiz);
  concexec := false;
  running↑.vazia := true
end;
```

ALGORITMO V.6 - Término de um Programa Concorrente

tinas initprocess e endprocess (algoritmos V.7 e V.9, respectivamente). A rotina initprocess é executada assim que o processo é chamado dentro de um par de comandos cobegin/coend. São utili

```
procedure initprocess;
var filadummy: fila; auxptr: nodoptr;
begin
  with running ↑.primeiro↑ do
    begin ntasks := ntasks + 1;
      new(auxptr);
      with auxptr↑ do
        begin pai := running↑.primeiro;
          inicializa demais campos
        end;
      with filadummy do
        begin vazia := false;
          primeiro := auxptr; último := auxptr
        end;
      transf(filadummy, filhos, 50);
      alloc(auxptr)
    end
end;
```

ALGORITMO V.7 - Inicialização de um Processo Concorrente

zados mais dois campos do nodo representativo de um processo ,

os campos ntasks e filhos. O campo ntasks é utilizado para registrar o número de processos ativados simultaneamente no cobegin/coend e representa o número de processos que running↑.primeiro↑ precisa esperar terminar para prosseguir sua execução.

```
procedure coend;
var filadummy: fila;
begin
  with running↑.primeiro↑ do
    if ntasks > 0 then
      begin entraRC; filadummy.vazia := true;
        repeat
          transf(filhos, ready, filhos.primeiro↑.schedprior)
        until filhos.vazia;
        suspende(filadummy, 50)
      end
end;
```

ALGORITMO V.8 - Ativação Simultânea de Processos Concorrentes

O campo filhos possui o tipo fila e é utilizado para encadear os processos que devem ser ativados simultaneamente. A função de initprocess é bloquear o processo recém-ativado até que tenham sido ativados todos os outros processos chamados dentro do mesmo par cobegin/coend. A rotina alloc chamada ao final de initprocess tem o objetivo de reservar uma região de memória para o recém-criado processo e de causar o retorno da execução para o comando cobegin/coend. Ao terminar a execução desse comando é chamada a rotina coend do algoritmo V.8, que encadeia todos os processos da lista running↑.primeiro↑.filhos na fila ready, ativando-os para serem executados quando houver processadores disponíveis. Observe-se que o processo running↑.primeiro↑ deixa de pertencer às filas globais do Sistema, ficando em uma condição implícita de bloqueio dada por sua caracterização como nodo não-folha da árvore de processos. Além disso, os

processos recém-inseridos na fila ready poderão, dependendo da situação relativa de suas prioridades para execução, ter suas chances de serem executados nas próximas interrupções de "time--slice" dos diversos processadores.

```
procedure endprocess;
var filadummy: fila;
begin
  with running↑ do
    begin entraRC;
      with primeiro↑.pai↑ do
        begin ntasks := ntasks - 1;
          if ntasks = 0 then
            begin filadummy.vazia := false;
              filadummy.primeiro := primeiro↑.pai;
              filadummy.último := primeiro↑.pai;
              transf(filadummy, ready, primeiro↑.pai↑.schedprior)
            end
        end;
      filadummy.vazia := true;
      dispose(primeiro);
      suspende(filadummy, 50)
    end
end;
```

ALGORITMO V.9 - Término de um Processo Concorrente

A execução de cada processo termina com a rotina endprocess do algoritmo V.9, que remove a folha da árvore de processos correspondente ao processo que terminou e reativa seu pai, se este já puder ser reativado, isto é, se

running↑.primeiro↑.pai↑.ntasks = 0

Observe-se no algoritmo V.9 que a chamada a suspende realiza-se com o valor de running↑.primeiro igual a nil, o que,

segundo comentado na seção V.4, indica que a memória ocupada por esse processo deve ser liberada para aproveitamento futuro por outros processos.

V.6- SEMÁFOROS BINÁRIOS

Como foi dito na seção IV.4, foram adotados semáforos binários para sincronização dos processos concorrentes introduzidos no Sistema P. Um semáforo binário (declarado como semaphore em um programa concorrente) é representado no núcleo por uma variável do tipo

```
type semáforo = record
                  valor: boolean;
                  filasem: fila
                end;
```

onde fila é o tipo definido na seção V.2 para representar filas de processos. Observe-se que essa representação de um semáforo binário é mais elaborada do que a sugerida no item III.2.2, segundo a qual um semáforo pode ser representado por um flag apenas . A representação pelo tipo semáforo acima definido tem as vantagens de evitar esperas ocupadas possivelmente longas (pois nada se conhece sobre a região crítica que o semáforo encapsula) e de permitir a aplicação de políticas na liberação dos processos para o uso de algum recurso protegido pelo semáforo.

A rotina initsem para inicializar semáforos binários apresentada na seção IV.4 é implementada no núcleo através do procedimento do algoritmo V.10, para o qual o compilador pode gerar chamadas durante a compilação de programas concorrentes. Observe-se que a inicialização do semáforo inclui a inicialização da fila associada a ele, realizada pela rotina do núcleo initfila , que sobre uma fila f realiza

```
f.vazia := true;
```

Uma chamada à função lógica empty apresentada na seção

```
procedure initsem(var S: semáforo; v: boolean);
begin entraRC;
  with S do
    begin valor := v;
      initfila(filasem)
    end;
  deixaRC
end;
```

ALGORITMO V.10 - Inicialização de um Semáforo Binário

IV.4, quando aplicada a um semáforo binário, é traduzida pelo compilador em uma chamada à função do algoritmo V.11.

```
function emptysem(S: semáforo): boolean;
begin
  entraRC;
  emptysem := S.filasem.vazia;
  deixaRC
end;
```

ALGORITMO V.11 - Teste de uma Fila de Semáforo

As operações P e V realizáveis sobre semáforos, quando aplicadas aos semáforos binários adotados no Sistema P, são implementadas segundo os procedimentos dos algoritmos V.12 e V.13, respectivamente. Conforme foi dito na seção IV.4, a operação P pode ser chamada com um ou dois parâmetros e o segundo refere-se a uma prioridade a ser utilizada na ordem relativa de bloqueio dos processos na fila do semáforo. Quando o programador a utiliza sem especificar essa prioridade, o compilador gera um número negativo como segundo parâmetro, indicando que deve ser usada a prioridade de escalonamento para execução (schedprior). Observe-se que se o semáforo possui valor false, então o processo que chamou a operação P (running↑.primeiro↑)

```
procedure P(var S: semáforo; p: integer);
begin entraRC;
  with S do
    if not valor then
      if p < 0 then suspende(filasem, running↑.primeiro↑.schedprior)
      else suspende(filasem, p)
    else
      begin valor := false;
        deixaRC
      end
end;
```

ALGORITMO V.12 - Operação P sobre Semáforos Binários

deve bloquear-se, sendo então escolhido um novo processo para execução. A operação V, no caso de haver processos bloqueados na fila do semáforo, ativa o primeiro processo dessa fila. O escalonamento desse processo para execução dependerá da situação relativa de sua prioridade para execução nas próximas interrupções de "time-slice" dos diversos processadores.

```
procedure V(var S: semáforo);
begin entraRC;
  with S.filasem do
    if not vazia then
      transf(filasem, ready, primeiro↑.schedprior)
    else
      S.valor := true;
  deixaRC
end;
```

ALGORITMO V.13 - Operação V sobre Semáforos Binários

Com relação às operações sobre semáforos binários apresentadas nesta seção, deve ser observado que todas constituem re

giões críticas sobre o semáforo particular manipulado em cada chamada. As operações P e V, em particular, são também regiões críticas sobre as filas ready e running↑. Conforme foi dito na seção V.3, essas regiões críticas devem ser executadas com a exclusão garantida pelas rotinas entraRC e deixaRC (algoritmo V.1).

V.7- FILAS DE EVENTOS

Esta seção descreve a implementação das filas de eventos adotadas no Sistema P para tratamento de condições dentro de regiões críticas (seção IV.4). Uma fila de eventos (declarada com o tipo queue em um programa concorrente) é representada por uma estrutura do tipo fila definido na seção V.2. Ao ser iniciada a execução de um programa concorrente, todas as variáveis do tipo queue declaradas no programa são inicializadas através de chamadas geradas automaticamente pelo compilador à rotina initfila mencionada na seção V.6. Como foi dito nessa seção, a aplicação de initfila a uma fila f realiza

```
f.vazia := true;
```

Foi dito na seção IV.4 que um programa concorrente pode chamar a função empty para verificar se uma fila de eventos está ou não vazia. O compilador traduz uma dessas chamadas em uma chamada à função do algoritmo V.14.

```
function emptyfila(f: fila): boolean;
begin
  emptyfila := f.vazia
end;
```

ALGORITMO V.14 - Teste de uma Fila de Eventos

A implementação das operações wait e signal introduzidas no Sistema P para manipulação de suas filas de eventos é re

alizada respectivamente de acordo com os algoritmos V.15 e V.16. Como no caso da rotina P sobre semáforos binários (seção V.6) , se a rotina wait for utilizada sem especificação de prioridade, o compilador gerará um valor negativo, indicando ao núcleo que a prioridade para execução (schedprior) deve ser empregada para posicionar o processo (running↑.primeiro↑) na fila de even - tos.

```
procedure wait(var f: fila; var S: semáforo; p: integer);
begin entraRC;
  with S, S.filasem do
    if vazia then valor := true
    else transf(filasem, ready, primeiro↑.schedprior);
  if p < 0 then suspende(f, running↑.primeiro↑.schedprior)
  else suspende(f, p)
end;
```

ALGORITMO V.15 - Operação wait sobre Filas de Eventos

```
procedure signal(var f: fila; var S: semáforo);
begin
  with f do
    if vazia then V(S)
    else
      begin entraRC;
        transf(f, ready, primeiro↑.schedprior);
        deixaRC
      end
end;
```

ALGORITMO V.16 - Operação signal sobre Filas de Eventos

Observe-se que a rotina wait, após reativar algum processo que possa eventualmente estar esperando para entrar na re

gião crítica, executa uma chamada a suspende, a fim de buscar um novo processo para execução. A rotina signal, no caso de haver processos bloqueados na fila de eventos f, ativa o primeiro desses processos, passando-o para a fila ready. Novamente o escalonamento desse processo para execução dependerá da situação relativa de sua prioridade para execução nas próximas interrupções de "time-slice" dos diversos processadores.

Uma observação interessante que deve ser feita com relação às operações de manipulação de filas de eventos é que elas não constituem regiões críticas sobre essas filas. Um processo que as execute possui exclusividade no acesso às filas , uma vez que a utilização de filas de eventos é restrita, por sua definição, à região crítica sobre o recurso ao qual se relacionam. A utilização dos procedimentos entraRC e deixaRC nos algoritmos V.15 e V.16 destina-se a proteger o semáforo e as filas ready e running↑.

V.8- OPERAÇÕES DE ENTRADA E SAÍDA

V.8.1- ORGANIZAÇÃO BÁSICA

Conforme foi dito no item II.4.7, as operações de entrada e saída no Sistema P podem ser realizadas diretamente para as unidades de e/s (através das rotinas standard unitread e unitwrite) ou através da utilização de arquivos. Nesse último caso, a chamada às rotinas básicas unitread e unitwrite é realizada após operações, transparentes ao usuário, de manipulação de diretórios, formatação e bufferização de dados realizadas por rotinas do primeiro segmento de PASCALSYSTEM (algoritmo II.3). A figura II.14 fornece uma boa idéia dessa estrutura das operações de e/s.

De acordo com o que foi dito na seção IV.1, é desejável que a extensão realizada sobre o Sistema P para processamento concorrente permita que possam continuar a ser executados programas seqüenciais, até mesmo os que não podem ser recompilados. Dessa forma, a estrutura de operações de entrada e

saída ilustrada na figura II.14 deve continuar a ser válida para o processamento de programas seqüenciais. A figura V.4 apresenta a estrutura adotada para as operações de entrada e saída

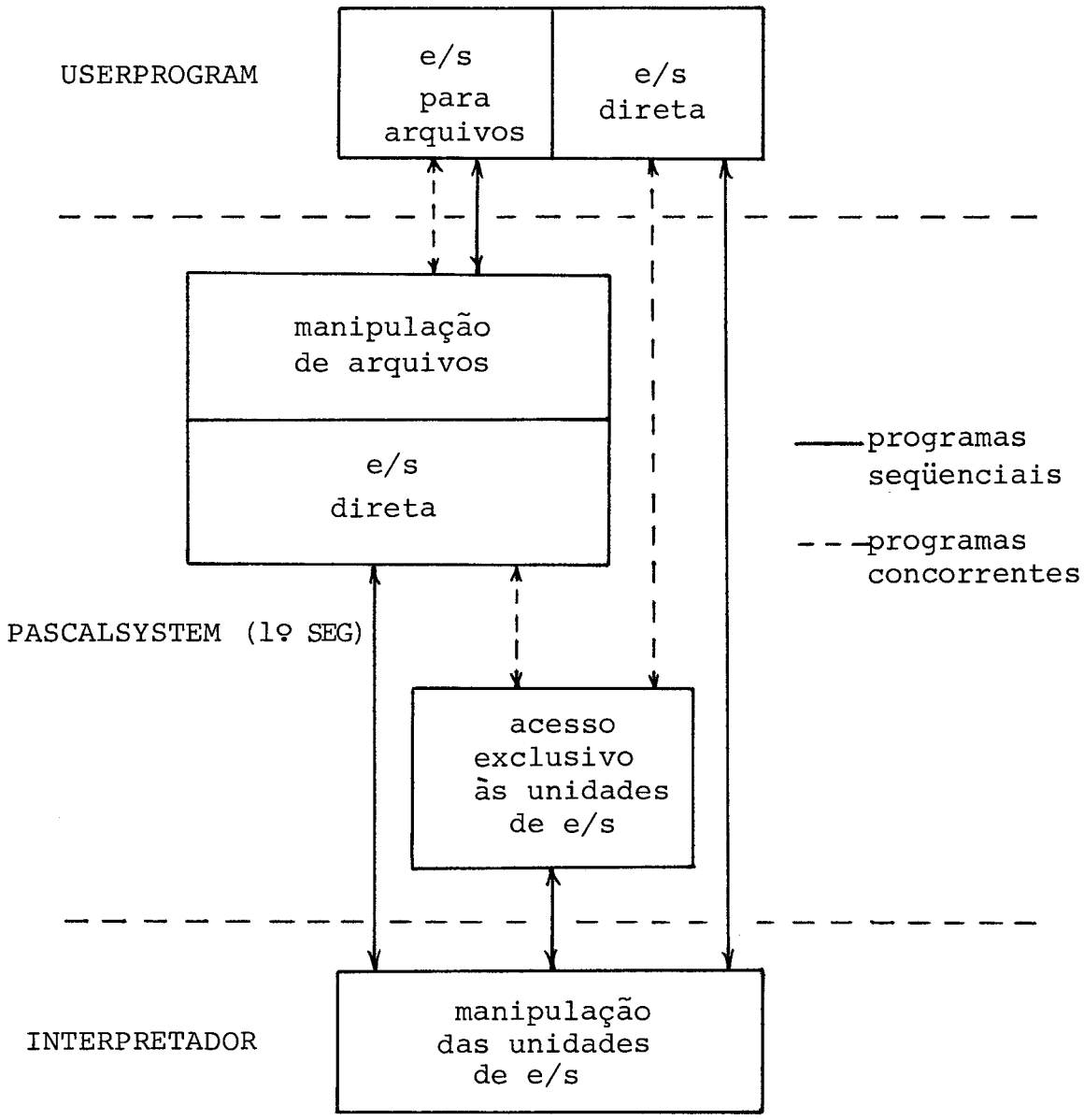


FIGURA V.4 - Hierarquia das Operações de E/S para Processamento Concorrente

no Sistema P, na extensão aqui proposta para processamento con-

corrente. A principal modificação em relação à estrutura da figura II.14 é quanto à inclusão de um módulo destinado a prover acesso exclusivo às unidades de e/s. Durante a compilação de programas concorrentes, chamadas às operações básicas unitread, unitwrite, unitwait e unitbusy (item II.3.5) são substituídas por chamadas a rotinas do núcleo que, após cuidarem do problema da exclusão mútua, realizam as chamadas às operações básicas originais. Essas rotinas do núcleo encontram-se descritas no item V.8.2. Com relação às operações de e/s para arquivos, as chamadas geradas pelo compilador são as mesmas no caso de programas seqüenciais ou concorrentes. Durante a execução, porém, pode ser reconhecida a existência de concorrência através das variáveis concexec e raizexec apresentadas na seção V.4. No caso da execução de um programa concorrente, o núcleo cuida para que os diretórios das unidades blocadas de e/s sejam manipulados com exclusão mútua e para que sejam chamadas as rotinas básicas de e/s apropriadas (para garantir acesso exclusivo às unidades), como tratado no item V.8.3.

Um problema comum às operações de e/s diretas e para arquivos é o tratamento de erros de e/s. Conforme descrito nos itens II.3.5 e II.4.7, a rotina standard ioresult pode ser utilizada para verificar a ocorrência de erros de e/s, o que é realizado através do campo iorslt da SYSCOM (item II.3.2). No caso de programas concorrentes esse campo não mais pode ser usado para armazenar o código de erros de e/s, uma vez que não há como saber a que operação ele se refere. Para o Sistema P adotou-se a solução de associar um campo similar a esse, chamado também iorslt, a cada nodo representativo de um processo. Assim, se p tem o tipo nodoptr descrito na seção V.2, então p↑.iorslt possui o código relativo à última operação de e/s realizada pelo processo representado por p↑. Sempre que ocorrer algum erro de e/s, durante a execução de um programa seqüencial ou concorrente, será executada a rotina setioresult do algoritmo V.17, que colocará o código do erro no campo adequado à situação. Durante a compilação de programas concorrentes o compilador traduz chamadas à rotina ioresult em chamadas à rotina cioresult do núcleo, que opera segundo o algoritmo V.18. O mesmo problema existe quan

```
procedure setioresult(resultado: integer);
begin
  if concexec then
    running↑.primeiro↑.iorslt := resultado
  else SYSCOM↑.iorslt := resultado
end;
```

ALGORITMO V.17 - Operação para Registrar Erros de E/S

```
function cioresult: integer;
begin
  cioresult := running↑.primeiro↑.iorslt
end;
```

ALGORITMO V.18 - Operação para Verificar a Ocorrência de Erros de E/S

to ao teste automático de erros de e/s mencionado no item II.4.5. No caso de programas concorrentes esses testes são realizados pela rotina ciocheck do algoritmo V.19.

```
procedure ciocheck;
begin
  if cioresult <> código de operação sem erro then
     informa  o erro  ocorrido ao usuário
end;
```

ALGORITMO V.19 - Teste Automático de Ocorrência de Erros de E/S

Nos dois próximos itens (V.8.2 e V.8.3) será utilizada a variável unitable definida por

```
var unitable: array[1..maxunit] of
                record ... end;
```

onde maxunit é o número de unidades de e/s do Sistema. Cada posição de unitable contém campos destinados a armazenar informações sobre a unidade de e/s correspondente. Esses campos serão apresentados ao longo dos itens V.8.2 e V.8.3 ao se fazerem necessários.

V.8.2- MANIPULAÇÃO DAS UNIDADES DE E/S

Conforme foi dito no item V.8.1, as operações de manipulação das unidades de e/s, no caso de programas concorrentes, devem ser realizadas com exclusão mútua entre os processos. Assim, todas as chamadas às operações de e/s direta geradas para um programa concorrente são interceptadas pelo núcleo, que então cuida para que sejam realizadas disciplinadamente. Para tanto, cada posição da unitable apresentada no item V.8.1 possui um semáforo binário, chamado sdev. No caso de unidades não-bloca - das de e/s o semáforo sdev é utilizado para prover exclusão mútua no acesso a essas unidades. Para unidades blocadas, sdev provê exclusão mútua no acesso a um monitor dessas unidades. A distinção entre as unidades blocadas e não-blocadas é realizada através do campo uisblkd da unitable. Todas as chamadas às rotinas unitread e unitwrite por programas concorrentes, sejam realizadas de forma direta ou não, são desviadas para a rotina cunitio do algoritmo V.20. Os parâmetros que ela recebe são todos os que receberiam as rotinas básicas, além de doread, que especifica se deve ser realizada uma operação de leitura (doread = true) ou escrita. Seus significados são:

- lunit: número da unidade de e/s;
- data: endereço de memória onde se encontram os dados a serem lidos ou escritos;
- length: número de bytes a serem transferidos;
- initblock: bloco inicial, no caso de unidades blocadas de e/s;
- async: se for igual a true, indica que a operação deve realizar-se assincronamente; se igual a false, sincronamente.

```
procedure cunitio(...; async, doread: boolean);
begin
  if raizexec then
    if doread then unitread(..., async) else unitwrite(..., async)
  else
    with unitable[lunit] do
      begin
        if uisblkd then getdisk(lunit, ...)
        else P(sdev, running↑.primeiro↑.schedprior);
        if async then
          if doread then unitread(..., true)
          else unitwrite(..., true)
        else
          if "pequeno" then
            begin disable;
              if doread then unitread(..., false)
              else unitwrite(..., false);
              if uisblkd then reldisk(lunit)
              else V(sdev)
            end
          else
            begin disable;
              if doread then unitread(..., true)
              else unitwrite(..., true);
              with running↑.primeiro↑ do
                schedprior := schedprior - 101;
              P(waitio, 50);
              if uisblkd then reldisk(lunit)
              else V(sdev);
              setpriority(running↑.primeiro↑.schedprior + 101)
            end
      end
end;
```

ALGORITMO V.20 - Um Monitor para as Unidades de E/S

Observe-se inicialmente que se a chamada a cunitio pro

vêm da raiz da árvore de processos (raizexec = true), então não há necessidade de sincronização, podendo as rotinas básicas ser chamadas imediatamente. Caso contrário, a sincronização é necessária e depois de obtida exclusão mútua surgem duas possibilidades. Se o pedido foi de uma e/s assíncrona, ela é imediatamente iniciada e o controle volta ao processo que realizou a chamada, sem liberar a unidade de e/s correspondente. Se o pedido foi de uma e/s síncrona, então mais duas possibilidades surgem. Se a e/s síncrona solicitada é rápida, de tal modo que não vale a pena escalonar outro processo para execução enquanto ela se realiza, então ela é realizada imediatamente de forma síncrona, com interrupções inibidas para garantir a liberação do recurso assim que se complete. Em caso contrário, apesar de ter sido feito um pedido de e/s síncrona, ela é realizada assincronamente, a fim de que o núcleo continue a ser executado e possa escalonar outro processo para execução enquanto ela se completa. O processo que solicitou a operação de e/s é bloqueado na fila do semáforo waitio da unitable de acordo com a prioridade "default" do Sistema (essa prioridade poderia ser qualquer, uma vez que essa fila tem no máximo um único processo). Observem-se dois fatos importantes:

(i) a operação de e/s é iniciada com interrupções inibidas, a fim de garantir o imediato encadeamento do processo na fila do semáforo waitio;

(ii) antes de ser bloqueado, o processo que solicitou a operação de e/s tem sua prioridade para execução transformada em um número negativo, o que o tornará mais prioritário que qualquer outro processo quando a operação se completar. O objetivo é garantir a liberação imediata do recurso, após o que a prioridade original para execução é restabelecida.

Como pode ser observado no algoritmo V.20, a região crítica RC sobre uma unidade não-blocada de e/s é garantida apenas pelo uso do semáforo sdev:

```
with unitable[lunit] do
  begin
    P(sdev);
    RC;
    V(sdev)
  end;
```

Para uma unidade blocada de e/s, porém, a exclusão mútua pe garantida através das operações getdisk e reldisk, segundo:

```
with unitable[lunit] do
  begin
    getdisk(lunit,...);
    RC;
    reldisk(lunit)
  end;
```

Essas operações são específicas para o caso mais comum em que as unidades blocadas de e/s são mapeadas em discos e têm o objetivo de ordenar os pedidos de e/s de modo a minimizar os efeitos de atraso causados por movimentos radiais excessivos do braço de leitura/escrita do disco. Esse escalonamento para uso das unidades blocadas de e/s utiliza três constantes: fblksize (tamanho de um bloco em bytes; item II.4.6), blockspertrack (número de blocos em uma trilha do disco) e maxtrack (número máximo de uma trilha no disco, a partir de 0). Observe-se que essas duas últimas constantes podem ser diferentes para cada unidade, dependendo do disco físico utilizado; nesse caso, podem ser campos de unitable. Os outros parâmetros que a rotina getdisk deve receber são os números da trilha inicial e da trilha final ocupadas pelos blocos da e/s solicitada. Supondo que o disco está organizado de modo que uma numeração crescente de blocos corresponde a uma numeração não-decrescente de trilhas, então os números das trilhas inicial e final são respectivamente

```
initblock div blockspertrack
```

e

(initblock + length div fblksize) div blockspertrack

Os algoritmos de getdisk e reldisk são similares aos apresentados em HOLT et al[27] e podem ser vistos no algoritmo V. 21. Essas operações utilizam outros campos de unitable:

- inuse, do tipo boolean, que indica se a unidade está sendo usada;
- direction, do tipo (up, down), que indica a direção corrente de deslocamento do braço do disco;
- upsweep e downsweep, filas de eventos, que servem para que processos possam esperar pela próxima varredura do braço do disco nas direções up e down, respectivamente;
- curtrack, do tipo integer, que é o número da trilha que está sendo correntemente utilizada em uma operação de e/s.

O princípio de funcionamento das operações getdisk e reldisk é que o próximo pedido de e/s a ser atendido é aquele cuja trilha inicial está mais próxima da trilha final do pedido que está sendo atendido. Para evitar o adiamento indefinido do atendimento a um determinado pedido,essa política é restrita à direção corrente de deslocamento do braço. Se essa direção é up, por exemplo, então são esgotados todos os pedidos de e/s nessa direção, na ordem ótima, antes que a direção seja invertida. Observe-se que essa ordem ótima pode não ser obtida quando o pedido de e/s inclui mais de uma trilha (lasttrack > track); o resultado obtido, porém, pode ser uma boa aproximação se o número de trilhas é pequeno.

Quando no algoritmo V.20 é ativada uma operação de e/s assíncrona, o processo que a solicitou continua sua execução e a unidade envolvida não é liberada. Se o programador solicitou uma operação de e/s assíncrona é porque pretendia realizar outras tarefas enquanto ela era realizada, verificando mais tarde se foi completada por meio das operações unitbusy e unitwait. Essas operações, no caso de programas concorrentes, são responsáveis pela liberação do recurso. O compilador, ao encontrar re

```
procedure getdisk(lunit: integer;
                  track, lasttrack: integer);
begin
  with unitable[lunit] do
    begin P(sdev, running↑.primeiro↑.schedprior);
      if inuse then
        if (curtrack < track) or
           ((curtrack = track) and (direction = down)) then
          wait(upsweep, sdev, track)
        else
          wait(downsweep, sdev, maxtrack - track);
      inuse := true;
      curtrack := lasttrack;
      V(sdev)
    end
end;
procedure reldisk(lunit: integer);
begin
  with unitable[lunit] do
    begin P(sdev, running↑.primeiro↑.schedprior);
      inuse := false;
      if direction = up then
        if upsweep.vazia then
          begin direction := down;
            signal(downsweep, sdev)
          end
        else signal(upsweep, sdev)
      else
        if downsweep.vazia then
          begin direction := up;
            signal(upsweep, sdev)
          end
        else signal(downsweep, sdev)
    end
end;
```

ALGORITMO V.21 - Otimização dos Movimentos dos Braços dos Discos

ferências a elas, gera chamadas às rotinas cunitbusy e cunitwait do núcleo (algoritmo V.22). Observe-se que assim a liberação das

```
function cunitbusy(lunit: integer): boolean;
var busy: boolean;
begin
  if raizexec then cunitbusy := unitbusy(lunit)
  else
    begin disable;
      busy := unitbusy(lunit);
      cunitbusy := busy;
      if not busy then
        with unitable[lunit] do
          if uisblkd then reldisk(lunit)
          else V(sdev)
      else enable
    end
end;
procedure cunitwait(lunit: integer);
begin
  if raizexec then unitwait(lunit)
  else
    repeat (* espera *) until not cunitbusy(lunit)
end;
```

ALGORITMO V.22 - Operações para Verificar o Término de Operações Assíncronas de E/S

unidades de e/s passa a ser responsabilidade do programador, no caso de operações assíncronas. A função cunitbusy tem parte de seu código executada com interrupções inibidas para garantir a imediata liberação da unidade de e/s.

Quando a rotina cunitio é chamada para executar uma operação síncrona de e/s e verifica que essa operação vai consumir um tempo razoavelmente grande, a chamada à rotina standard

correspondente é realizada de maneira assíncrona e o processo é bloqueado na fila do semáforo waitio associado à unidade de e/s. A reativação desse processo é feita pela rotina iointerrupt do algoritmo V.23 que é ativada sempre que termina uma operação assíncrona de e/s. Observe-se que a reativação do processo bloqueado na fila do semáforo waitio pode ser vista como uma sinalização originada do processo que foi interrompido pelo término da operação de e/s.

```
procedure iointerrupt(lunit: integer);
begin
  with unitable[lunit] do
    if not waitio.filasem.vazia then V(waitio)
end;
```

ALGORITMO V.23 - Tratamento de Interrupções por Final de Operações de E/S

V.8.3- ENTRADA E SAÍDA PARA ARQUIVOS

O principal problema de concorrência existente nas operações de e/s para arquivos é o da manipulação de diretórios. Como foi dito no item II.4.6, o diretório de uma unidade blocada de e/s é copiado para uma área alocada dinamicamente no heap sempre que necessário, sendo daí removido ao ocorrer a alocação dinâmica de qualquer outro dado. O diretório assim carregado é endereçado por um apontador comum a todas as unidades blocadas de e/s, gdirp. Na extensão para processamento concorrente esse apontador único foi substituído por vários apontadores, um para cada unidade blocada, alocados como campos de unitable. Para uma unidade blocada de número lunit esse apontador é unitable[lunit].dirp. Existe dessa forma a possibilidade de manipular unidades blocadas diferentes ao mesmo tempo.

Uma operação de e/s para arquivo que consulte o diretó

rio ou realize alterações nele constitui uma região crítica sobre esse diretório e deve ser executada com exclusão mútua com relação aos demais processos que possam competir pelo uso do mesmo diretório. Para tanto, cada posição de unitable correspondente a uma unidade blocada possui um semáforo sdir. A região crítica sobre um determinado diretório é então precedida pela operação getdir e sucedida pela operação reldir (algoritmo V.24) .

```
procedure getdir(lunit: integer);
begin
  with unitable[lunit] do
    begin
      if concexec then
        if not raizexec then
          P(sdir, running↑.primeiro↑.schedprior);
      new(dirp)
    end
end;
procedure reldir(lunit: integer);
begin
  with unitable[lunit] do
    begin
      dispose(dirp);
      if concexec then
        if not raizexec then
          V(sdir)
    end
end;
```

ALGORITMO V.24 - Operações para Entrada e Saída de Regiões Críticas sobre Diretórios

Entre essas duas operações são realizados todos os trabalhos necessários de leitura, pesquisa, atualização e reescrita do diretório. Observe-se que getdir e reldir são chamadas mesmo que não exista concorrência. Nesse caso efetuam apenas a alocação e re-

moção dinâmicas do espaço para o diretório. Um outro problema de concorrência que existe nas operações de e/s para arquivos é o das chamadas finais às rotinas básicas de e/s. A solução desse problema utiliza as operações apresentadas nos itens V.8.2 para e/s direta concorrente. Dessa forma, tem-se para as operações finais de acesso às unidades de e/s:

```
if concexec then
  cunitio(..., true)
else unitread(...)
```

ou

```
if concexec then
  cunitio(..., false)
else unitwrite(...)
```

CAPÍTULO VI

COMENTÁRIOS FINAIS

Este capítulo destina-se a algumas observações gerais com relação ao material apresentado nos capítulos IV e V, especificamente sobre as extensões realizadas sobre o Pascal UCSD para programação concorrente e o projeto de seu núcleo.

Conforme pode ser visto no capítulo IV, a extensão do Pascal UCSD para programação concorrente baseou-se na incorporação de ferramentas de um nível de abstração relativamente baixo, de modo a poder permitir uma maior flexibilidade em termos de políticas de escalonamento e alguns problemas específicos de sincronismo. Em contraste com conceitos menos flexíveis, como o de monitor apresentado no item III.2.5, a utilização dos semáforos binários e filas de eventos adotados para estender o Pascal UCSD proporciona pouca segurança, no sentido em que impossibilita a realização de testes em tempo de compilação que previnam, por exemplo, referências a recursos compartilhados fora das regiões críticas apropriadas. Por outro lado, sua utilização para resolver outros problemas de sincronização é relativamente simples, como conseqüência de sua maior flexibilidade.

O ganho em flexibilidade (e conseqüente perda em segurança) advindo do uso da extensão do Pascal UCSD para programação concorrente exige que o programador tenha que se preocupar com alguns problemas importantes que aparecem em programação concorrente. Um desses problemas é a prevenção de situações de bloqueio perpétuo ("deadlock"); conforme pode ser visto em COFFMAN et al [5], essas situações configuram-se quando, por exemplo, um processo A bloqueia-se à espera que uma condição C1 seja cumprida por um outro processo B que, para tanto, precisa do cumprimento de uma condição C2 pelo processo A. Em problemas de alocação de recursos, situações de bloqueio perpétuo podem ocorrer quando, por exemplo, dois processos A e B competem pelos re

cursos R1 e R2 e tentam alocá-los em ordens diferentes. Se A tenta alocar os recursos na ordem R1-R2 e B na ordem R2-R1, um problema de bloqueio perpétuo ocorre se a alocação de R2 por B interpõe-se à alocação de R1 e de R2 por A, por exemplo. Observe-se que esses casos de regiões críticas aninhadas sobre recursos diferentes ocorrem durante a execução do próprio núcleo proposto no capítulo V: para realizar certas operações de e/s para arquivos, o processo que deseja fazê-lo precisa obter acesso exclusivo ao diretório da unidade blocada de e/s correspondente e depois então obter acesso à própria unidade de e/s, para ler e/ou escrever o diretório. Nesses casos, porém, pode-se garantir que não ocorrem situações de bloqueio perpétuo, uma vez que o aninhamento dessas regiões críticas é realizado pelo núcleo sempre na mesma ordem. Obrigar os processos concorrentes a utilizarem o aninhamento de regiões críticas através da alocação hierárquica dos recursos envolvidos, isto é, tentando alocá-los sempre na mesma ordem, é uma disciplina que o programador deve adotar, com vistas a prevenir a ocorrência de situações de bloqueio perpétuo. Ainda com relação ao aninhamento de regiões críticas, um cuidado especial que o programador deve ter é o de garantir que não sejam feitas tentativas recursivas de alocação do mesmo recurso. Por exemplo, se o processo A aloca o recurso R e passa a utilizá-lo através da região crítica RC, então o processamento de RC não pode de modo algum conduzir a uma nova tentativa de alocação de R, o que claramente provocaria uma situação de "deadlock".

Um outro problema que passa a ser responsabilidade do programador como conseqüência da relativa flexibilidade das características de programação concorrente do Pascal UCSD estendido é o do escalonamento dos processos concorrentes para execução, para saída de filas de semáforos e de filas de eventos. Conforme foi dito na seção IV.4, um modo "default" de programação é ignorar as prioridades relativas dos diversos processos e deixar que o núcleo se encarregue do escalonamento desses processos nas diversas filas. Nesse caso, todas as filas do Sistema são processadas segundo a política FCFS, que estabelece que a saída dos processos de uma determinada fila em que se encontram ocor-

re na mesma ordem em que eles entraram na fila. Dessa forma, desde que não ocorram situações de bloqueio perpétuo ou de "loops" infinitos na execução de um determinado processo, pode-se dizer que a saída de um processo de uma determinada fila não é adiada indefinidamente. Assim é que, por exemplo, todos os processos prontos para execução à espera de um processador disponível têm em algum momento suas chances de serem executados (nesse caso o efeito de eventuais "loops" pode ser diminuído pela aplicação de interrupções periódicas de "time-slice"). Da mesma forma, processos bloqueados em filas de semáforos ou filas de eventos têm sua saída dessas filas em um tempo finito assegurada. Todavia, se o programador opta por utilizar a especificação explícita de prioridades no processamento das filas do Sistema, através da operação setpriority ou de ordens relativas de bloqueio nas filas de semáforos e de eventos, então o problema de não adiar indefinidamente o atendimento a um determinado processo passa também a ser responsabilidade do programador. Este deve observar que a possibilidade de explicitar prioridades relativas para o escalonamento dos processos concorrentes foi incorporada ao Pascal UCSD em sua extensão com a finalidade de ampliar a classe de programas a que ele seria aplicável. Especialmete no caso de programas concorrentes para processamento em tempo real, sabe-se que é essencial poder realizar a distinção entre processos através de diferentes prioridades para execução ou para utilização de recursos compartilhados. Mais uma vez observa-se que a maior flexibilidade inerente à programação em Pascal UCSD estendido implica também em que o programador deva preocupar-se com problemas que de outra forma seriam resolvidos pelo compilador ou pelo núcleo.

Também no núcleo existem alguns problemas de adequação à situação particular a que se pretende aplicá-lo. O núcleo sugerido no capítulo V tem a característica importante de utilizar interrupções de "time-slice" para realizar o escalonamento para execução por prioridades proposto na seção IV.4. Porém, como foi observado na seção V.4, pode haver aplicações que possuam situações prementes de escalonamento para as quais seja inadmissível ter que esperar um período de "time-slice" para serem realiza -

das. Como diminuir esse período poderia causar uma degradação considerável no desempenho do Sistema, a solução estaria em interromper todos os processadores quando da ativação de algum processo. Os processadores assim interrompidos poderiam verificar a situação relativa dos processos que estivessem executando em relação aos processos esperando por processadores disponíveis e passar a executá-los, se fosse o caso. Observe-se que o efeito global dessas interrupções poderia ser o de trocar processos entre processadores simplesmente, o que poderia causar perdas relevantes de eficiência, dependendo da arquitetura particular utilizada. Uma solução um pouco melhor poderia basear-se na existência de uma fila global em que estivessem encadeados todos os processos em execução nos diversos processadores, permitindo a cada processador saber o número de processos menos prioritários que o seu que estivessem em execução. Comparando esse número ao de processos mais prioritários esperando para serem executados, chegar-se-ia a uma conclusão sobre suspender ou não cada processo quando da interrupção. Em um esquema ainda um pouco mais rápido, o próprio processo que efetuou a ativação de novos processos poderia formar uma lista identificando os que precisariam ser suspensos, interrompendo todos os processadores para que consultassem essa lista.

O núcleo também poderia ser alterado no sentido de evitar as políticas estabelecidas pelo algoritmo V.21 para otimizar os movimentos dos braços dos discos. Essas seriam deixadas para o nível superior de um processo gerente,dando ao Sistema mais flexibilidade, porém obrigando o programa concorrente a conhecer detalhes do mapeamento das unidades blocadas de e/s.

Observe-se finalmente que a aplicabilidade da extensão proposta neste trabalho para o Pascal UCSD só pode ser avaliada em função dos resultados obtidos com algumas implementações experimentais. Como foi observado ao longo do texto, fatores como a arquitetura adotada, o sistema concorrente que se pretende programar e a implementação do núcleo, entre outros, são condicionantes da maior ou menor aplicabilidade da programação concorrente em Pascal UCSD estendido.

APÊNDICE A

PROCEDIMENTOS INTRÍNSECOS DO PASCAL UCSD

Este apêndice fornece uma descrição breve dos procedimentos intrínsecos da linguagem Pascal UCSD que não fazem parte da definição original da linguagem Pascal (JENSEN & WIRTH[28]) ou que não possuem, na definição original, a mesma função (como RESET e REWRITE). Uma descrição detalhada de cada um desses procedimentos pode ser encontrada em SHILLINGTON & ACKLAND[36].

BLOCKREAD — Function para ler um número variável de blocos de um arquivo "sem tipo" (ver "untyped files" na referência recomendada acima).

BLOCKWRITE — Function para escrever um número variável de blocos em um arquivo "sem tipo" (ver "untyped files" na referência recomendada acima).

CLOSE — Procedure para fechar arquivos.

CONCAT — Intrínseco usado para concatenar variáveis do tipo STRING.

DELETE — Intrínseco usado para remover caracteres de variáveis do tipo STRING.

DRAWLINE — Intrínseco gráfico.

DRAWBLOCK — Intrínseco gráfico.

EXIT — Intrínseco usado para sair de uma rotina em qualquer ponto.

GOTOXY — Procedure usada para endereçar o cursor de um terminal de vídeo, colocando-o na posição da tela que corresponde a uma determinada linha e coluna.

FILLCHAR — Procedure rápida para inicializar variáveis do tipo packed array of char.

HALT — Interrompe a execução de um programa do usuário, podendo resultar numa chamada ao "debugger" interativo.

IDSEARCH — Rotina usada pelo compilador Pascal e pelo montador para PDP-11.

INSERT — Intrínseco usado para inserir caracteres em variáveis do tipo STRING.

IORESULT — Function que retorna o resultado da última operação de e/s realizada.

LENGTH — Intrínseco que retorna o comprimento dinâmico de uma variável do tipo STRING.

MARK — Usada para marcar o topo do heap em alocação dinâmica de memória.

MOVELEFT — Intrínseco de baixo nível para mover quantidades grandes de bytes.

MOVERIGHT — Intrínseco de baixo nível para mover quantidades grandes de bytes.

REWRITE — Procedure para abrir um novo arquivo.

RESET — Procedure para abrir um arquivo existente.

POS — Intrínseco que retorna a posição de um determi

nado padrão numa variável do tipo STRING.

PWROFTEN — Function que retorna como um resultado do tipo real o número 10 elevado à potência do parâmetro do tipo integer fornecido.

RELEASE — Intrínseco usado para liberar espaços de memória ocupados por variáveis alocadas dinamicamente no heap.

SEEK — Usada para realizar acesso aleatório a registros de um arquivo.

SIZEOF — Function que retorna o número de bytes alocados para uma variável.

STR — Procedure para converter inteiros longos em variáveis do tipo STRING.

TIME — Function que retorna o tempo decorrido desde o último "bootstrap" do Sistema (retorna 0 se o microcomputador não possui um relógio de tempo real).

TREESEARCH — Rotina usada exclusivamente pelo compilador Pascal.

UNITBUSY — Intrínseco de baixo nível para determinar o estado de uma unidade de e/s.

UNITCLEAR — Intrínseco de baixo nível para cancelar operações de e/s.

UNITREAD — Intrínseco de baixo nível para ler de uma unidade de e/s.

UNITWAIT — Intrínseco de baixo nível para esperar o término de uma operação de e/s.

UNITWRITE Intrínseco de baixo nível para escrever em uma unidade de e/s.

APÊNDICE B

PRIMITIVAS DE SINCRONISMO : EXEMPLOS

B.1- O PROBLEMA DO BUFFER CIRCULAR

Este apêndice dedica-se a exemplificar os métodos de sincronismo entre processos concorrentes descritos na seção III.2. Todos os exemplos apresentados destinam-se a solucionar o problema de processos produtores e consumidores de itens de um buffer circular. Esse buffer (B) pode ser visto na figura B.1, onde também são mostrados dois apontadores: p para indicar o próximo espaço vazio a ser utilizado por um produtor, e c para indicar o próximo item a ser consumido. Os valores de p e c variam de 0 a maxbuf - 1, sendo incrementados através de

$$(p + 1) \text{ mod } maxbuf$$

e

$$(c + 1) \text{ mod } maxbuf,$$

respectivamente. Em algumas das soluções apresentadas está sendo suposto que B se compõe por elementos do tipo abstrato item.

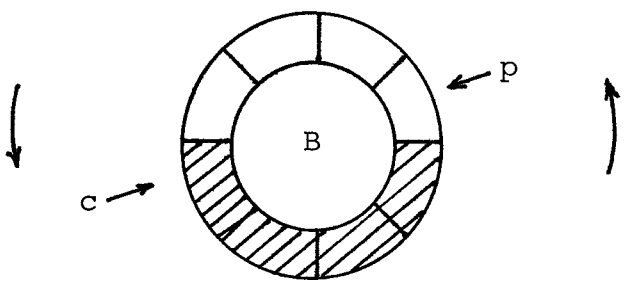


FIGURA B.1 - Um Buffer Circular

O problema consiste em garantir exclusão mútua entre um número qualquer de produtores e consumidores no acesso ao buffer.

B.2- SOLUÇÃO COM SEMÁFOROS

A solução do problema da seção B.1 pelo uso de semáforos (item III.2.2) utiliza um semáforo binário para exclusão mútua (mutex, inicializado com 1) e dois semáforos gerais: um para contar as posições cheias (posições-cheias, inicializado com 0) e um para contar as posições vazias (posições-vazias, inicializado com maxbuf).

Cada processo produtor deve executar os comandos do algoritmo B.1.

```
P(posições-vazias);
P(mutex);
deposita item em B[p];
p := (p + 1) mod maxbuf;
V(mutex);
V(posições-cheias);
```

ALGORITMO B.1 - Utilização de um Buffer Circular através de Semáforos

Analogamente, um processo consumidor deve executar o algoritmo B.2.

```
P(posições-cheias);
P(mutex);
retira item de B[c];
c := (c + 1) mod maxbuf;
V(mutex);
V(posições-vazias);
```

ALGORITMO B.2 - Utilização de um Buffer Circular através de Semáforos

Observe-se que a ordem em que as operações P são executadas é fundamental, a fim de evitar problemas de bloqueio perpétuo ("deadlock").

B.3- SOLUÇÃO COM REGIÕES CRÍTICAS

Para resolver o problema proposto na seção B.1 através do uso de regiões críticas simples (item III.2.2), é necessário utilizar também dois semáforos binários, csem e psem, inicializados com 0, que possam ser utilizados para bloquear respectivamente processos consumidores e produtores dentro de suas regiões críticas. A fim de que possam ser utilizados em conjunto com regiões críticas simples esses semáforos deverão ser manipulados pelas operações

P'(S) : P(S); libera a região crítica;

V'(S) : if a fila de S não está vazia then
begin V(S);
sai da região crítica, sem liberá-la devido à entrada do processo da fila de S
end;

em substituição às operações P(S) e V(S) definidas sobre um semáforo S. Também deve ser usado um contador (cont) de posições preenchidas, inicializado com 0.

Para um processo produtor, o algoritmo a ser seguido é o algoritmo B.3

```
with B do
  begin
    if cont = maxbuf then P'(psem);
    deposita item em B[p];
    p:= (p + 1) mod maxbuf;
    cont := cont + 1;
    V'(csem)
  end;
```

ALGORITMO B.3 - Utilização de um Buffer Circular através de Regiões Críticas Simples

Um processo consumidor deve executar o algoritmo B.4.

```
with B do
  begin
    if cont = 0 then P'(csem);
    retira item de B[c];
    c := (c + 1) mod maxbuf;
    cont := cont - 1;
    V'(psem)
  end;
```

ALGORITMO B.4 - Utilização de um Buffer Circular através de Regiões Críticas Simples

Essa solução segue a linha proposta em HANSEN[14].

B.4- SOLUÇÃO COM REGIÕES CRÍTICAS CONDICIONAIS

A solução do problema apresentado na seção B.1 através do uso de regiões críticas condicionais (item III.2.3) utiliza um contador (cont) de posições preenchidas inicializado com 0.

Um processo produtor deve executar a seqüência de comandos do algoritmo B.5.

```
with B
  when cont < maxbuf do
    begin
      deposita item em B[p];
      p := (p + 1) mod maxbuf;
      cont := cont + 1
    end;
```

ALGORITMO B.5 - Utilização de um Buffer Circular através de Regiões Críticas Condicionais

De maneira análoga, um processo consumidor deve executar o algoritmo B.6.

```
with B
  when cont > 0 do
    begin
      retira item de B[c];
      c := (c + 1) mod maxbuf;
      cont := cont - 1
    end;
```

ALGORITMO B.6 - Utilização de um Buffer Circular através de Regiões Críticas Condicionais

B.5- SOLUÇÃO COM FILAS DE EVENTOS

Para solucionar o problema da seção B.1 com o uso de filas de eventos (item III.2.3), são definidas duas dessas filas:

```
var algumvazio,
    algumcheio: event B;
```

e um contador (cont) de posições preenchidas inicializado com 0.

Cada produtor de itens do buffer deve executar o algoritmo B.7.

```
with B do
  begin
    if cont = maxbuf then
      await(algumvazio);
    deposita item em B[p];
    p := (p + 1) mod maxbuf;
    cont := cont + 1;
    cause(algumcheio)
  end;
```

ALGORITMO B.7 - Utilização de um Buffer Circular através de Filas de Eventos

Para um consumidor de itens do buffer, o procedimento é o do algoritmo B.8.

```
with B do
  begin
    if cont = 0 then
      await(algumcheio);
    retira item de B[c];
    c := (c + 1) mod maxbuf;
    cont := cont - 1;
    cause(algumvazio)
  end;
```

ALGORITMO B.8 - Utilização de um Buffer Circular através de Filas de Eventos

B.6- SOLUÇÃO COM TROCA DE MENSAGENS

Esta seção apresenta uma solução para o problema proposto na seção B.1 através do uso de troca de mensagens (item III.2.4). A solução apresentada a seguir utiliza um processo dedicado a gerenciar o buffer circular. A linguagem utilizada para especificar esse processo segue a linha do Pascal, apesar de ser propositalmente indeterminada. Está sendo suposto sobre a linguagem de especificação que existe um tipo implícito queue, destinado a definir filas internas nas quais mensagens que não podem ser atendidas são encadeadas. Sobre uma variável do tipo queue supõem-se definidas as operações

encadeia(fila, atributos da mensagem)

e

libera(fila, atributos da mensagem)

Se uma fila está vazia, então a função empty fornece o resultado true. Também é suposta a existência do tipo implícito

processid para identificação de processos. O algoritmo B.9 descreve o processo gerente.

```
process buffer;

var B: array[0..maxbuf-1] of item;
    p, c: (0..maxbuf - 1);
    cont: (0..maxbuf);
    esperacheio, esperavazio: queue;
    m: string;
    dummy: processid;
    it: item;

procedure coloca(x: item);
begin
  B[p] := x;
  p := (p + 1) mod maxbuf;
  cont := cont + 1
end;

procedure retira(var x: item);
begin
  x := B[c];
  c := (c + 1) mod maxbuf;
  cont := cont - 1
end;
```

(continua...)

```
begin p := 0; c := 0; cont := 0;
  loop
    wait(dummy, m, it);
    case m of
      'colocar':
        if cont = maxbuf then
          encadeia(esperavazio, dummy, m, it)
        else
          begin
            coloca(it);
            send(dummy, 'item recebido', it);
            if not empty(esperacheio) then
              begin
                libera(esperacheio, dummy, m, it);
                retira(it);
                send(dummy, 'item enviado', it)
              end
          end;
      'retirar':
        if cont = 0 then
          encadeia(esperacheio, dummy, m, it)
        else
          begin
            retira(it);
            send(dummy, 'item enviado', it);
            if not empty(esperavazio) then
              begin
                libera(esperavazio, dummy, m, it);
                coloca(it);
                send(dummy, 'item recebido', it)
              end
          end
    end
  end loop
end;
```

ALGORITMO B.9 - Gerenciamento de um Buffer Circular através de Trocas de Mensagens

Um processo que deseje utilizar o buffer envia ao processo buffer a mensagem correspondente ('colocar' para produtores e 'retirar' para consumidores) e aguarda a resposta.

B.7- SOLUÇÃO EM PASCAL CONCORRENTE

Esta seção ilustra o conceito de monitor (item III.2.5), como utilizado na linguagem Pascal Concorrente, na solução do problema da seção B.1.

O monitor para esse problema é definido como o tipo abstrato do algoritmo B.10.

```
type buffer = monitor;
     var B: array[0..maxbuf-1] of item;
         p, c: (0..maxbuf - 1);
         cont: (o..maxbuf);
         produtor, consumidor: queue;
     procedure entry produz(x: item);
     begin
       if cont = maxbuf then delay(produtor);
       B[p] := x;
       p := (p + 1) mod maxbuf;
       cont := cont + 1;
       continue(consumidor)
     end;
     procedure entry consome(var x: item);
     begin
       if cont = 0 then delay(consumidor);
       x := B[c];
       c := (c + 1) mod maxbuf;
       cont := cont - 1;
       continue(produtor)
     end;
     begin p := 0; c := 0; cont := 0 end;
```

ALGORITMO B.10 - Gerenciamento de um Buffer Circular em Pascal Concorrente

Ao ser criado um monitor desse tipo, por exemplo

```
var itembuf: buffer;
```

o comando inicial é executado. Um processo produtor tem acesso ao monitor através de

```
itembuf.produz(item)
```

e um processo consumidor através de

```
itembuf.consome(item)
```

B.8- SOLUÇÃO EM MODULA

A solução em Modula para o problema da seção B.1 ilustra o conceito de monitor (item III.2.5) existente nessa linguagem. Para o problema em questão, o monitor é o do algoritmo B.11.

```
interface module buffer;
  define produz, consome;
  var B: array[0..maxbuf-1] of item;
      p, c: (0..maxbuf - 1);
      cont: (0..maxbuf);
      algumcheio, algumvazio: signal;
  procedure produz(x: item);
  begin
    if cont = maxbuf then wait(algumvazio);
    B[p] := x;
    p := (p + 1) mod maxbuf;
    cont := cont + 1;
    send(algumcheio)
  end produz;
  procedure consome(var x: item);
  begin
    if cont = 0 then wait(algumcheio);
    x := B[c];
    c := (c + 1) mod maxbuf;
    cont := cont - 1;
    send(algumvazio)
  end consome;
begin p := 0; c := 0; cont := 0 end buffer;
```

ALGORITMO B.11 - Gerenciamento de um Buffer Circular em Modula

Ao ser criado esse módulo, o comando de inicialização é executado. Um processo produtor tem acesso ao buffer através de

produz(item)

e um processo consumidor através de

consome(item)

B.9- SOLUÇÃO COM CSP

Esta seção apresenta a solução do problema da seção B.1 através do uso dos CSP, introduzidos no item III.2.6. No algoritmo B.12 encontra-se codificado o processo gerente do recurso (buffer); a notação utilizada não corresponde, por questões de clareza, à originalmente proposta em HOARE[26]. Também a utilização de comandos de saída (send) em guardas não faz parte da proposta original, tendo sido adotada para simplificar a solução (segundo sugestão do próprio autor em HOARE[26]).

```
process buffer;
var B: array[0..maxbuf-1] of item;
    p, c: (0..maxbuf - 1);
    cont: (0..maxbuf);
begin
  p := 0; c := 0; cont := 0;
  do (cont < maxbuf;
      receive(produtor,B[p])) → p := (p + 1) mod maxbuf;
                                cont := cont + 1
  or (cont > 0;
      send(consumidor,B[c])) → c := (c + 1) mod maxbuf;
                               cont := cont - 1
  od
end;
```

ALGORITMO B.12 - Utilização de CSP para Gerenciar um Buffer Circular

Essa solução aplica-se ao caso em que há um único processo produtor (chamado produtor), que executa

send(buffer, item)

e um único processo consumidor (chamado consumidor), que executa

receive(consumidor, item)

B.10- SOLUÇÃO COM DP

O problema proposto na seção B.1 pode ser solucionado pelo emprego de DP (item III.2.6), de acordo com o algoritmo B.13 para um processo gerente.

```
process buffer;
var B: array[0..maxbuf-1] of item;
    p, c: (0..maxbuf - 1);
    cont: (0..maxbuf);
procedure produz(x: item);
begin
  when cont < maxbuf → B[p] := x;
                       p := (p + 1) mod maxbuf;
                       cont := cont + 1
  end
end;
procedure consome(var x: item);
begin
  when cont > 0 → x := B[c];
                  c := (c + 1) mod maxbuf;
                  cont := cont - 1
  end
end;
begin p := 0; c := 0; cont := 0 end;
```

ALGORITMO B.13 - Utilização de DP para Gerenciar um Buffer Circular

Após terminar a execução do comando inicial, o processo buffer passa a executar as rotinas produz e consome, de acordo com as chamadas e com os guardas em cada rotina. Para um processo produtor se comunicar com o processo buffer, ele executa

call buffer.produz(item)

e um processo consumidor executa

call buffer.consome(item)

B.11- SOLUÇÃO EM ADA

A solução do problema da seção B.1 em Ada (item III. 2.7) pode ser realizada através da task apresentada no algoritmo B.14. Observe-se que a especificação da task divide-se em duas partes: uma de "interface", onde são definidas as entries acessíveis externamente, e uma de implementação, onde o corpo da task é escrito.

Um processo produtor tem acesso ao buffer através de

buffer.produz(item)

e um processo consumidor através de

buffer.consome(item)

```
task buffer is
  entry produz(x: in item);
  entry consome(x: out item);
end;
task body buffer is
  B: array(0..maxbuf-1) of item;
  p, c: integer range 0..maxbuf-1 := 0;
  cont: integer range 0..maxbuf := 0;
begin
  loop
    select
      when cont < maxbuf →
           accept produz(x: in item) do
             B[p] := x;
           end;
           p := (p + 1) mod maxbuf;
           cont := cont + 1
    or
      when cont > 0 →
           accept consome(x: out item) do
             x := B[c];
           end;
           c := (c + 1) mod maxbuf;
           cont := cont - 1
    end select
  end loop
end buffer;
```

ALGORITMO B.14 - Gerenciamento de um Buffer Circular em Ada

B.12- SOLUÇÃO EM EDISON

O problema apresentado na seção B.1 pode ser solucionado em Edison (item III.2.7) através do emprego do module programado no algoritmo B.15. As entidades marcadas com asteriscos (*) dentro do module são entidades exportadas, acessíveis

ao bloco que o envolve.

```
module buffer
  array B[0:maxbuf-1] (item)
  var p, c, cont: int
* proc produz(x: item)
  begin
    when cont < maxbuf do
      B[p] := x;
      p := (p + 1) mod maxbuf;
      cont := cont + 1
    end
  end
* proc consome(var x: item)
  begin
    when cont > 0 do
      x := B[c];
      c := (c + 1) mod maxbuf;
      cont := cont - 1
    end
  end
begin p := 0; c := 0; cont := 0 end
```

ALGORITMO B.15 - Gerenciamento de um Buffer Circular em Edison

O comando de inicialização é executado assim que o bloco que envolve o module é ativado. Para um processo produtor, o buffer é acessível através de

produz(item)

e para um processo consumidor, através de

consome(item)

APÊNDICE C

EXTENSÕES PROPOSTAS PARA PROGRAMAÇÃO CONCORRENTE EM PASCAL UCSD

C.1- INTRODUÇÃO

Este apêndice apresenta uma descrição informal das principais características incorporadas ao Sistema P em sua extensão proposta neste trabalho para processamento concorrente. As informações aqui contidas são um resumo do material apresentado nos capítulos IV e V desta documentação.

As seções a seguir apresentam as principais regras, procedimentos e tipos intrínsecos relacionados ao desenvolvimento de programas concorrentes no Sistema P.

C.2- PROGRAMAS CONCORRENTES E PROCESSOS CONCORRENTES

A identificação de um programa concorrente é realizada através da palavra reservada concurrent utilizada em seu cabeçalho. Assim, um programa concorrente deve começar por concurrent program ... , caso contrário é tratado como um programa seqüencial normal. Deve ser observado que apenas programas que comecem com a palavra concurrent têm acesso às ferramentas descritas no restante desta seção e nas seguintes.

Em um programa concorrente, processos concorrentes são também identificados através da palavra concurrent quando utilizada na especificação de uma rotina do tipo procedure. Processos concorrentes são, portanto, rotinas do tipo concurrent procedure ou concurrent segment procedure. Observe-se que nenhuma rotina do tipo function pode ser tratada como processo concorrente.

Processos concorrentes são ativados por comandos do

tipo

```
cobegin
  P1; P2; ... ; Pn
coend;
```

onde P1, P2, ..., Pn são processos concorrentes. Observe-se que os comandos existentes entre um cobegin e um coend têm necessariamente que ser chamadas a processos concorrentes, valendo também o contrário: processos concorrentes só podem ser chamados dentro de um par de comandos cobegin/coend.

A função dos comandos cobegin/coend é iniciar a execução simultânea de um certo número de processos concorrentes; um comando cobegin/coend termina quando tiverem terminado todos os processos que ele ativou. Assim, no exemplo acima, os processos P1, P2, ..., Pn são disparados simultaneamente e o comando seguinte ao cobegin/coend é executado apenas depois do término de todos os processos, de P1 a Pn.

A todo processo concorrente é associada uma prioridade para execução igual a 50 quando de sua ativação em um comando cobegin/coend. Durante a execução essa prioridade pode ser alterada pelo próprio processo através da rotina setpriority(p), onde p é a nova prioridade para execução, devendo ser uma expressão inteira não-negativa menor ou igual a 100 (o desrespeito a esses limites torna ineficaz a operação setpriority). Observe-se que um menor inteiro corresponde a uma maior prioridade.

C.3- SEMÁFOROS BINÁRIOS

Um semáforo binário S é declarado como

```
var S: semaphore;
```

e tem, em princípio, valor indeterminado. Para atribuir um valor a um semáforo binário S usa-se a operação initsem(S,v) ,

que atribui o valor v ao semáforo S. Observe-se que v tem que ser uma expressão lógica (valor true ou false), em função do fato de que S é um semáforo binário.

Uma chamada à operação P(S) sobre uma variável do tipo semaphore causa o bloqueio, na fila associada a S, do processo que a executou, caso o semáforo tenha valor false. Caso contrário o valor é transformado em false e o processo continua. A entrada na fila é ordenada pelo valor de prioridade para execução (ver seção C.2): os processos de execução mais prioritária ocupam os primeiros lugares na fila. Essa ordenação pode ser alterada através da explicitação de uma ordem na chamada a P. Assim, a chamada P(S,p), onde S é uma variável do tipo semaphore e p é uma expressão inteira qualquer, causa o bloqueio do processo (se for o caso) na ordem relativa crescente dada por p.

Uma chamada à operação V(S) sobre uma variável S do tipo semaphore libera o primeiro dos processos que possam estar bloqueados na fila dessa variável. Em caso de não haver nenhum, o semáforo recebe valor true.

Uma chamada à função empty(S) sobre uma variável S do tipo semaphore retorna valor true se não há processos encadeados na fila de S e valor false em caso contrário.

Chamadas a initsem, P, V e empty (sobre semáforos) excluem-se mutuamente no tempo.

C.4- FILAS DE EVENTOS

Uma fila de eventos Q é declarada como

```
var Q: queue;
```

Uma chamada à operação wait(Q,S), onde Q tem o tipo queue e S tem o tipo semaphore, causa o bloqueio, na fila Q, do processo que a executou. Ao mesmo tempo verifica o estado

da fila associada a S: se vazia, atribui a S o valor true; caso contrário, ativa o primeiro processo que nela se encontra. A entrada na fila Q é ordenada pelo valor de prioridade para execução (ver seção C.2): os processos de execução mais prioritária ocupam os primeiros lugares na fila. Essa ordenação pode ser alterada através da explicitação de uma ordem na chamada a wait. Assim, a chamada wait(Q,S,p), onde Q tem o tipo queue, S o tipo semaphore, e p é uma expressão inteira qualquer, causa o bloqueio do processo na ordem relativa crescente dada por p.

Uma chamada à operação signal(Q,S), onde Q é do tipo queue e S do tipo semaphore, libera o primeiro dos processos que possam estar bloqueados em Q. Em caso de não haver nenhum, é consultada a fila de S: se vazia, S recebe o valor true; se não, é ativado o primeiro processo de sua fila.

Uma chamada à função empty(Q) sobre uma variável Q do tipo queue retorna valor true se não há processos encadeados em Q e valor false em caso contrário.

O programador deve cuidar para que chamadas às operações wait, signal e empty (sobre filas de eventos) ocorram apenas dentro das regiões críticas protegidas pelo semáforo especificado nas operações wait e signal.

APÊNDICE D

EXTENSÕES DO INTERPRETADOR DE CÓDIGO P

Ao longo do capítulo V foram mencionados diversos procedimentos necessários à realização do núcleo do Pascal UCSD estendido e que foram incorporados aos processadores virtuais na categoria de rotinas standard. Este apêndice reproduz de maneira suscinta a descrição do funcionamento desses procedimentos , constituindo a especificação informal de uma interface entre uma implementação do núcleo sugerido no capítulo V e o processador virtual.

ALLOC(p)

Essa rotina é utilizada na implementação dos comandos cobegin/coend para especificação da concorrência entre processos. Ela é chamada assim que um processo é inicializado dentro de um desses comandos e tem a função de alocar um espaço de memória em que o processo recém-inicializado possa desenvolver-se quando de sua execução, transportando-o para ele. Assim, causa o retorno imediato da execução ao processo pai, que então pode inicializar os demais filhos antes de ativá-los simultaneamente. O estado inicial desse processo que foi inicializado é armazenado em p↑, seu descritor, onde p tem o tipo nodoptr definido na seção V.2.

BUSYWAIT(f)

Realiza espera ocupada sobre o flag f (variável do tipo boolean).

DISABLE

Inibe as interrupções do processador virtual.

DISPOSE(p)

Libera o espaço de memória ocupado por p↑, alocado dinamicamente, sendo p um apontador qualquer definido em Pascal . O valor de p torna-se igual a nil.

ENABLE

Habilita as interrupções do processador virtual.

SCHEDULE(p1, p2)

Suspende a execução do processo representado por p1↑ , substituindo-a pela de p2↑. As variáveis p1 e p2 têm o tipo nodoptr definido na seção V.2. Essa rotina tem algumas atribuições importantes: (i) se p1 = nil, então o processo p1↑ terminou e o espaço de memória que ele ocupava pode ser liberado; (ii) o estado do processador quando de sua chamada deve ser salvo em p1↑ e restaurado a partir de p2↑; (iii) ao final de sua execução, deve chamar o procedimento deixaRC do algoritmo V.1.

REFERÊNCIAS BIBLIOGRÁFICAS

1- BARBOSA, V.C.; VALLE, L.D. - Máquinas Stack, Boletim Técnico & Informativo do NCE/UFRJ, 9, 7/8(Ago./Set.1981), pp. 319-330.

2- BARNES, J.G.P. - An Overview of Ada, Software - Practice and Experience, 10, 11(November 1980), pp. 851-887.

3- BOWLES, K.L. - Beginner's Guide for the UCSD Pascal System, BYTE Publications, Inc., Martinsville, 1980.

4- CAMPBELL, R.H.; HABERMANN, A.N. - The Specification of Process Synchronization by Path Expressions, In Lecture Notes in Computer Science Series No. 16 (Operating Systems), Springer-Verlag, Berlin, 1974, pp. 89-102.

5- COFFMAN Jr., E.G.; ELPHICK, M.J.; SHOSHANI, A. - System Deadlocks, Computing Surveys, 3, 2(June 1971), pp. 67-78.

6- CONWAY, M. - A Multiprocessor System Design, Proceedings of the AFIPS 1963 Fall Joint Computer Conference, 24, Spartan Books, New York, 1963, pp. 139-146.

7- DIJKSTRA, E.W. - Co-operating Sequential Processes, In F. Genuys (Ed.), Programming Languages, Academic Press , New York, 1968, pp. 43-112.

8- DIJKSTRA, E.W. - Hierarchical Ordering of Sequential Processes, In Operating Systems Techniques, Academic Press, New York, 1972, pp. 72-93.

9- DIJKSTRA, E.W. - Guarded Commands, Nondeterminacy and Formal Derivation of Programs, Communications of the ACM, 18, 8(August 1975), pp. 453-457.

10- DIJKSTRA, E.W. - A Discipline of Programming, Prentice - Hall, Inc., Englewood Cliffs, 1976.

11- GOOS, G. - Some Basic Principles in Structuring Operating Systems, In Operating Systems Techniques, Academic Press, New York, 1972, pp. 94-113.

12- HANSEN, P.B. - The Nucleus of a Multiprogramming System, Communications of the ACM, 13, 4(April 1970), pp. 238-241.

13- HANSEN, P.B. - Structured Multiprogramming, Communications of the ACM, 15, 7(July 1972), pp. 574-578.

14- HANSEN, P.B. - A Comparison of Two Synchronizing Concepts, Acta Informatica, 1, 3(1972), pp. 190-199.

15- HANSEN, P.B. - Operating System Principles, Prentice-Hall, Inc., Englewood Cliffs, 1973.

16- HANSEN, P.B. - The Programming Language Concurrent Pascal, IEEE Transactions on Software Engineering, 1, 2 (June 1975), pp. 199-207.

17- HANSEN, P.B. - The Architecture of Concurrent Programs , Prentice-Hall, Inc., Englewood Cliffs, 1977.

18- HANSEN, P.B.; STAUNSTRUP, J. - Specification and Implementation of Mutual Exclusion, IEEE Transactions on Software Engineering, 4, 5(September 1978), pp. 365-370.

19- HANSEN, P.B. - Distributed Processes: a Concurrent Programming Concept, Communications of the ACM, 21, 11(November 1978), pp. 934-941.

20- HANSEN, P.B. - Edison - A Multiprocessor Language, Software - Practice and Experience, 11, 4(April 1981), pp. 325-361.

21- HANSEN, P.B. - The Design of Edison, Software - Practice and Experience, 11, 4(April 1981), pp. 363-396.

22- HANSEN, P.B. - Edison Programs, Software - Practice and Experience, 11, 4(April 1981), pp. 397-414.

23- HAYNES, L.S.; LAU, R.L.; SIEWIOREK, D.P.; MIZELL, D.W. - A Survey of Highly Parallel Computing, Computer, 15, 1 (January 1982), pp. 9-24.

24- HOARE, C.A.R. - Towards a Theory of Parallel Programming, In Operating Systems Techniques, Academic Press, New York, 1972, pp. 61-71.

25- HOARE, C.A.R. - Monitors: an Operating System Structuring Concept, Communications of the ACM, 17, 10 (October 1974), pp. 549-557.

26- HOARE, C.A.R. - Communicating Sequential Processes, Communications of the ACM, 21, 8(August 1978), pp. 666-677.

27- HOLT, R.C.; GRAHAM, G.S.; LAZOWSKA, E.D.; SCOTT, M.A. - Structured Concurrent Programming with Operating Systems Applications, Addison-Wesley Publishing Company , Inc., Reading, 1978.

28- JENSEN, K.; WIRTH, N. - Pascal User Manual and Report, Springer-Verlag, New York, 1975.

29- KEEDY, J.L. - On Structuring Operating Systems with Monitors, The Australian Computer Journal, 10, 1 (February 1978), pp. 23-27.

30- KESSELS, J.L.W. - An Alternative to Event Queues for Synchronization in Monitors, Communications of the ACM , 20, 7(July 1977), pp. 500-503.

31- LEDGARD, H. - Ada - An Introduction, In Ada Reference Ma-

nual (U.S. DoD, July 1980), Springer-Verlag, New York, 1981.

32- MOTTA R.L., J. - Relatório Técnico do CEPEL, em preparo.

33- SEGRE, L.; SANTOS, S.M. - O Conceito de Monitor como Instrumento de Sincronização em Programação Concorrente , Relatórios Técnicos do Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, ES 03-81, Rio de Janeiro, 1981.

34- SEGRE, L. - Mecanismos de Comunicação para Processos Concorrentes, Relatórios Técnicos do Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, ES 09-81, Rio de Janeiro, 1981.

35- SHAW, A.C. - The Logical Design of Operating Systems, Prentice-Hall, Inc., Englewood Cliffs, 1974.

36- SHILLINGTON, K.A.; ACKLAND, G.M. (Eds.) - UCSD Pascal Version I.5, University of California, San Diego, 1978.

37- SHRIRA, L.; FRANCEZ, N. - An Experimental Implementation of CSP, 2nd International Conference on Distributed Computing Systems, Paris, 8-10 April 1981, IEEE, New York, 1981, pp. 125- 136.

38- STERLING, T.L. - Parallel Computer Processing for Power Electronic Networks Simulation, M.S./E.E. Thesis, MIT , Cambridge, 1981.

39- WEITZMAN, C. - Distributed Micro/Minicomputer Systems, Prentice-Hall, Inc., Englewood Cliffs, 1980.

40- WELSH, J.; LISTER, A.; SALZMAN, E.J. - A Comparison of Two Notations for Process Communication, In J. Tobias (Ed.), Lecture Notes in Computer Science Series No. 79, Springer-Verlag, Berlin, 1980, pp. 225-254.

41- WELSH, J.; LISTER, A. - A Comparative Study of Task Communication in Ada, Software - Practice and Experience, 11, 3(March 1981), pp. 257-290.

42- WIRTH, N. - The Programming Language Pascal, Acta Informatica, 1, 1(1971), pp. 35-63.

43- WIRTH, N. - Modula: a Language for Modular Multiprogramming, Software - Practice and Experience, 7, 1 (January 1977), pp. 3-35.