UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

CURSO DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

88/66

IMPLEMENTAÇÃO DO SISTEMA PASCAL CONCORRENTE NO COMPUTADOR LABO-8034

por

GIL CARLOS RODRIGUES MEDEIROS

Tese submetida como requisito parcial para

a obtenção do grau de Mestre em

Ciência da Computação

Prof. Carlos Alberto Heuser

Prof. Simão Sirineo Toscani

Orientadores

UFRGS
BIBLIOTECA
CPD/PGCC

SABi
UFRGS
05223696

Porto Alegre, fevereiro de 1981.

AGRADECIMENTOS

Aos Professores Carlos Alberto Heuser e Simão Sirineo Toscani, pela orientação, incentivo e apoio técnico-científico.

Aos Auxiliares de Pesquisa Flávio Ferreira, Eduardo Arioli, Marcelo Lomando e Nelson Mattos, pela inestimável ajuda que prestaram, para o êxito deste trabalho.

Aos Professores Raul Weber e Dalcídio Cláudio, pelo apoio técnico na área de "hardware" e na definição dos algorítmos de ponto flutuante.

À LABO Eletrônica S.A., pela colaboração e pelo interesse demonstrado na pesquisa.

À Universidade Federal de Pelotas, pela oportunidade e pelo apoio financeiro, e aos colegas do Departamento de Matemática e Estatística (UFPEL), pelo incentivo.

Ao Curso de Pós-Graduação em Ciência da Computação, pelas condições de estudo e trabalho oferecidas; à CAPES e à FINEP, pelo apoio financeiro.

A todos aqueles que contribuiram, direta ou indiretamente, para a realização deste trabalho.

ATÉ 76

A Deus, Pai Supremo de toda ciência e sabedoria.

A minha esposa, Sandra Maria, em reconhecimento pela dedicação e compreensão.

A meus filhos, Daniel e Otávio.

A meus pais, Izabelino e Theresinha, em reconhecimento pela formação básica e pelos ensinamentos dos valores fundamentais da vida.

SINOPSE

O trabalho descreve a implementação do Sistema Pascal Concorrente de Brinch Hansen no minicomputador LABO-8034 (Nixdorf 8870/1).

O sistema é composto por dois compiladores - um para Pascal Concorrente e outro para Pascal Sequencial - e um sistema operacional básico, mono-usuário, que podem ser usados para desenvolvimento de outros sistemas operacionais, em linguagem de alto nível.

O trabalho de implementação inclui o estudo do sistema de Brinch Hansen, a criação de dois programas em Assembler do LABO-8034 - um "kernel", que executa as funções básicas do sistema, e um interpretador para o código virtual gerado pelos compiladores - e a definição e inicialização de um disco com o sistema.

ABSTRACT

The implementation of the Brinch Hansen's Concurrent Pascal System on the LABO-8034 computer (Nixdorf 8870/1) is described.

The system comprises two compilers - the Concurrent Pascal compiler and the Sequential Pascal compiler - and a basic operating system, single user oriented, suitable for the development of operating systems using a high level language (Pascal).

The work includes the understanding of the Brinch Hansen's system, the development of two assembly programs - a kernel that executes the basic functions of operating systems and an interpreter for the virtual code generated by the Pascal compilers - and also the definition and generation of the system.

SUMÁRIO

1. INTRODUÇÃO -- 1

2. O SISTEMA PASCAL DE BRINCH HANSEN ---------------------- 3

2.1 Pascal Concorrente -------------------------------- 4

2.1.1 Processos Concorrentes ---------------------- 4

2.1.2 Monitores ---------------------------------- 6

2.1.3 Classes ------------------------------------ 8

2.1.4 Programas Seqüenciais ---------------------- 8

2.2 O Ambiente de Programação do Pascal Concorrente ---- 9

2.2.1 O "Kernel" ---------------------------------- 9

2.2.1.1 Multiplexação de Processos ----------- 10

2.2.1.2 Implementação de Monitores ----------- 11

2.2.1.3 Entrada e Saída ----------------------- 12

2.2.2 O Interpretador ----------------------------- 12

2.2.3 Os Compiladores ----------------------------- 14

2.3 Sistema Operacional SOLO -------------------------- 15

3. CARACTERÍSTICAS DO HARDWARE --------------------------- 19

3.1 Sistema de Interrupções ---------------------------- 20

3.2 Características Limitantes ------------------------- 21

3.3 Configuração -- 21

4. O SISTEMA IMPLEMENTADO ------------------------------- 23

4.1 Características Gerais ----------------------------- 23

4.2 Estrutura da Memória Virtual ---------------------- 26

4.3 Periféricos Implementados ------------------------ 27

4.4 Alocação do Disco -------------------------------- 28

4.5 Entrada e Saída ---------------------------------- 29

4.6 Prioridades -------------------------------------- 30

4.7 Descritores de Processos ------------------------- 31

4.8 Código Virtual ----------------------------------- 32

4.9 Ponto Flutuante ---------------------------------- 33

5. KERNEL -- 35

5.1 Estruturas de Dados ------------------------------ 39

5.2 Funções Internas --------------------------------- 46

5.3 Primitivas --------------------------------------- 64

5.3.1 Primitivas para Controle de Processos ------- 66

5.3.2 Primitivas para Implementação de Monitores -- 70

5.3.3 Primitivas para Entrada e Saída ------------- 75

5.3.4 Primitivas Auxiliares ----------------------- 76

5.4 Tratamento de Interrupções ----------------------- 77

5.5 Periféricos -------------------------------------- 79

5.5.1 Disco Magnético ----------------------------- 81

5.5.2 Terminais Vídeo-Teclado --------------------- 84

5.5.3 Impressora ---------------------------------- 90

6. INTERPRETADOR --------------------------------------- 98

6.1 A Máquina Virtual -------------------------------- 98

6.2 As Instruções Virtuais --------------------------- 103

7. INICIALIZAÇÃO DO SISTEMA ---------------------------- 115

7.1 Carga do Sistema ---------------------------------- 115

7.2 Inicialização -------------------------------------- 117

8. CONCLUSÕES -- 119

APÊNDICE A -- 123

BIBLIOGRAFIA -- 126

LISTA DE FIGURAS

Figura 1 Memória virtual e seu acesso via registradores - 14

Figura 2 Processos e monitores ---------------------------- 17

Figura 3 Alocação da memória ------------------------------ 24

Figura 4 Segmentos de código ------------------------------ 24

Figura 5 Segmentos de dados ------------------------------- 24

Figura 6 Segmento de dados -------------------------------- 25

Figura 7 Pilha -- 25

Figura 8 Variáveis permanentes ---------------------------- 25

Figura 9 Variáveis temporárias ---------------------------- 25

Figura 10 Memória virtual --------------------------------- 27

Figura 11 Segmento comum ---------------------------------- 27

Figura 12 Formato e representação de números de ponto flutuante -------------------------------- 34

Figura 13 Diagrama de execução das classes do "kernel" --- 38

LISTA DE TABELAS

Tabela 1 Funções de controle de vídeo e códigos -------- 86

Tabela 2 Registradores da máquina virtual -------------- 99

Tabela 3 Operações sobre conjuntos --------------------- 108

Tabela 4 Chamadas de rotinas --------------------------- 109

Tabela 5 Operações especiais --------------------------- 112

Tabela 6 Tempos de execução de um mesmo programa em diversos sistemas -------------------------- 122

LISTA DE ABREVIATURAS

ALM	"Asynchronous Line Module"
ATT	"Address Translation Table"
COMPRB	comprimento em bytes
COMPRW	comprimento em words
conj	conjunto
DESL	deslocamento
DIST	distância
DMA	"Direct Memory Access"
DWS	"Display Work Station"
elem	elemento
fig.	figura
Hz	Hertz
IPL	"Initial Program Load"
KB	"Kilo Bytes" (= 1024 Bytes)
KW	"Kilo Words" (= 1024 palavras)
MAX	máximo
MB	"Mega Bytes"
MIN	mínimo
ms	milissegundo
PC	"Program Counter"
PF	ponto flutuante
UCP	Unidade Central de Processamento

1 INTRODUÇÃO

O objetivo deste trabalho é implementar, no minicomputador LABO-8034, um sistema de programação que possua facilidades para a definição e desenvolvimento de sistemas operacionais. Tais facilidades devem satisfazer as necessidades da programação concorrente, com todas as suas implicações:

- criação de processos,
- comunicação entre processos,
- controle de entrada e saída, etc.

O Sistema Pascal Concorrente desenvolvido por Brinch Hansen[1] de 1972 a 1975, possui estas facilidades, ainda acrescidas de algumas características, como

- simplicidade,
- confiabilidade,
- adaptabilidade,
- eficiência,
- portabilidade e
- generalidade,

que o tornam um sistema de boa qualidade para o fim desejado. Este sistema foi implementado, inicialmente, num computador PDP-11/45. Apenas uma pequena parte do sistema, aproximadamente 4%, é escrita em linguagem dependente de máquina (Assembler ou outra linguagem de baixo nível).

A metodologia para implantação deste sistema no computador destino é baseada no transporte do "software", o que exige uma simples cópia dos programas escritos em linguagem de alto nível e a implementação dos dois seguintes programas

dependentes de máquina:

- um interpretador, que simula a máquina virtual cujo código é gerado pelos compiladores Pascal do sistema, e

- um "kernel" para sistemas operacionais, que executa algumas funções básicas definidas para o sistema.

Nos capítulos 2 e 3, são apresentados, respectivamente, o sistema original, com as caraterísticas básicas que o identificam, e o "hardware" do computador utilizado (somente as características mais importantes para a implementação). Detalhes sobre o sistema, mais vinculados com a implementação, são apresentados no capítulo 4. Os capítulos 5 e 6 dedicam-se a implementação do "kernel" e do interpretador, respectivamente. Para completar o trabalho, descreve-se os procedimentos definidos para a inicialização do sistema no computador, no capítulo 7, e faz-se uma comparação do desempenho do sistema com o alcançado por Brinch Hansen e outros pesquisadores.

2 O SISTEMA PASCAL DE BRINCH HANSEN

Com o crescimento da complexidade do "software" básico, surgiu, entre os pesquisadores, a necessidade de utilizar linguagens de mais alto nível na programação de sistemas. Pascal é uma linguagem abstrata, que oculta do programador os detalhes irrelevantes da máquina. A linguagem é de fácil entendimento por programadores que já estejam familiarizados com Fortran, Algol, Cobol ou PL/I. Além disso, é, também, bastante eficiente para a programação de sistemas. Daí, ser o Pascal, atualmente, uma das linguagens mais empregadas nesta área.

Brinch Hansen utilizou o Pascal para definição e desenvolvimento de um sistema de programação concorrente[1]. A partir da linguagem original, desenvolveu uma nova linguagem de programação - Pascal Concorrente - cujo compilador foi escrito por Al Hartman[2] em Pascal Seqüencial. Do compilador de Pascal Concorrente, foi derivado um compilador para Pascal Seqüencial, compatível com a estrutura da linguagem definida. O Pascal Concorrente é uma extensão do Pascal Seqüencial, na qual foram utilizados os conceitos de processos concorrentes e monitores. Assim como o Pascal Seqüencial reduz o esforço de programação de compiladores e outros programas semelhantes (seqüenciais), o Pascal Concorrente assegura este mesmo auxílio na programação de sistemas operacionais.

Este sistema de programação concorrente já foi utilizado por diversos pesquisadores, na criação e implementação de sistemas operacionais, em diferentes

computadores.[3,4]

2.1 Pascal Concorrente

O Pascal Concorrente destina-se a programação de processos paralelos que interagem em estruturas de dados compartilhados através de monitores.

Esta linguagem é própria para a construção de sistemas operacionais, devido a seus mecanismos de multiprogramação. Além disso, por ser uma linguagem de alto nível, reduz o esforço de programação, facilitando também a manutenção e a atualização dos sistemas com ela implementados.

Por razões de segurança[1], algumas das características do Pascal Seqüencial, como "pointers", por exemplo, não são permitidas no Pascal Concorrente. Além disso, muitos dos erros de programação que dependem do tempo de execução são prevenidos pelo compilador.[1,2] Este verifica que as variáveis privadas de um processo não sejam acessíveis a outros processos. A única maneira de comunicação entre processos é através de monitores. Um monitor define todas as operações possíveis sobre uma estrutura de dados compartilhados.

2.1.1 Processos Concorrentes

Um processo é uma atividade assíncrona, tal como a execução de um programa pela UCP.[5] Processos concorrentes são

programas que, executados simultaneamente, competem por recursos comuns (máquina e acesso a variáveis). Um conjunto de processos concorrentes que se comunicam, cooperando uns com os outros, para a execução de uma tarefa global bem determinada, forma um programa concorrente.

Há sistemas em que diversos programas concorrentes podem ser executados simultaneamente. No sistema de Brinch Hansen, todos os processos sendo executados pelo computador são componentes de um único programa concorrente. Além disso, um programa Pascal Concorrente consiste de um número fixo de processos, monitores e classes, e sua alocação é estática. Estes componentes e suas estruturas de dados existirão para sempre após a inicialização do sistema[1]. Assim, um programa concorrente só pode ser extendido (ou reduzido) por recompilação.

O escopo mais externo de um programa concorrente é um processo anônimo e sem parâmetros, chamado de "processo inicial". Seguindo a hierarquia de processos, só existe mais um nível, pois todos os demais processos são disparados diretamente pelo processo inicial. O propósito deste processo é, simplesmente, inicializar todos os outros componentes do sistema, sendo ele automaticamente inicializado após a carga do programa.

2.1.2 Monitores

Um monitor é um tipo de dado abstrato que define uma estrutura de dados (variáveis compartilhadas) e as operações (rotinas) que podem ser executadas sobre estes dados por processos concorrentes.[1,5,6,7]

Uma das principais diferenças entre processos e monitores consiste em que os primeiros são componentes ativos do sistema, pois executam programas bem determinados, e os últimos são componentes passivos, já que se constituem de um conjunto de rotinas que nada fazem até que sejam chamadas pelos processos.[1]

Os monitores possuem, ainda, uma característica adicional, que lhes confere sua importância em programação concorrente. Só uma rotina (das que definem as operações sobre os dados compartilhados) de um monitor pode ser executada de cada vez. Isto garante ao processo que executa o monitor, acesso exclusivo à estrutura de dados, enquanto a rotina chamada está sendo executada.

Além das estruturas de dados compartilhados, também a ordem na qual processos competitivos usam recursos físicos comuns, pode ser controlada por monitores.

As operações definidas por um monitor podem ser usadas para sincronizar processos e trocar dados entre eles.

No Pascal Concorrente foi incluido um tipo de dados simples, chamado QUEUE (fila), que só pode ser usado pelas rotinas de um monitor, e serve para controlar um "medium-term scheduling" de processos (o "short-term scheduling" de chamadas

simultâneas a um monitor é controlado pela máquina que executa os processos concorrentes). Numa mesma fila, somente um processo pode estar esperando a ocorrência de algum evento. Um processo deve ser retardado numa fila de espera, sempre que não consegue, por algum motivo, executar totalmente a operação que requisitou ao monitor. As operacões definidas sobre variáveis do tipo QUEUE são DELAY e CONTINUE ("wait" e "signal"[5,7]). Com a primeira, uma rotina de um monitor pode colocar o processo que a chamou numa determinada fila. Deste modo, o processo perde o acesso exclusivo às variáveis do monitor, até que um outro processo chame o mesmo monitor e execute, em uma de suas rotinas, a operacão CONTINUE sobre aquela mesma fila. Quando uma rotina de um monitor executa a operação CONTINUE sobre uma fila, o processo que a chamou retorna do monitor e, se existe algum processo esperando na fila, este reassume a execução do monitor, a partir do ponto em que havia sido retardado.

Filas de múltiplos processos podem ser facilmente implementadas como "arrays" cujos elementos são do tipo QUEUE (fila de um só processo).

Para completar um monitor, define-se, ainda, uma operação inicial, que é executada uma só vez, quando a estrutura de dados é criada, na inicialização do monitor. Esta é realizada através do comando INIT, que também serve para inicializar processos e classes.

2.1.3 Classes

Classes são tipos de dados semelhantes a monitores. Uma classe define uma estrutura de dados e as operações que podem ser executadas sobre estes dados. No entanto, uma classe só pode ser acessada por um único processo ou monitor, e o acesso exclusivo às variáveis de uma classe, é completamente garantido em tempo de compilação. Uma classe só pode ser inicializada uma vez.

Este tipo de dados abstrato proporciona acesso controlado aos dados de um componente do sistema. Esta possibilidade de dividir um componente do sistema em diversos componentes menores, torna a linguagem Pascal Concorrente útil para o projeto de sistemas hierárquicos.

2.1.4 Programas Seqüenciais

A maioria dos programas de um sistema operacional Pascal Concorrente são escritos em Pascal Seqüencial. Assim, compiladores, editores, "drivers" de entrada e saída, interpretadores para linguagens de controle de tarefas, alocadores de disco e programas de usuários são, todos, programas seqüenciais.

Os processos de um programa concorrente tem a capacidade de iniciar a execução de programas seqüenciais compilados separadamente. Um processo que controla a execução de tais programas seqüenciais é chamado de "job process". Este

tipo de processo deve possuir uma declaração do programa seqüencial, indicando um identificador para o programa, uma lista de parâmetros e um conjunto de direitos de acesso a entradas em rotinas especiais definidas no processo. Estas rotinas especiais só são acessíveis pelos programas seqüenciais iniciados pelo processo que as define.

2.2 O Ambiente de Programação do Pascal Concorrente

O ambiente para programação com Pascal é um sistema formado pelos seguintes elementos:

- um "kernel",
- um interpretador e
- os compiladores para programas escritos em Pascal Concorrente e Seqüencial.

2.2.1 O "Kernel"

O "kernel" do sistema Pascal Concorrente é um programa dependente de máquina, normalmente escrito em linguagem "assembler" do computador utilizado, que executa as funções básicas de um sistema operacional, no seu mais baixo nível. Entre estas funções estão a multiplexação do processador entre os diversos processos concorrentes e a concessão a estes de acesso exclusivo a monitores, bem como a execução de instruções de entrada e saída.

2.2.1.1 Multiplexação de Processos

Cada processo é representado, no "kernel", por um registro descritor, que mantém o estado dos registradores utilizados por este processo, enquanto o mesmo não está sendo executado. Este registro é alocado na memória do "kernel" no momento da inicialização do processo. Qualquer referência a um processo é feita através do endereço de seu registro descritor.

O computador executa um processo de cada vez. Os demais processos ficam esperando a ocorrência de algum evento em diversas filas de múltiplos processos. Uma fila de múltiplos processos é representada por uma seqüência de referências a registros de processos. O processo sendo executado é dito estar no estado "running"; uma referência a seu registro é mantida no "kernel". Os outros processos aptos a usarem o processador estão no estado "ready" e ficam numa fila de mesmo nome, esperando a sua vez de passarem a "running". Os processos que não podem concorrer pela UCP, estão "bloqueados" esperando a ocorrência de algum evento, na fila correspondente. Estes eventos podem ser:

- término de uma operação de entrada/saída requisitada (bloqueado por IO),
- incremento do contador de segundos pelo relógio (bloqueado por WAIT),
- liberação de entrada em um monitor (bloqueado por chamada de uma sub-rotina de entrada no monitor) ou
- liberação de processos pelo operador, via uma tecla específica (BELL) do teclado (bloqueado pela operação

CONTROL de IO para terminal).

Alguns processos, ainda, podem ficar bloqueados dentro de monitores (DELAY), em filas de um só processo (tipo QUEUE) a espera de que algum outro processo, tendo entrado no mesmo monitor, execute a operação CONTINUE sobre a mesma fila.

2.2.1.2 Implementação de Monitores

Cada variável do tipo MONITOR é representada no "kernel" por uma estrutura de dados chamada "porta". Esta estrutura é formada por uma variável booleana, que indica se o acesso está livre ou não, e uma fila de processos esperando para entrar no monitor. Basicamente, duas operações podem ser executadas sobre uma estrutura porta: ENTER e LEAVE. A operação ENTER é executada sobre uma porta, quando um processo chama o "kernel" para entrar no monitor correspondente. Se o acesso está livre, o processo entra no monitor e bloqueia o acesso para outros processos, senão, ele perde a UCP e é colocado na fila de espera daquela porta. A operação LEAVE é executada quando um processo sai de um monitor. A porta correspondente é verificada; se há processos esperando na fila, o acesso é dado a um só deles, em caso contrário (fila vazia), o acesso fica livre. Como estas operações são executadas no "kernel" e este é executado de forma indivisível, isto garante o acesso exclusivo a monitores.

Quando uma variável monitor é inicializada, o "kernel" aloca e inicializa uma porta em sua memória e passa o

endereço desta ao processo que o chamou. O monitor é identificado através desta referência a sua porta.

2.2.1.3 Entrada e Saída

O "kernel" também é chamado para executar o comando de entrada e saída (IO) do Pascal Concorrente. Tal comando é traduzido na chamada de uma rotina do "kernel" que inicia a transferência de dados e bloqueia o processo que o executa até a operação ser terminada. Ao término da operação, o periférico utilizado provoca uma interrupção da UCP, o que causa a chamada de uma outra rotina do "kernel", que libera o processo bloqueado a sua espera para novamente concorrer pela UCP. O "kernel" não garante que somente um processo de cada vez tente usar um periférico. Isto deve ser controlado pelo sistema operacional (escrito em Pascal Concorrente).

2.2.2 O Interpretador

Os compiladores de Pascal Concorrente e Seqüencial geram código para uma máquina virtual. A máquina virtual e sua implementação no Sistema LABO estão descritas em capítulo específico neste texto.

O código virtual foi projetado diretamente para aproximar uma linguagem de alto nível das linguagens de máquinas existentes. Ele foi usado no sistema Pascal por tornar

a geração de código mais simples e os compiladores portáveis.

Na máquina virtual definida para Pascal existem, aproximadamente, 50 instruções virtuais diferentes. Algumas destas instruções, que envolvem modos de endereçamento e tipos de dados diversos, são subdivididas em diferentes códigos de operação, para tornar mais rápida a interpretação do código virtual por "software". Isto expande para 112 o conjunto de códigos de operação. Alguns deles são utilizados somente pelo Pascal Concorrente, outros somente pelo Seqüencial; a maioria é comum a ambos.

As construções da linguagem e o código virtual correspondente foram definidos por Hartmann, através de grafos de sintaxe, na descrição do compilador para Pascal Concorrente[2].

O interpretador é um programa escrito em linguagem Assembler que simula a execução das instruções da máquina virtual no computador utilizado. Ele consiste de uma série de rotinas, que executam as instruções virtuais, e uma tabela de operações, definindo o endereço destas rotinas na memória.

Uma instrução virtual consiste de um código de operação seguido, possivelmente, por alguns argumentos, cada um ocupando uma palavra de memória. O código de operação é um índice usado na tabela de operações para desviar para a rotina correspondente. A maioria dos operandos das instruções são retirados da pilha de operação ("stack") da máquina virtual, e o resultado da operação, geralmente, é a inserção de novos valores na pilha, para uso por outras instruções.

A máquina virtual possui uma série de registradores, que são usados pelo interpretador, para

executar o código virtual de um processo. Alguns deles são registradores de trabalho e seus valores só tem significado durante a execução de uma única instrução virtual. Os outros tem funções fixas durante toda a execução de um processo (ver figura 1). Quatro destes servem para endereçar os dados do processo. O "heap top" H define o tamanho corrente do "heap" (descrito adiante, na seção 4.1) e os outros (B, G e S) são usados para endereçar a pilha.

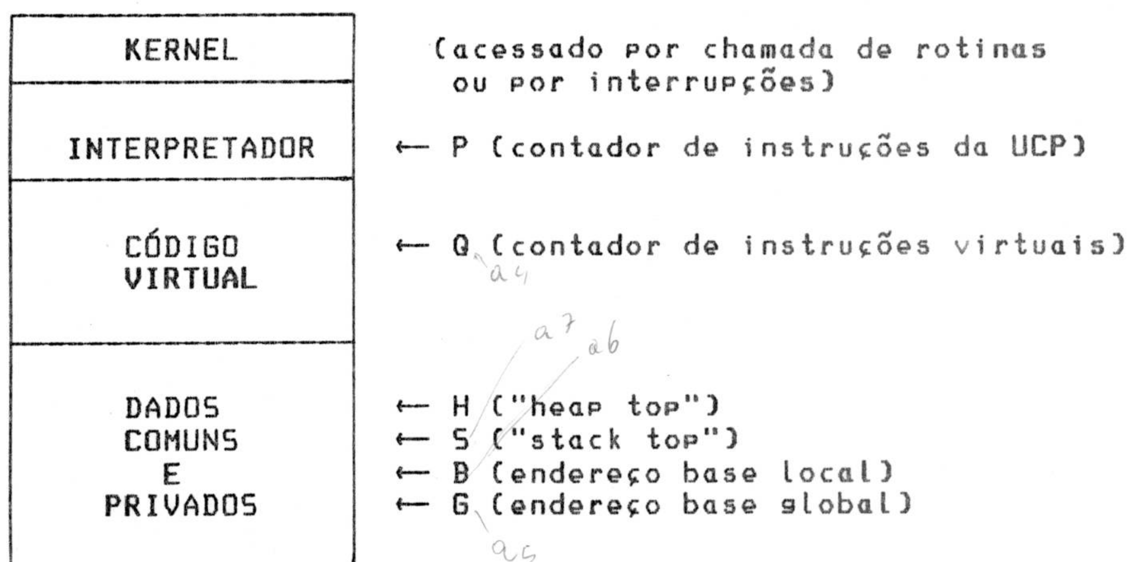


Fig. 1: Memória virtual e seu acesso via registradores.

2.2.3 Os Compiladores

Existem dois compiladores Pascal, um para programas concorrentes e outro para programas seqüenciais. Ambos foram escritos na linguagem Pascal Seqüencial.

O trabalho de compilação é dividido em 7 passos, cada um executado por um programa seqüencial. Quatro arquivos

são utilizados durante a compilação de um programa:

- arquivo com o texto fonte;
- listagem resultante;
- arquivo de código intermediário, entrada para o passo corrente(saída do passo anterior - não usado no passo 1);
- arquivo de código intermediário, saída do passo corrente (entrada para o passo seguinte ou código objeto final, no último passo).

Cada passo dos compiladores executa funções específicas, quais sejam, nesta ordem:

- análise de símbolos;
- análise sintática;
- análise de escopo;
- análise de declarações;
- análise de comandos;
- seleção de código e
- montagem do código virtual final.

Uma descrição dos passos e outras informações pertinentes aos compiladores encontram-se, resumidamente, em Brinch Hansen[1] e, detalhadamente, em Hartmann[2].

2.3 Sistema Operacinal SOLO

Não interessa

SOLO é um dos primeiros sistemas operacionais escritos na linguagem de programação Pascal Concorrente. É um sistema operacional de um só usuário, simples, mas útil para o desenvolvimento de programas Pascal.

O sistema SOLO proporciona a edição, compilação e armazenamento de programas seqüenciais e concorrentes. Estes programas podem acessar aos diferentes periféricos de diversas maneiras: por caracter, por página ou por acesso direto ao dispositivo. Processos concorrentes específicos controlam cada uma das atividades básicas de um programa: entrada, processamento e saída de arquivos de dados. Qualquer programa pode chamar um outro programa, ou a si próprio, recursivamente, passando e recebendo parâmetros. A maior parte da proteção do sistema é efetuada em tempo de compilação, através da verificação dos direitos de acesso; apenas uma pequena parte da proteção é verificada em tempo de execução, sem usar mecanismos de "hardware". Este sistema é um dos mais característicos exemplos de programas concorrentes hierárquicos, implementados por meio de tipos de dados abstratos, como monitores, classes e processos.

O sistema operacional consiste de um programa concorrente que controla a execução de diversos programas seqüenciais, cada um com uma função específica:

- "drivers" para os periféricos disponíveis;
- controle de tarefas;
- sistema de arquivos em disco;
- controle de entrada e saída; e
- diversos utilitários, além dos compiladores.

Um comando do sistema, START, pode ser usado para iniciar a execução de outros programas concorrentes armazenados em disco. Através da tecla BELL (CONTROL G) do teclado do terminal console, o operador pode desativar a tarefa

correntemente sendo executada, o que provoca a reinicialização do sistema SOLO.

A figura 2 proporciona uma visão global e resumida do fluxo de dados no sistema, através dos processos e monitores ("buffers").

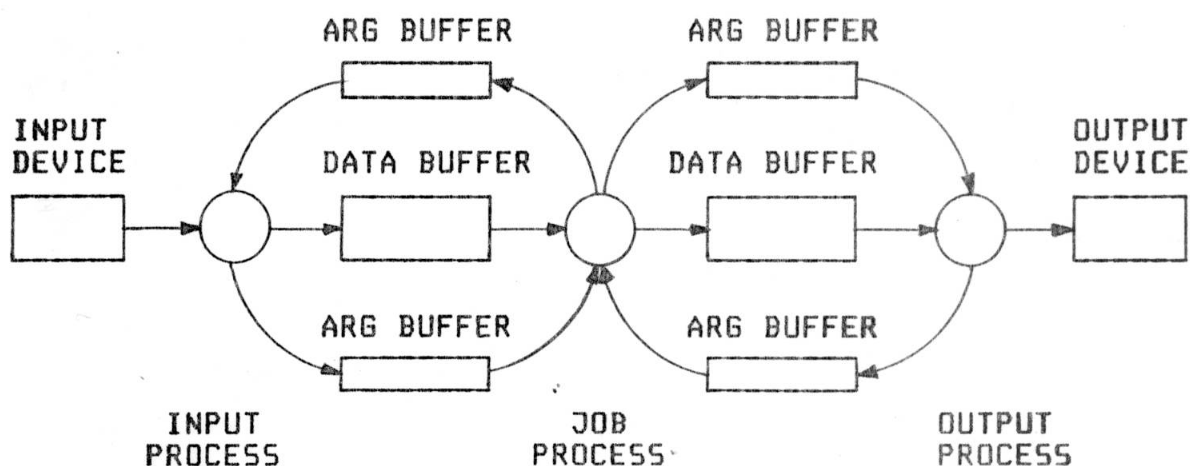


Fig. 2: Processos e monitores.

Toda a comunicação entre os processos é feita através de "buffers" de argumentos. O processo de tarefas ("job process") faz todo o processamento e inicia entradas e saídas pelos respectivos processos, enviando a estes um argumento que indica o nome do dispositivo de entrada/saída ou do arquivo em disco. O processo correspondente executa a operação e, ao término desta, envia ao processo de tarefas um outro argumento, indicando como a operação foi executada (fim normal ou com erros). Os dados são transportados de um processo a outro através de "buffers" de dados (512 bytes cada).

Todos os detalhes sobre o sistema SOLO, e outros sistemas operacionais implementados com Pascal Concorrente, encontram-se na bibliografia referenciada.[1,4] O intuito único

desta seção foi dar uma introdução ao SOLO, em virtude deste sistema operacional ser referenciado em diversas situações no decorrer deste texto.

3 CARACTERÍSTICAS DE HARDWARE

O equipamento ao qual se destina o "software" desenvolvido neste trabalho é o minicomputador LABO-8034 (Nixdorf-8870/1).

As características gerais e funcionais deste computador encontram-se descritas na bibliografia referenciada.[8,9,10] Neste capítulo são abordadas apenas as características mais relevantes do "hardware" para o entendimento do sistema implementado.

A configuração do Sistema 8034 pode ser bastante variada, constituindo-se de

- processador,
- memória (64 a 256 KB),
- relógio interno com freqüência programável,
- discos magnéticos (de 1 a 4 "drives" com 2 discos de 5 MB, um fixo e um removível, ou de 1 a 2 "drives" com discos de 33 MB),
- terminais vídeo-teclado (até 8 terminais ligados diretamente, sem modens),
- fitas magnéticas (um controlador com, no máximo, 4 dispositivos),
- fita cassete (1 unidade),
- disco flexível (1 unidade),
- impressoras (até 2 seriais ou 1 de linhas e 1 serial),
- interface de comunicações (1 controlador de linha para comunicação via modens).

A memória é dividida em palavras de 16 bits. O processador usa endereços de 15 bits, o que lhe dá a capacidade de endereçar até 32 KW de memória virtual, sendo esta dividida em 64 páginas de 1 KB (512 palavras). As páginas que constituem um espaço virtual podem estar espalhadas na memória real, sendo ligadas e seqüenciadas através de uma "tabela de tradução de endereços" (ATT). Esta tabela é usada pela UCP para transformar um endereço virtual em endereço real de memória. Ela é constituida de registradores de "hardware" e pode ser carregada e descarregada por instruções especiais.

Existem 5 níveis de interrupção e um ou dois canais de acesso direto à memória. Um dos canais de DMA é dedicado ao controlador de discos magnéticos e o outro é usado por outros periféricos, como fita magnética, impressora, etc.

A UCP possui, além do PC ("Program Counter", com 15 bits), 4 registradores de 16 bits para uso em transferências de dados e para as operações de aritmética e lógica; dois destes registradores ainda podem ser usados como indexadores.

3.1 Sistema de Interrupções

O processador admite 5 níveis de interrupção, cujas causas são as seguintes:

- falta de energia;
- atraso no tempo de resposta ("time-out");
- erro de paridade;
- relógio e

- entrada/saída.

A prioridade, no atendimento às interrupções, decresce na ordem em que estão acima apresentadas. Não há prioridades entre as interrupções sinalizadas pelos diferentes dispositivos de entrada e saída.

3.2 Características Limitantes

O "hardware" deste minicomputador não possui instruções específicas para processamento de estruturas de dados especiais, como pilhas, por exemplo. Também não existem instruções para processamento de números de ponto flutuante ou de precisão extendida. Uma outra característica limitante para o desenvolvimento de "software" básico (sistemas de programacão) é a falta de interrupções programáveis ("traps") para comunicação com o "kernel" do sistema e execução de suas funções, ou para qualquer outra finalidade.

3.3 Configuração

O "software" desenvolvido destina-se a qualquer configuração do Sistema LABO-8034, em termos de UCP, memória e existência ou não dos periféricos programados. O sistema implementado se auto-reconfigura a cada vez que é carregado na memória, em função da configuração do "hardware" naquele momento.

A configuração mínima, no entanto, para o funcionamento do sistema, na versão implementada, é a seguinte:

- UCP e memória - 64 KB de memória com processador de memória virtual (ou seja, UCP com ATT para processamento de endereços), como configuração básica; contudo a capacidade de memória exigida depende dos sistemas operacionais que serão utilizados (o sistema SOLO necessita de 96 KB);
- Disco - um disco do tipo "cartridge" de 5 MB, com o sistema;
- Console - um terminal vídeo-teclado, do tipo DWS (DAP), ligado ao canal 0 do controlador ALM.

4 O SISTEMA IMPLEMENTADO

A implementação do Pascal Concorrente no LABO-8034 consiste no desenvolvimento dos dois programas escritos em linguagem Assembler que compõem o sistema: "kernel" e interpretador. A maior parte destes dois programas é simplesmente transportada, traduzindo-se o que foi escrito por Brinch Hansen, em Assembler do PDP-11/45, para o Assembler do LABO-8034 e ajustando-se a estrutura do "kernel" às características do novo equipamento, procurando melhorar, sempre que possível, a eficiência dos trechos mais críticos.

4.1 Características Gerais

Um programa Pascal Concorrente define um número fixo de processos. Durante a execução de um programa, a memória principal é subdividida em dois tipos de segmentos, cujos comprimentos são determinados na compilação:

- segmentos de código e
- segmentos de dados (ver figura 3).

Os segmentos de código são o código virtual gerado pelo compilador de Pascal Concorrente, o interpretador que executa o código virtual no LABO-8034, e o "kernel" que executa as funções básicas definidas para sistemas operacionais escritos com Pascal Concorrente (ver figura 4).

Os segmentos de dados são definidos pelo compilador, um para cada processo, e alocados na memória, em

tempo de execução, durante a inicialização do processo correspondente (ver figura 5).

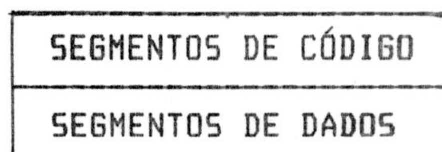


Fig. 3: Alocação da memória.

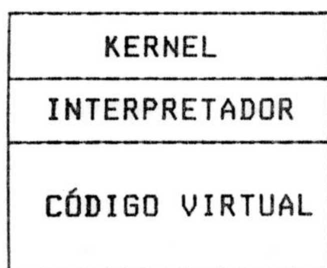


Fig. 4: Segmentos de código.

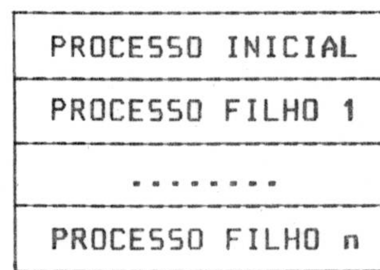


Fig. 5: Segmentos de dados.

O segmento de dados de um processo contém a pilha e o "heap" do processo, os quais crescem em sentidos opostos, conforme mostram as flechas na figura 6. Um erro (STACKLIMIT ou HEAPLIMIT) ocorre quando uma tentativa é feita de aumentar um deles sem que haja espaço suficiente.

A pilha contém as variáveis permanentes de um processo e suas variáveis temporárias usadas em sub-rotinas (ver figura 7). O "heap" de um processo só pode ser usado pelos programas seqüenciais que são executados por esse processo. Sua utilização é feita através de "pointers", e a alocação dos dados, através da função padrão NEW do Pascal Seqüencial.

Variáveis permanentes são as variáveis globais dos processos, monitores e classes, e os parâmetros com que estes

foram inicializados. Para os monitores, há, ainda, o endereço de sua porta no "kernel" (ver seção 2.2.1.2).

Uma chamada de sub-rotina determina a alocação na pilha de suas variáveis temporárias, que são representadas pelos parâmetros, variáveis, resultados temporários e uma ligação dinâmica da sub-rotina ao contexto em que é chamada (ver figura 9).

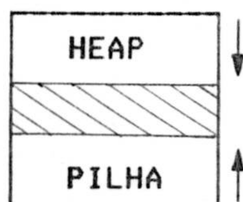


Fig. 6: Segmento de dados.

Fig. 7: Pilha.

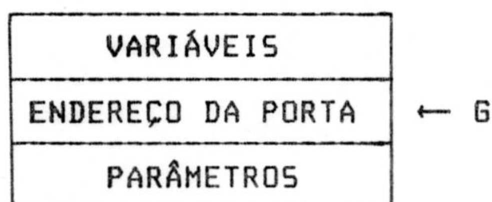


Fig. 8: Variáveis permanentes.

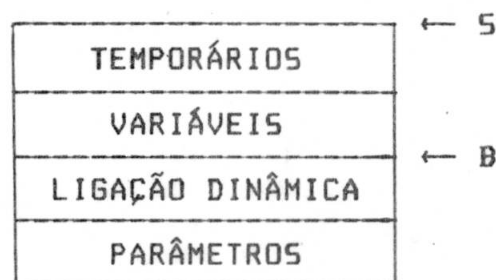


Fig. 9:Variáveis temporárias.

Um processo só pode operar sobre um conjunto de variáveis permanentes e um conjunto de variáveis temporárias de cada vez. As permanentes são endereçadas relativamente ao endereço base global G e as temporárias em relação ao endereço base local B e ao "stack top" S (temporários) (ver figuras 8 e 9).

A área para ligação dinâmica contém uma cópia dos valores dos registradores S, B, G e Q da máquina virtual, usados pelo processo antes de chamar a sub-rotina e, ainda, o número da linha (do programa fonte) corrente para facilitar a localização de erros em tempo de execução. Os registradores são restaurados, com os valores mantidos na ligação dinâmica, ao término da execução da sub-rotina (antes do retorno ao ponto de chamada).

4.2 Estrutura da Memória Virtual

Cada processo dispõe de um espaço virtual de 32 KW, porém só tem acesso ao espaço necessário e suficiente (<=32 KW) para sua execução, que é determinado em tempo de compilação.

A memória virtual de um processo é formada por um segmento comum, que é compartilhado por todos os processos, e pelo seu próprio segmento privado de dados (ver figura 10).

O segmento comum consiste do "kernel", do interpretador, do código virtual do programa concorrente e do segmento de dados do processo inicial (ver figura 11). Assim, o processo inicial não possui segmento privado de dados. A razão para o "kernel" fazer parte do segmento comum (e não de um segmento separado) é o alto custo, em tempo de execução, exigido para a troca de espaço virtual, o que seria necessário cada vez que o "kernel" fosse acessado, caso não fizesse parte deste segmento. Portanto, todos os segmentos de código fazem parte do segmento comum.

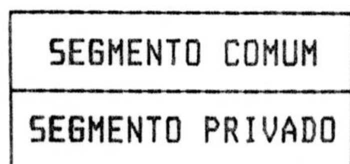


Fig. 10: Memória virtual.

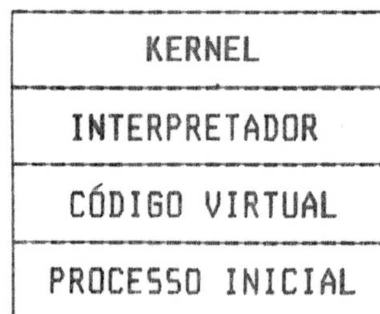


Fig. 11: Segmento comum.

4.3 Periféricos Implementados

Os periféricos são definidos por classes do "kernel". Na implementação inicial, são definidos os seguintes dispositivos de entrada e saída:

a) Discos magnéticos - é programado apenas o controlador para discos de 5MB, com setores (páginas de discos) de 256 palavras (512 "bytes"), 12 setores por trilha, podendo ser utilizados até 8 discos do tipo "cartridge" com 408 cilindros;

b) Terminais - o controlador de terminais vídeo-teclado programado suporta a utilização simultânea de até 4 terminais sem inteligência local;

c) Impressora - o dispositivo de impressão programado é uma impressora serial matricial, permitindo o uso de apenas um dos dois transportadores de papel disponíveis (o da esquerda), para simular uma impressora de linhas;

d) Fita magnética - está sendo programada, utilizando-se um canal disponível de acesso direto à memória, e será integrada ao sistema numa segunda etapa.

4.4 ALocação do Disco

O disco do sistema é subdividido, basicamente, em 4 segmentos contíguos:

- Segmento bloco zero - este segmento (1 setor) contém um programa em código de máquina que faz a carga do "kernel" e do interpretador na memória e determina a configuração da mesma; é chamado (carregado no endereço 26000_8 e executado) na inicialização, pelo programa IPL, e na reinicialização do sistema, pelo próprio "kernel" (0,25 KW);

- Segmento do "kernel" - contém o código de máquina do "kernel" e do interpretador (5,75 KW);

- Segmento do sistema operacional - contém o código virtual do sistema operaciomal em uso (32 KW);

- Segmento do sistema e do usuário - este segmento é composto por um sistema de arquivos definido para o sistema operacional, que contém arquivos de programas e dados do sistema e do usuário (restante do disco).

Os três primeiros segmentos ocupam áreas maiores do que as necessárias a fim de permitir futuras expansões ou quaisquer alterações que modifiquem os tamanhos dos programas.

A organização do último segmento é definida no sistema operacional. O sistema SOLO (modificado), por exemplo, define a seguinte organização para este segmento[1]:

- segmento para outro sistema operacional de uso temporário chamado pelo programa START (32 KW);

- lista de setores livres, contendo uma indicação, para o sistema de arquivos, de todos os setores de disco

disponíveis (0,75 KW);

- mapa de setores do catálogo, contendo o comprimento do catálogo de arquivos do disco e os endereços de seus setores no disco (0,25 KW);

- área de programas e arquivos de dados do SOLO e do usuário.

4.5 Entrada e Saída

A instrução de entrada e saída do Pascal Concorrente - IO - é modificada para permitir o acesso aos diversos canais ou unidades de um mesmo periférico, através de um só nome e indicando o número da unidade desejada. Isso não altera a forma da instrução, do ponto de vista do compilador e da primitiva IO do "kernel" (ver seção 5.3.3), que a definem com 3 operandos:

- variável de um tipo passivo arbitrário, que indica o endereço do "buffer" para transferências de dados;

- variável de um tipo passivo arbitrário, que indica o endereço de uma estrutura de dados que define a operação desejada (ver seções 5.1-a e 5.5);

- constante de um tipo de enumeração arbitrário, que indica o dispositivo (controlador) de IO com o qual deve ser realizada a operação requisitada.

A modificação da instrução reflete-se apenas na definição da estrutura de dados do segundo operando nos programas concorrentes, e na análise da operação requisitada

pela rotina que inicia sua execução sobre o periférico indicado, dentro do "kernel". Assim, a estrutura de dados, referenciada como IOPARAM, consiste de 4 elementos que indicam, nesta ordem (ver seção 5.1-a):

- a operação de IO (IOOPERATION);
- o estado - operação COMPLETADA ou ERRO resultante da execução da mesma (IORESULT);
- um argumento definido de acordo com as características específicas de cada periférico;
- a unidade do periférico, no caso de controladores que controlam mais de uma unidade, como os de disco e de terminais .

Para alguns periféricos, são definidas operações que, no sistema original de Brinch Hansen, não tem significado algum. É o caso, por exemplo, da programação da impressora serial para simular uma impressora de linhas (ver seção 5.5.3), e da programação do vídeo dos terminais (ver seção 5.5.2).

4.6 Prioridades

Durante sua execução, um processo pode assumir 3 prioridades distintas. A prioridade inicial, atribuida em sua criação, é 2 (mais baixo nível). Enquanto um processo está dentro de um monitor, sua prioridade é 0 (mais alto nível). Há ainda um nível intermediário de prioridade (1) que é atribuido aos processos que, não estando dentro de monitores, recebem um aviso de término da operação de IO requisitada anteriormente,

mudando seu estado de "bloqueado" para apto a concorrer pela UCP ("ready"). Quando a fatia de tempo dada ao processo, para sua execução, é completada ou ultrapassada, lhe é atribuida prioridade 2, a menos que o processo esteja executando dentro de monitor.

Um processo com prioridade 0, sendo executado, não perde a UCP quando se esgota sua fatia de tempo, detendo-a, portanto, até ter sua prioridade diminuida ou ser bloqueado por algum motivo inerente ao próprio processo. No caso dos outros níveis (prioridades 1 e 2), quando se esgota a fatia de tempo, o processo sendo executado perde a UCP para outro de maior (ou mesma) prioridade. O critério de seleção do "scheduler" é não deixar na fila READY qualquer processo com prioridade maior que a do processo sendo executado.

4.7 Descritores de Processos

IMPORTAN

Cada processo é representado, no "kernel", por um registro descritor (ver seção 2.2.1.1). A estrutura de um registro descritor de processo é apresentada junto às estrutura de dados do "kernel" (ver seção 5.1-a). Um descritor é definido como sendo um tipo PROCESS, que define um RECORD de 4 campos:

- LINK, que serve para ligar o registro a uma fila de múltiplos processos (lista duplamente encadeada);

- HEAD, que mantém informações relacionados com o sistema e com a execução do processo pelo mesmo (tipo HEADTYPE);

- REG, para manter uma cópia do estado dos registradores da máquina virtual quando o processo liberou a UCP pela última vez; e

- MAP, que mantém uma cópia da ATT do processo (tabela com os números das páginas de memória alocadas para o processo).

Quando um processo é selecionado para execução (estado "running"), seu descritor é retirado da fila em que estava ligado (LINK), seu HEAD é copiado para um outro registro do tipo HEADTYPE, mantido no "kernel" para o processo "running", seu REG é copiado para os registradores da máquina virtual e o MAP é carregado na ATT do computador para estabelecer o mapeamento da memória virtual do processo. Quando o processo libera a UCP, ocorrem as mesmas operações ao contrário, exceto para MAP, que não precisa ser atualizado.

4.8 Código Virtual

O código virtual gerado pelos compiladores Pascal Concorrente e Seqüencial é interpretado e executado por um interpretador escrito em linguagem Assembler do computador.

Com a implementação do interpretador, podem ser facilmente implantados os compiladores e os demais programas que compõem os sistemas operacionais já escritos com Pascal Concorrente. Inicialmente, o único sistema operacional implantado é o SOLO, cujos programas são, inclusive, utilizados para testar o interpretador e o "kernel". Para implantação do

sistema, basta, portanto, criar um disco com o sistema completo, conforme a organização já descrita neste texto (ver seção 4.4), e iniciar a execução do código objeto do "kernel" através de um procedimento de IPL, a partir daquele disco.

Os compiladores originais geram código virtual apropriado ao computador PDP-11/45, onde foram inicialmente implementados, por Brinch Hansen. Devido a arquitetura daquele computador, os códigos de instruções e a maioria de seus operandos são números pares, pois o processador endereça a "bytes", e os endereços das palavras são 0,2,4,6,etc.

Neste trabalho, os compiladores são modificados para gerarem código virtual mais compatível com a arquitetura do LABO-8034, cujo processador endereça a palavras. As modificações são feitas nos passos 6 e 7 dos compiladores e envolvem somente os códigos de operação e a geração de endereços, deslocamentos e argumentos afins. Elas visam a simplificação do interpretador com o conseqüente incremento de sua eficiência.

4.9 Ponto Flutuante

O equipamento utilizado não possui processador de ponto flutuante. Os números reais (números de ponto flutuante) são processados por "software", através de um conjunto de rotinas, escritas em Assembler, inseridas nas peças de código que executam as instruções virtuais correspondentes. As rotinas estão totalmente adaptadas ao tipo de processamento sobre pilha

da máquina virtual ("stack machine").

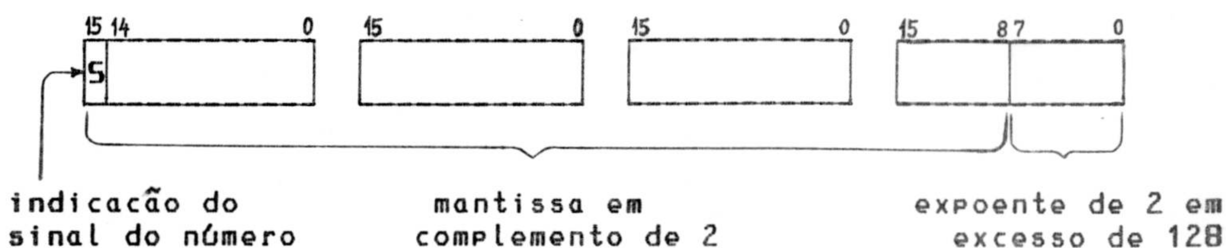
As operações implementadas são :

- aritméticas: +, - (unário e binário), *, /;

- relacionais: <, =, >, <=, <>, >=;

- conversões: inteiro --> real, real --> inteiro;

- valor absoluto.



Normalização:
bit S <> bit anterior

Representação		Expoente
0	-->	-128
128	-->	0
255	-->	127

Fig. 12: Formato e representação de números de ponto flutuante.

Um número real é representado em 4 palavras (64 bits) e seu formato é mostrado na figura 12. Assim, o intervalo dos reais é, aproximadamente, de -10^{38} a $+10^{38}$, e o menor real positivo não nulo (<>0) é 10^{-39}, aproximadamente. A precisão de um número real, cuja mantissa é representada em 3,5 palavras (56 bits), é de, aproximadamente, 16 dígitos decimais significativos. Todos os números reais são normalizados, exceto o zero, que é representado por zeros na mantissa e no expoente.

O algorítmo empregado na programação do processador de ponto flutuante não é apresentado neste trabalho. Ele é desenvolvido, segundo a conveniência, com partes de algorítmos conhecidos e técnicas de programação visando a simplificação e adequação ao sistema.

5 KERNEL

IMPORTANTE

O "kernel" aqui descrito é um conjunto de rotinas e estruturas de dados que implementam as funções básicas necessárias para a execução de programas concorrentes (sistemas operacionais escritos em Pascal Concorrente).

Inicialmente, uma versão abstrata do "kernel" é escrita numa linguagem semelhante ao Pascal Concorrente e, posteriormente, esta versão é traduzida (reescrita) para a linguagem Assembler do computador. Algumas diferenças entre a linguagem abstrata utilizada e o Pascal Concorrente são:

- uso indiscriminado do símbolo '@' para designar endereços de estruturas de dados e os conteúdos das estruturas cujos endereços são dados, seguindo a forma de "pointers" do Pascal Seqüencial;

- uso de PACKED RECORD para compactação de informações a nível de bits;

- definição de alguns tipos de dados estruturados e constantes sem sentido no contexto do Pascal (definição comentada);

- uso da cláusula 'UNIV' indiscriminadamente;

- definição de variáveis com cláusula "HARDWARE", para identificar variáveis que podem ser usadas e modificadas tanto pelo programa como pelo "hardware" (processador e controladores);

- declaração de rotinas dentro de classes, que podem ser chamadas tanto de dentro como de fora ("ENTRY") da classe a que pertencem.

A filosofia de Brinch Hansen[1] é seguida na organização do "kernel". Assim, os diversos componentes são agrupados em um conjunto de estruturas de dados que representam processos, monitores e periféricos. Cada estrutura consiste de uma parte que define a forma dos dados na memória, e outra que define as operações que podem ser realizadas sobre estes dados. Isto lembra o conceito de classe do Pascal Concorrente, e assim são referidas, estas estruturas, no decorrer deste capítulo.

As classes que compõem o "kernel" são as seguintes:

a) NEWCORE - aloca registros descritores de processos e portas de monitores na memória ("heap") reservada ao "kernel" (ver seções 2.2.1.1 e 2.2.1.2); o tamanho desta área de memória é definido em tempo de compilação (montagem) do "kernel", em função do número máximo de processos e monitores, estabelecido, para o sistema, pelo programador;

b) PROCESSQUEUE - implementa filas de múltiplos processos através de listas duplamente encadeadas (ver seção 2.2.1.1);

c) SIGNAL - implementa filas nas quais processos podem esperar por um sinal do relógio (incremento do contador de segundos), ou por algum sinal externo (tecla 'BELL', por exemplo);

d) TIME - faz atualizações de tempo real, em termos de segundos e décimos de milésimo de segundo;

e) TIMER - mede intervalos de tempo;

f) CLOCK - implementa o relógio e suas funções;

g) CORE - aloca memória para segmentos de dados dos processos;

h) VIRTUAL - determina a memória virtual para os processos e controla a utilização da mesma;

i) RUNNING - cria e executa processos, distribuindo entre eles o recurso "tempo de UCP";

j) READY - seleciona processos para execução e atualiza a fila de mesmo nome;

l) GATE - controla o acesso exclusivo a monitores;

m) periféricos - tratamento básico de entrada/saída (uma classe para cada periférico).

Cada classe é formada por uma série de rotinas que executam as operações definidas sobre uma estrutura de dados. Existem, ainda, algumas rotinas que não são agrupáveis na forma de classes. A figura 13 mostra os diversos caminhos de acesso às classes durante a execução de uma função do "kernel". Pode ser observado que existe uma estrutura hierárquica na execução das classes do "kernel", que só é quebrada pelas classes READY e RUNNING, as quais se comunicam entre si.

As rotinas do "kernel" executam um dos três tipos de funções abaixo:

- primitivas;
- tratamento de interrupções;
- funções internas.

Nas seções seguintes, são apresentadas as estruturas de dados de cada classe e uma descrição de cada uma das rotinas do "kernel", individualmente, com indicação da classe a que pertencem. As classes que implementam os periféricos são apresentadas separadamente. Os aspectos de implementação são apresentados através da versão abstrata em

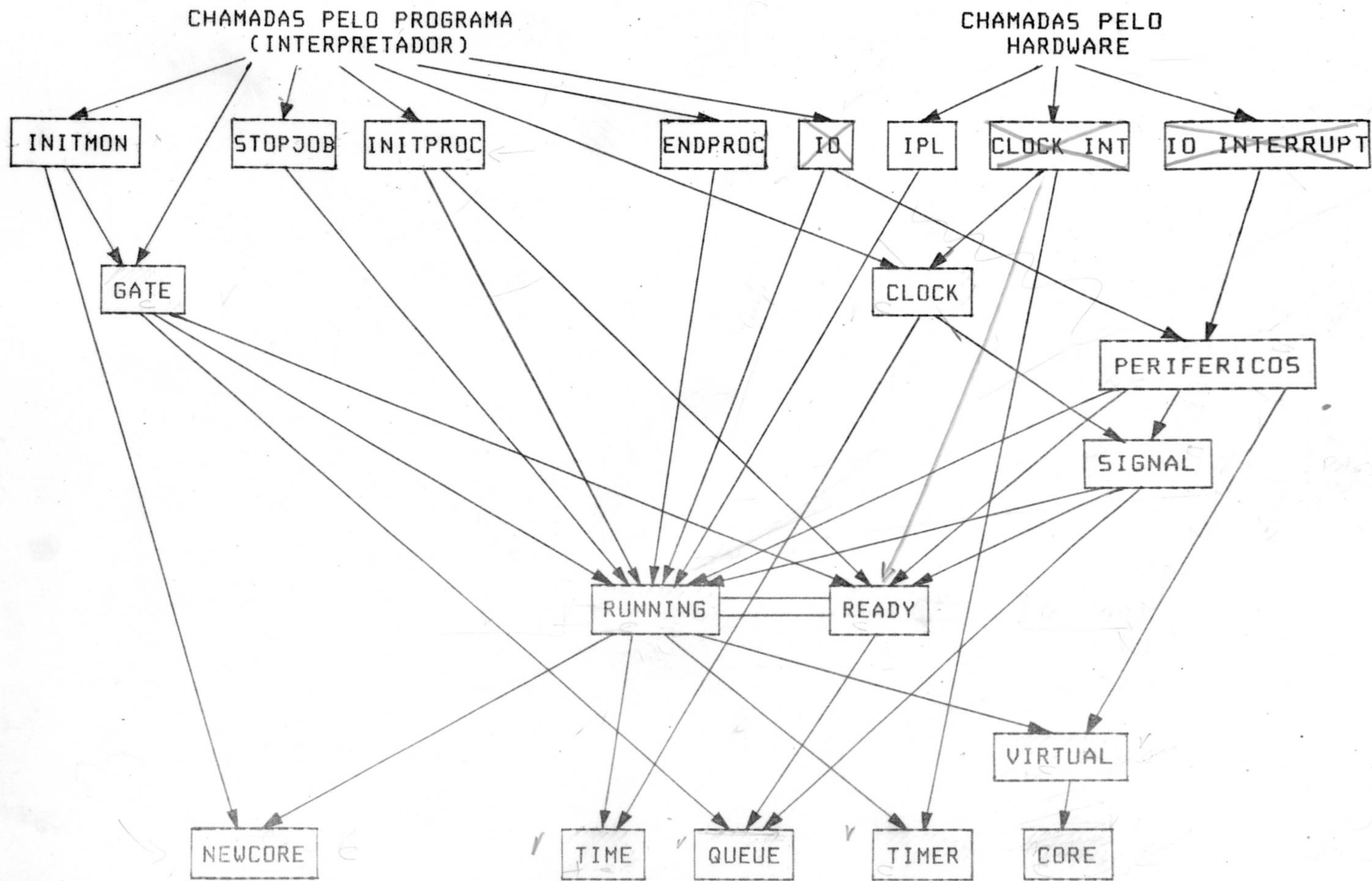


Fig. 13: Diagrama de execução das classes do "kernel".

Pascal Concorrente referida no início deste capítulo.

5.1 Estruturas de Dados

Nesta seção, são apresentadas as diversas estruturas de dados que compõem as classes do "kernel". As rotinas que operam sobre estas estruturas são descritas nas seções seguintes. Aqui, apenas são indicadas as suas declarações de forma simplificada, para dar uma visão da estrutura de cada classe. Além das estruturas de dados, são também apresentadas as rotinas que inicializam as variáveis permanentes[1] de cada classe (operação inicial da classe).

a) Tipos de dados do "kernel" - declaração dos tipos de dados utilizados para definição das estruturas de dados do "kernel":

```
CONST NIL = 0;
      MAXGATES = 25 "PORTAS DE MONITORES"; GATEREC = 3 "WORDS";
      MAXPROCESSES = 10 "PROCESSOS"; PROCESSREC = 68 "WORDS";

" TIPOS DE DADOS BASICOS "

TYPE  ADDRESS = 0..65535;
TYPE  HEADTYPE = RECORD
                   INDEX, "END. DE REF. AO PROCESSO NO KERNEL"
                   HEAPTOP,
                   LINE, "NO. DA LINHA FONTE SENDO EXECUTADA"
                   RESULT : INTEGER;
                   RUNTIME:TIME; "TEMPO DECORRIDO DE EXECUCAO"
                   SLICE, "PARTE JA USADA DA FATIA DE TEMPO"
                   NESTING, "NO. CHAMADAS EMBUT. DE MONITORES"
                   PRIORITY : INTEGER;
                   OVERTIME, "ULTRAPASSOU 1 FATIA DE TEMPO"
                   JOB, "EXECUTANDO PROGRAMA SEQUENCIAL"
                   CONTINUE : BOOLEAN; "= NOT STOP"
                   OPCODE:INTEGER; "PRIMITIVA CHAMADA E ...."
                   PARAM:ARRAY (.1..4.) OF INTEGER; "...ARGS."
                   OPLINE:INTEGER "COMANDO QUE CHAMOU PRIMIT."
                 END;
```

```
TYPE  REGTYPE   = RECORD
                    S, W, Q, X, PC,
                    B, G,
                    TA, TB, TC,
                    FE : INTEGER;
                    FM : REAL
                  END;

TYPE  MAPREG = 0..255; "1 BYTE"
      MAPTYPE = ARRAY (.0..63.) OF MAPREG;

TYPE  QUEUETYPE= RECORD
                    SUCC : @QUEUETYPE;
                    PRED : @QUEUETYPE
                  END;

TYPE  PROCESS   = RECORD
                    LINK : QUEUETYPE;
                    HEAD : HEADTYPE;
                    REG : REGTYPE;
                    MAP : MAPTYPE
                  END;
TYPE  PROCESSREF = @PROCESS;

" TIPOS DE DADOS DE IO "

TYPE STRING= @ARRAY (.1.."UNTIL STRING@(.I.)=(:0:)".) OF CHAR;
     BUFFERTYPE= @ARRAY (.1.."COMPR. INTRINSICO NO CONTEXTO".)
                          OF CHAR;
TYPE IODEVICE = (ALMO, CDISK, TAPEO, PRINTERO, CARDREADER);
     IOOPERATION = (INPUT, OUTPUT, MOVE, CONTROL);
     IORESULT= (COMPLETE, INTERVENTION, TRANSMISSION, FAILURE,
                  ENDFILE, ENDMEDIUM, STARTMEDIUM);

TYPE  IOPARAM =   RECORD
                    OPERATION : IOOPERATION;
                    STATUS : IORESULT;
                    ARG : "UNIV" INTEGER;
                    UNIT : INTEGER
                  END;
```

b) Classe NEWCORE - os dados desta classe controlam a utilização do "heap" do "kernel", indicando o início da área livre e o seu tamanho:

```
VAR  NEWCORE : CLASS;
  CONST  SPACE = MAXPROCESSES*PROCESSREC  "DESCRITORES"
                 + MAXGATES*GATEREC  "PORTAS";
         BASEADDRESS = ..."ENDER. DA ROTINA DE CARGA DO
                           SISTEMA OPERACIONAL";
         SPACELIMIT = 'SPACE LIMIT(:0:)';
  VAR  TOP, FREE : INTEGER;
```

```
FUNCTION ENTRY NEW .....;
.........................

BEGIN
  TOP := BASEADDRESS;
  FREE := SPACE;
END;
```

c) Classe PROCESSQUEUE - a estrutura de dados desta classe é uma lista duplamente encadeada (entretanto, apenas o nodo cabeça da lista é declarado internamente):

```
TYPE  PROCESSQUEUE = CLASS;
  VAR  QUEUE : QUEUETYPE;

  FUNCTION ENTRY GET .....;
  .........................

  PROCEDURE ENTRY PUT .....;
  .........................

  FUNCTION ENTRY ANY .....;
  .........................

  FUNCTION ENTRY EMPTY .....;
  ...........................

  BEGIN
    QUEUE.SUCC := @QUEUE;
    QUEUE.PRED := @QUEUE;
  END;
```

d) Classe SIGNAL - esta classe usa a classe PROCESSQUEUE para implementar uma fila de processos que esperam por algum sinal da máquina (relógio) ou do operador:

```
TYPE  SIGNAL = CLASS;
  VAR  AWAITING : PROCESSQUEUE;

  PROCEDURE ENTRY AWAIT .....;
  ...........................

  PROCEDURE ENTRY SEND .....;
  ...........................

  BEGIN
    INIT  AWAITING;
  END;
```

U F R G S
BIBLIOTECA
CPD/PGCC

e) Classe TIME - a classe TIME define um registro de tempo real com dois ítens que indicam o número de segundos e uma fração de segundo (1/10000) já transcorridos desde o início da contagem:

```
TYPE  TIME = CLASS;
  VAR ENTRY  REALTIME : RECORD
                          SEC : INTEGER;
                          FRACTION : INTEGER
                        END;

  PROCEDURE ENTRY ADD .....;
  .........................

  BEGIN
    REALTIME.SEC := 0;
    REALTIME.FRACTION := 0;
  END;
```

f) Classe TIMER - apenas uma variável permanente - PERIOD - é definida por esta classe e sua funcão é marcar pequenos intervalos de tempo:

```
VAR  TIMER : CLASS;
  CONST  SMALLINCR = 10;
         LARGEINCR = 200; "EM DECIMOS DE MILISSEGUNDO"
  VAR  PERIOD : INTEGER;

  FUNCTION ENTRY ELAPSED .....;
  .............................

  PROCEDURE ENTRY TICK .....;
  ..........................

  PROCEDURE ENTRY RESET .....;
  ............................

  BEGIN
    PERIOD := 0;
  END;
```

g) Classe CLOCK - aqui são usadas as classes TIME e SIGNAL, para implementar, respectivamente, o relógio do sistema e a fila de processos que esperam o próximo sinal de um segundo deste relógio:

```
VAR  CLOCK : CLASS;
  VAR  NOW : TIME;
       NEXTTIME : SIGNAL;

  PROCEDURE ENTRY INCREMENT .....;
  ................................

  PROCEDURE ENTRY WAIT .....;
  ...........................

  FUNCTION ENTRY REALTIME .....;
  ..............................

  BEGIN
    INIT  NOW,
          NEXTTIME;
  END;
```

h) Classe CORE - esta classe controla a alocação de memória real, através de variáveis que indicam a primeira página livre da memória e o número de páginas livres:

```
VAR  CORE : CLASS;
  CONST  CORECAPACITY = ... "ENTRE 64 E 256 PAGINAS DE 1 KB";
         CORELIMIT = 'CORE LIMIT(:0:)';
  VAR  TOP,
       FREE : INTEGER;

  PROCEDURE ENTRY ALLOC .....;
  ............................

  BEGIN
    TOP := "PAGINA" 0 ;
    FREE := CORECAPACITY "PAGINAS";
  END;
```

i) Classe VIRTUAL - a classe VIRTUAL controla os mecanismos de endereçamento da memória virtual de cada processo e define variáveis para a alocação particionada deste espaço virtual:

```
VAR  VIRTUAL : CLASS;
  CONST  VIRTUALSPACE = 64 "PAGINAS DE 1 KB";
         VIRTUALLIMIT = 'VIRTUAL LIMIT(:0:)';
  VAR  COMMON : INTEGER;
  VAR ENTRY  HEAPTOP : INTEGER;
  VAR "HARDWARE"  ATT : MAPTYPE;
```

```
PROCEDURE ENTRY GETMAP .....;
.............................

PROCEDURE PUTMAP .....;
.......................

PROCEDURE ENTRY REALADDRESS .....;
..................................

PROCEDURE ENTRY DEFPRIVATE .....;
.................................

PROCEDURE ENTRY DEFCOMMON .....;
................................

BEGIN  END;
```

j) Classe RUNNING - esta classe controla a criação e execução de processos, mantendo a identificação (referência) de todos os processos criados e, separadamente, a identificação do processo sendo executado, juntamente com uma cópia de seu registro descritor para uso do processador (quando o processo perde a UCP, seu descritor é atualizado):

```
VAR  RUNNING : CLASS;
  CONST  MAXPARAM = 20;
         PARAMLIMIT = 'PARAMETER LIMIT(:0:)';
         STARTADDR = ... "END. INICIAL DO INTERPRETADOR";
  VAR  CONSTADDR : INTEGER;
       PARAMADDR : @ARRAY (.1..MAXPARAM.) OF INTEGER;
       NEXTINDEX : INTEGER;
  VAR ENTRY PROCESSID:ARRAY (.1..MAXPROCESSES.) OF PROCESSREF;
      ENTRY  USER : PROCESSREF;
      ENTRY  HEAD : HEADTYPE;
  VAR "HARDWARE"  REG : REGTYPE;

  PROCEDURE ENTRY POPPARAM .....;
  ...............................

  PROCEDURE ENTRY INITPARENT .....;
  .................................

  PROCEDURE ENTRY INITCHILD .....;
  ................................

  PROCEDURE ENTRY SERVE .....;
  ............................
```

```
FUNCTION ENTRY PREEMPTED .....;
...............................

PROCEDURE "ENTRY" UPDATE .....;
...............................

PROCEDURE ENTRY ENTER .....;
............................

PROCEDURE ENTRY LEAVE .....;
............................

PROCEDURE ENTRY STARTIO .....;
..............................

PROCEDURE ENTRY SYSTEMERROR .....;
..................................

BEGIN
  NEXTINDEX := 1;
END;
```

l) Classe READY - três filas de processos são definidas nesta classe juntamente com uma variável booleana, para implementar a fila de processos "ready" (com seus três níveis de prioridade) e para indicar o estado da UCP (ocupada ou desocupada):

```
VAR  READY : CLASS;
  VAR  TOP "PRIORIDADE 0",
       MIDDLE "PRIORIDADE 1",
       BOTTOM "PRIORIDADE 2" : PROCESSQUEUE;
       IDLING : BOOLEAN;

  PROCEDURE "ENTRY" ENTER .....;
  ..............................

  PROCEDURE "ENTRY" RESCHEDULE .....;
  ...................................

  PROCEDURE ENTRY SELECT .....;
  .............................

  PROCEDURE ENTRY ENDIO .....;
  ............................

  BEGIN
    INIT  TOP, MIDDLE, BOTTOM ;
    IDLING := FALSE;
  END;
```

m) Classe GATE - esta classe define, para cada monitor, uma variável booleana e uma fila de processos, que implementam, respectivamente, o estado do monitor (livre ou ocupado) e a fila de espera para entrada no mesmo:

```
TYPE  GATESTRUCT = RECORD
                     OPEN : BOOLEAN;
                     WAITING : PROCESSQUEUE
                   END;

TYPE  GATE = CLASS(G : @GATESTRUCT);

  PROCEDURE ENTRY ENTER .....;
  ............................

  PROCEDURE "ENTRY" LEAVE .....;
  ..............................

  PROCEDURE ENTRY DELAY .....;
  ............................

  PROCEDURE ENTRY CONTINUE .....;
  ...............................

  BEGIN
    RUNNING.ENTER;
    WITH  G@
    DO  BEGIN
        OPEN := FALSE;
        INIT  WAITING;
        END;
  END;
```

5.2 Funções Internas

As funções internas do "kernel" são aquelas rotinas que só podem ser chamadas por outras rotinas do "kernel". Cada uma delas é apresentada a seguir através de:

- breve descrição da função;
- indicação dos parâmetros;
- rotina que implementa a função.

A maior parte das funções são rotinas (de entrada ou locais) de alguma classe. Assim, seus nomes são compostos do nome da classe e do nome da rotina, separados por um ponto e nesta ordem. Nos outros casos, o nome da função é o nome da rotina que a implementa.

1) NEWCORE.NEW - a função retorna com o endereço inicial da área livre - no "heap" do "kernel" - e incrementa este endereço da quantidade de memória solicitada. O tamanho da área livre é decrementado da mesma quantidade.

Parâmetro: LENGTH - número de palavras requeridas.

Rotina:

```
FUNCTION ENTRY NEW(LENGTH : INTEGER) : UNIV INTEGER;
BEGIN
  IF  LENGTH > FREE
  THEN KERNELERROR(SPACELIMIT);
  FREE := FREE - LENGTH;
  NEW := TOP;
  TOP := TOP + LENGTH;
END;
```

2) PROCESSQUEUE.GET - o primeiro processo da fila é retirado, e o endereço de seu descritor é retornado como valor desta função.

Parâmetros: nenhum.

Rotina:

```
FUNCTION ENTRY GET : @QUEUETYPE;
VAR  FIRST, SECOND : @QUEUETYPE;
BEGIN
  FIRST := QUEUE.SUCC;
  SECOND := FIRST@.SUCC;
  QUEUE.SUCC := SECOND;
  SECOND@.PRED := FIRST@.PRED;
  GET := FIRST;
END;
```

3) PROCESSQUEUE.PUT - insere um processo no fim da fila.

Parâmetro:

NEWELEM - endereço do descritor do processo.

Rotina:

```
PROCEDURE ENTRY PUT(NEWELEM : @QUEUETYPE);
VAR  LAST : @QUEUETYPE;
BEGIN
  LAST := QUEUE.PRED;
  QUEUE.PRED := NEWELEM;
  NEWELEM@.PRED := LAST;
  NEWELEM@.SUCC := LAST@.SUCC;
  LAST@.SUCC := NEWELEM;
END;
```

4) PROCESSQUEUE.ANY - indica se há algum processo na fila.

Parâmetros: nenhum.

Rotina:

```
FUNCTION ENTRY ANY : BOOLEAN;
BEGIN
  ANY := QUEUE.SUCC <> @QUEUE;
END;
```

5) PROCESSQUEUE.EMPTY - indica se a fila está vazia.

Parâmetros: nenhum.

Rotina:

```
FUNCTION ENTRY EMPTY : BOOLEAN;
BEGIN
  EMPTY := QUEUE.SUCC = @QUEUE;
END;
```

6) SIGNAL.AWAIT - insere o processo "running" na fila de espera pela ocorrência do sinal definido (relógio ou operador).

Parâmetros: nenhum.

Rotina:

```
PROCEDURE ENTRY AWAIT;
BEGIN
  AWAITING.PUT(RUNNING.PREEMPTED);
END;
```

7) SIGNAL.SEND - esta rotina é chamada quando ocorre o sinal definido. Se a fila de espera não está vazia, todos os processos que nela se encontrem são retirados, um a um, e colocados na fila "ready" e, a seguir, o "scheduler" é chamado para decidir qual o próximo processo a ser executado. Se a fila está vazia, o sinal é ignorado.

Parâmetros: nenhum

Rotina:

```
PROCEDURE ENTRY SEND;
BEGIN
  IF AWAITING.ANY
  THEN BEGIN
       REPEAT READY.ENTER(AWAITING.GET)
       UNTIL  AWAITING.EMPTY;
       READY.RESCHEDULE;
       END;
END;
```

8) TIME.ADD - adiciona um intervalo de tempo ao registro de tempo real e indica se houve incremento do contador de segundos.

Parâmetros:

INCR - intervalo a ser adicionado, em décimos de milissegundo.

TURN - aviso de incremento ou não do contador de segundos.

Rotina:

```
PROCEDURE ENTRY ADD(INCR : INTEGER; VAR TURN : BOOLEAN);
BEGIN
  WITH REALTIME
  DO BEGIN
     FRACTION := FRACTION + INCR;
     TURN := FRACTION >= 10000;
     IF TURN
     THEN BEGIN
          SEC := SEC + 1;
          FRACTION := FRACTION - 10000;
          END;
     END;
END;
```

9) TIMER.ELAPSED - esta função retorna com o tempo decorrido desde que o processo "running" está sendo executado pela UCP, e zera o contador deste período.

Parâmetros: nenhum.

Rotina:

```
FUNCTION ENTRY ELAPSED : INTEGER;
BEGIN
  ELAPSED := PERIOD + SMALLINCR;
  PERIOD := 0;
END;
```

10) TIMER.TICK - atualiza o período, adicionando o intervalo de tempo correspondente à freqüência de interrupções do relógio. Este intervalo é enviado ao procedimento de chamada para outras atualizações necessárias.

Parâmetro: INTERVAL - intervalo de tempo.

Rotina:

```
PROCEDURE ENTRY TICK(VAR INTERVAL : INTEGER);
BEGIN
  INTERVAL := LARGEINCR;
  PERIOD := PERIOD + LARGEINCR;
END;
```

11) TIMER.RESET - inicializa a contagem do período (período igual a zero).

Parâmetros: nenhum.

Rotina:

```
PROCEDURE ENTRY RESET;
BEGIN
  PERIOD := 0;
END;
```

12) CLOCK.INCREMENT - atualiza o tempo real do relógio do sistema e, se houve incremento do contador de segundos, libera os processos que estejam a espera, em NEXTTIME, para concorrerem pela UCP, e troca o estado do sinalizador de tempo (lâmpada 4) no painel do computador.

Parâmetro: INTERVAL - intervalo de tempo.

Rotina:

```
PROCEDURE ENTRY INCREMENT (INTERVAL : INTEGER);
CONST ON = TRUE; OFF = FALSE;
VAR SECINCR : BOOLEAN;
VAR "HARDWARE" LAMP4 : BOOLEAN;
BEGIN
  NOW.ADD(INTERVAL,SECINCR);
  IF SECINCR
  THEN BEGIN
       IF  NOW.REALTIME.SEC MOD 2 = 0
       THEN LAMP4 := OFF
       ELSE LAMP4 := ON;
       NEXTTIME.SEND;
       END;
END;
```

13) CORE.ALLOC - aloca páginas contíguas da memória real, de acordo com o número de palavras solicitadas. O número (endereço) da primeira página e o número de páginas alocadas são enviados ao procedimento que chamou esta rotina, através dos parâmetros. As variáveis de controle - TOP e FREE - são atualizadas.

Parâmetros:

LENGTH - número de palavras de memória solicitadas.

FIRST - número da primeira página alocada.

PAGES - número de páginas alocadas.

Rotina:

```
PROCEDURE ENTRY ALLOC(LENGTH : INTEGER;
                      VAR FIRST: INTEGER; VAR PAGES: INTEGER);
BEGIN
  PAGES := (LENGTH - 1) DIV 512 + 1;
  IF PAGES > FREE
  THEN KERNELERROR (CORELIMIT);
  FREE := FREE - PAGES;
  FIRST := TOP;
  TOP := TOP + PAGES;
END;
```

14) VIRTUAL.GETMAP - copia o mapa de páginas de memória alocadas ao processo indicado, do seu registro descritor para os registradores de "hardware".

Parâmetros: P - identificação do processo.

Rotina:

```
PROCEDURE ENTRY GETMAP (P : PROCESSREF);
BEGIN
  ATT := P@.MAP;
END;
```

15) VIRTUAL.PUTMAP - copia os registradores de "hardware", que contém o mapa de páginas de memória do processo "running", no registro descritor do processo indicado. Esta sub-rotina é de uso local para a classe VIRTUAL.

Parâmetros: P - identificação do processo.

Rotina:

```
PROCEDURE PUTMAP (P : PROCESSREF);
BEGIN
  P@.MAP := ATT;
END;
```

16) VIRTUAL.REALADDRESS - esta rotina recebe o endereço de uma palavra da memória virtual e devolve o correspondente endereço de memória real, em bytes.

Parâmetros:

VIRTUALADDR - endereço virtual.

PREFIX - bits mais significativos do endereço real.

REST - 16 bits menos significativos do endereço real (determinado em bytes).

Rotina:

```
PROCEDURE ENTRY REALADDRESS (VIRTUALDDR : ADDRESS;
                             VAR PREFIX : 0..3;
                             VAR REST : ADDRESS);
VAR BITS8A17 "DO END.REAL" : ADDRESS;
BEGIN
  REST := (VIRTUALADDR*2) MOD 256;
  BITS8A17 := ATT(.VIRTUALADDR DIV 512.)  "INP 5";
  REST := REST + (BITS8A17 MOD 256) * 256;
  PREFIX := BITS8A17 DIV 256;
END;
```

17) VIRTUAL.DEFPRIVATE - define a memória virtual do processo que está sendo criado (inicializado). Uma área privada de dados é alocada e anexada a área comum, para formar a memória virtual do processo.

Parâmetro:

LENGTH - tamanho da área solicitada, em palavras.

Rotina:

```
PROCEDURE ENTRY DEFPRIVATE (LENGTH : 0..131071);
VAR BASE,PAGES,TOTAL,PAGE:INTEGER;
BEGIN
  IF LENGTH > 32768
  THEN KERNELERROR (VIRTUALLIMIT);
  CORE.ALLOC (LENGTH,BASE,PAGES);
  PUTMAP(RUNNING.USER);
  TOTAL := COMMON + PAGES;
  IF TOTAL > VIRTUALSPACE
  THEN KERNELERROR (VIRTUALLIMIT);
```

```
  WITH RUNNING.USER@
  DO FOR PAGE := COMMON TO TOTAL-1
     DO BEGIN
        MAP (.PAGE.) := BASE;
        BASE := BASE + 1;
        END;
END;
```

18) VIRTUAL.DEFCOMMON - de maneira semelhante à rotina anterior, define a memória virtual do processo inicial, quando este é inicializado. A memória assim definida é também a área comum a todos os demais processos.

Parâmetro:

LENGTH - tamanho da área comum a todos os processos ("kernel" + interpretador + código do programa + área comum de dados), em palavras.

Rotina:

```
PROCEDURE ENTRY DEFCOMMON (LENGTH : 0..131071);
BEGIN
  IF LENGTH > 32768
  THEN KERNELERROR (VIRTUALLIMIT);
  CORE.ALLOC (LENGTH,BASE,PAGES);
  COMMON := PAGES;
  HEAPTOP := COMMON * 512;
  PUTMAP (RUNNING.USER);"MAPA(ATT) PRONTO NA
                          INICIALIZACAO DO KERNEL"
END;
```

19) RUNNING.POPPARAM - esta rotina simula a passagem de parâmetros (para um outro processo a ser criado) da memória do processo inicial para variáveis da classe RUNNING. Isto é feito, mantendo na classe somente o endereço da área de parâmetros, sendo estes passados diretamente ao processo a que se destinam, posteriormente.

Parâmetro:

PARAMLENGTH - tamanho da área de parâmetros, em palavras.

Rotina:

```
PROCEDURE ENTRY POPPARAM (PARAMLENGTH:INTEGER);
BEGIN
  IF PARAMLENGTH > MAXPARAM
  THEN KERNELERROR (PARAMLIMIT);
  WITH USER@.REG
  DO BEGIN
     PARAMADDR := S;
     S := S + PARAMLENGTH;
     END;
END;
```

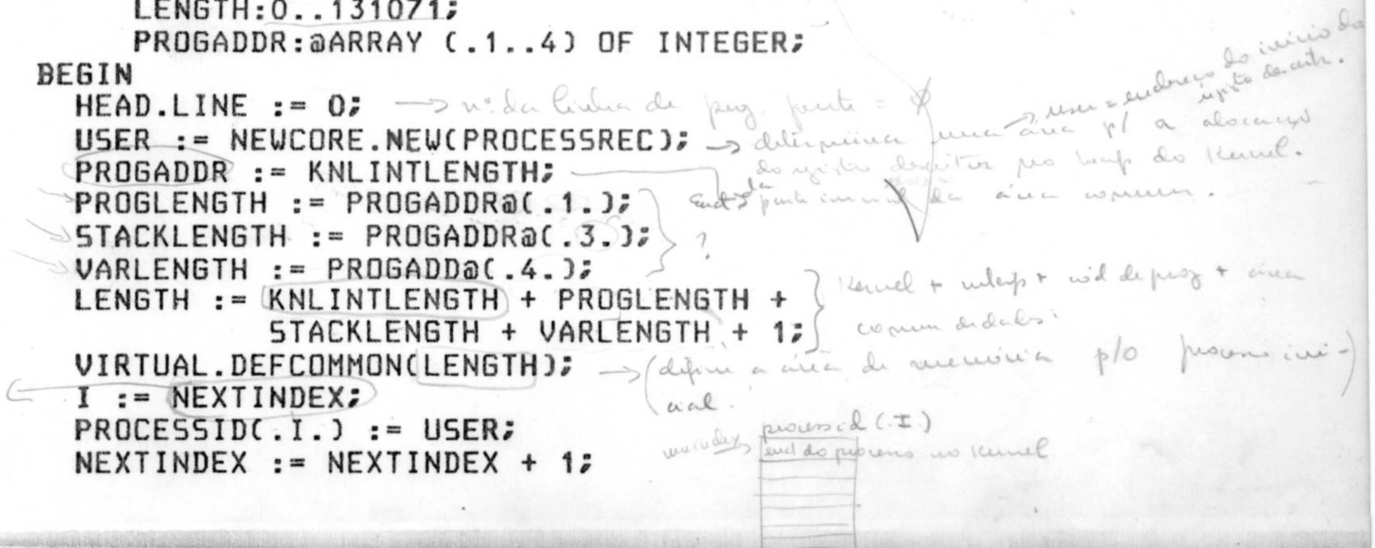
20) RUNNING.INITPARENT - esta rotina é chamada durante a inicialização do "kernel" para iniciar a execução do processo inicial ("parent"). E solicitado espaço, no "heap", para alocação de um registro descritor do processo, e definido o comprimento da área comum a todos os processos. Os registradores do "hardware" são inicializados para a execução do processo.

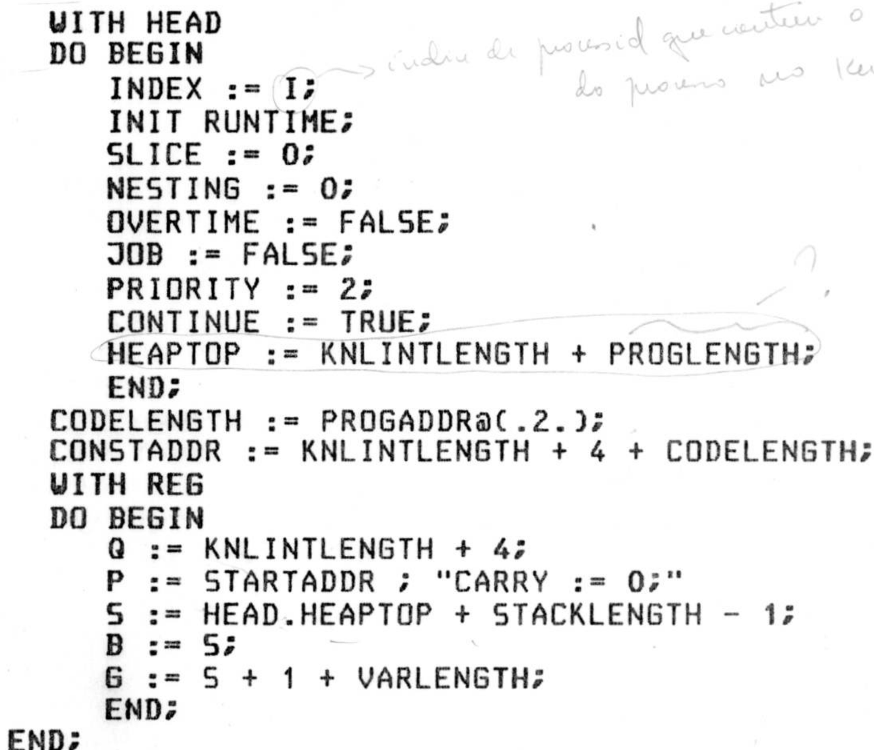
Parâmetro:

KNLINTLENGTH - tamanho do "kernel" mais tamanho do interpretador, em palavras (parte inicial da área comum).

Rotina:

```
PROCEDURE ENTRY INITPARENT (KNLINTLENGTH:INTEGER);
VAR  PROGLENGTH, VARLENGTH,
     CODELENGTH,STACKLENGTH:INTEGER;
     I : INTEGER;
     LENGTH:0..131071;
     PROGADDR:@ARRAY (.1..4) OF INTEGER;
BEGIN
  HEAD.LINE := 0;
  USER := NEWCORE.NEW(PROCESSREC);
  PROGADDR := KNLINTLENGTH;
  PROGLENGTH := PROGADDR@(.1.);
  STACKLENGTH := PROGADDR@(.3.);
  VARLENGTH := PROGADD@(.4.);
  LENGTH := KNLINTLENGTH + PROGLENGTH +
            STACKLENGTH + VARLENGTH + 1;
  VIRTUAL.DEFCOMMON(LENGTH);
  I := NEXTINDEX;
  PROCESSID(.I.) := USER;
  NEXTINDEX := NEXTINDEX + 1;
```





```
  WITH HEAD
  DO BEGIN
     INDEX := I;
     INIT RUNTIME;
     SLICE := 0;
     NESTING := 0;
     OVERTIME := FALSE;
     JOB := FALSE;
     PRIORITY := 2;
     CONTINUE := TRUE;
     HEAPTOP := KNLINTLENGTH + PROGLENGTH;
     END;
  CODELENGTH := PROGADDR@(.2.);
  CONSTADDR := KNLINTLENGTH + 4 + CODELENGTH;
  WITH REG
  DO BEGIN
     Q := KNLINTLENGTH + 4;
     P := STARTADDR ; "CARRY := 0;"
     S := HEAD.HEAPTOP + STACKLENGTH - 1;
     B := S;
     G := S + 1 + VARLENGTH;
     END;
END;
```

21) RUNNING.INITCHILD - esta rotina é semelhante a anterior; sua função é criar (inicializar) um processo indicado pelo processo que chamou o "kernel". Os parâmetros passados por este último são transportados para a área privada de dados do processo sendo inicializado.

Parâmetros:

PARAMLENGTH - tamanho da área de parâmetros passada ao processo.

VARLENGTH - tamanho da área de variáveis permanentes do processo.

STACKLENGTH - tamanho máximo da "stack" necessária para o processo.

QVALUE - endereço inicial do código virtual do processo.

Rotina:

```
PROCEDURE ENTRY INITCHILD (PARAMLENGTH,VARLENGTH,
                           STACKLENGTH,QVALUE:INTEGER);
VAR  I:INTEGER; LENGTH: 0..131071;
     PARAMPOINTER : @INTEGER;
BEGIN
  USER := NEWCORE.NEW(PROCESSREC);
  LENGTH := PARAMLENGTH + VARLENGTH + STACKLENGTH + 1;
  VIRTUAL.DEFPRIVATE(LENGTH);
  I := NEXTINDEX;
  PROCESSID(.I.) := USER;
  NEXTINDEX := NEXTINDEX + 1;
  WITH HEAD
  DO BEGIN
     INDEX := I;
     INIT RUNTIME;
     SLICE := 0;
     NESTING := 0;
     OVERTIME := FALSE;
     JOB := FALSE;
     PRIORITY := 2;
     CONTINUE := TRUE;
     HEAPTOP := VIRTUAL.HEAPTOP;
     END;
  WITH REG
  DO BEGIN
     Q := QVALUE;
     S := HEAD.HEAPTOP + STACKLENGTH-1;
     B := S;
     G := S + 1 + VARLENGTH
     P := STARTADDR ; "CARRY := 0;"
     PARAMPOINTER := G;
     FOR I := 1 TO PARAMLENGTH
     DO BEGIN
        PARAMPOINTER := PARAMPOINTER + 1;
        PARAMPOINTER@ := PARAMADDR@(.I.);
        END;
     END;
  READY.RESCHEDULE;
END;
```

22) RUNNING.SERVE - inicia a execução do processo selecionado, copiando os valores dos registradores, de seu descritor para o "hardware", e trocando seu estado para "running". A classe TIMER é chamada para inicializar o período - tempo de uso da UCP por um processo.

Parâmetro:

P - identificação do processo selecionado.

Rotina:

```
PROCEDURE ENTRY SERVE(P:PROCESSREF);
BEGIN
  TIMER.RESET;
  USER := P;
  HEAD := USER@.HEAD;
  IF HEAD.NESTING = 0
  THEN HEAD.OVERTIME := FALSE;
  REG := USER@.REG;
  VIRTUAL.GETMAP(USER);
END;
```

23) RUNNING.PREEMPTED - a função retorna com a identificação do processo correntemente no estado "running", parando a sua execução e atualizando seu registro descritor. Momentaneamente, nenhum processo fica no estado "running".

Parâmetros: nenhum.

Rotina:

```
FUNCTION ENTRY PREEMPTED:PROCESSREF;
BEGIN
  UPDATE;
  PREEMPTED := USER;
  USER@.HEAD := HEAD;
  USER@.REG := REG;
  USER := NIL;
END;
```

24) RUNNING.UPDATE - atualiza os indicadores de tempo (SLICE, RUNTIME, OVERTIME) do processo que está em execução pela UCP ("running"). Além disso, a prioridade pode ser diminuida, se o processo está fora de monitores. Esta operação pode ser requerida tanto por outra rotina desta classe (PREEMPTED), como por uma chamada externa.

Parâmetros: nenhum

Rotina:

```
PROCEDURE "ENTRY" UPDATE;
CONST MAXSLICE = 200; "20 MILISSEGUNDOS"
VAR  DUMMY:BOOLEAN;
BEGIN
  WITH HEAD
```

```
  DO BEGIN
     SLICE := SLICE + TIMER.ELAPSED;
     IF SLICE >= MAXSLICE
     THEN BEGIN
          SLICE := SLICE-MAXSLICE;
          RUNTIME.ADD(MAXSLICE,DUMMY);
          OVERTIME := TRUE;
          IF NESTING = 0
          THEN PRIORITY := 2;
          END;
      END;
END;
```

25) RUNNING.ENTER - concede maior prioridade ao processo que entra num monitor e controla, juntamente com a rotina LEAVE da mesma classe, chamadas embutidas de monitores.

Parâmetros: nenhum.

Rotina:

```
PROCEDURE ENTRY ENTER;
BEGIN
  WITH HEAD
  DO BEGIN
     NESTING := NESTING + 1;
     PRIORITY := 0;
     END;
END;
```

26) RUNNING.LEAVE - controla embutimento de chamadas de monitores e prioridade do processo que está deixando um monitor.

Parâmetros: nenhum.

Rotina:

```
PROCEDURE ENTRY LEAVE;
BEGIN
  WITH HEAD
  DO BEGIN
     NESTING := NESTING - 1;
     IF NESTING = 0
     THEN BEGIN
          PRIORITY := 2;
          READY.RESCHEDULE;
          END;
     END;
END;
```

27) RUNNING.STARTIO - esta rotina é chamada quando uma operação de IO é executada. O processo recebe prioridade de nível médio, a menos que esteja dentro de um monitor (prioridade alta).

Parâmetros: nenhum.

Rotina:

```
PROCEDURE ENTRY STARTIO;
BEGIN
  IF HEAD.NESTING = 0
  THEN HEAD.PRIORITY := 1;
END;
```

28) READY.ENTER - insere um processo na fila "ready" de acordo com a sua prioridade. Esta operação pode ser requerida tanto dentro como fora da classe READY.

Parâmetro: P - identificação do processo.

Rotina:

```
PROCEDURE "ENTRY" ENTER (P:PROCESSREF);
BEGIN
  CASE P@.HEAD.PRIORITY OF
    0: TOP.PUT(P);
    1: MIDDLE.PUT(P);
    2: BOTTOM.PUT(P);
    END;
END;
```

29) READY.RESCHEDULE - quando a UCP está ocupada, esta rotina decide se o processo que está sendo executado, deve continuar no estado "running" ou deve ser desativado ("preempted") em favor de outros. Esta rotina pode ser chamada tanto de dentro como de fora da classe READY.

Parâmetros: nenhum.

Rotina:

```
PROCEDURE "ENTRY" RESCHEDULE;
BEGIN
  WITH RUNNING
  DO IF USER <> NIL
     THEN WITH HEAD
          DO BEGIN
             UPDATE;
             IF PRIORITY > 0 AND (TOP.ANY OR OVERTIME) OR
                PRIORITY = 2 AND MIDDLE.ANY
             THEN "READY"ENTER(PREEMPTED);
             END;
END;
```

30) READY.SELECT - seleciona, para ser executado (passar ao estado "running"), o primeiro processo da fila "ready" com prioridade maior ou igual a prioridade de todos os outros processo da fila. Se a fila está vazia, o estado da UCP passa para "desocupada" ("idle") até que ocorra uma interrupção (isto é executado pela instrução fictícia WAITCPU).

Parâmetros: nenhum.

Rotina:

```
PROCEDURE ENTRY SELECT;
VAR  Q:@QUEUETYPE;
BEGIN
   IF NOT IDLING
   THEN BEGIN
        REPEAT
        IF TOP.ANY
        THEN Q := TOP.GET
        ELSE IF MIDDLE.ANY
             THEN Q := MIDDLE.GET
             ELSE IF BOTTOM.ANY
                  THEN Q := BOTTOM.GET
                  ELSE BEGIN
                       IDLING := TRUE;
                       WAITCPU;
                       IDLING := FALSE;
                       Q := NIL;
                       END
                  UNTIL Q <> NIL
                  RUNNING.SERVE(Q);
        END;
END;
```

31) READY.ENDIO - esta rotina é sempre chamada ao término de qualquer operação de IO requisitada, para trocar o

U F R G S
BIBLIOTECA
CPD/PGCC

processo de posse da UCP ou recarregar o mapa de páginas de memória do processo que estava sendo executado quando ocorreu a interrupção.

Parâmetros: nenhum.

Rotina:

```
PROCEDURE ENTRY ENDIO;
BEGIN
  RESCHEDULE;
  IF RUNNING.USER <> NIL
  THEN VIRTUAL.GETMAP(RUNNING.USER);
END;
```

32) IOFAIL - responde a uma operação de IO requisitada para um periférico inexistente ou não conectado (ausente na configuração utilizada).

Parâmetro:

PARAMADDR - endereço do RECORD de parâmetros de IO.

Rotina:

```
PROCEDURE IOFAIL (VAR PARAMADDR:@IOPARAM);
BEGIN
  PARAMADDR@.STATUS := FAILURE;
END;
```

33) DEVICEPRESENT - indica se um periférico está presente ou não na configuração do computador.

Parâmetro:

DEVCONTROL - endereço do registro de comandos do periférico.

Rotina:

```
FUNCTION DEVICEPRESENT(DEVCONTROL:UNIV @PACKED RECORD
             CLEARCONTROL:BOOLEAN;NOTUSED:ARRAY(.1..7.)
             OF BOOLEAN END):BOOLEAN;
VAR  "HARDWARE" TIME-OUT:BOOLEAN;
BEGIN
  DEVCONTROL@.CLEARCONTROL := TRUE "OUT DEV+2";
```

```
  IF  TIME←OUT                          "INP 2"
  THEN DEVICEPRESENT := FALSE
  ELSE DEVICEPRESENT := TRUE;
END;
```

34) WRITETEXT - mostra (imprime) no terminal mestre (console) uma mensagem do "kernel".

Parâmetro: TEXT - endereço da mensagem.

Rotina:

```
PROCEDURE WRITETEXT (TEXT:STRING);
CONST NULLCHAR = '(:0:)';
VAR  I : INTEGER;
BEGIN
  I := 1;
  WHILE TEXT@(.I.) <> NULLCHAR
  DO BEGIN
     ALMDTERMINAL.KERNELWRITE(TEXT@(.I.));
     I := I + 1;
     END;
END;
```

35) WRITEINT - converte um número inteiro em ASCII (decimal) na forma exigida para ser impresso no terminal pela rotina anterior.

Parâmetro: NUMBER - número a ser impresso.

Rotina:

```
PROCEDURE WRITEINT (NUMBER:INTEGER);
CONST NULLCHAR = '(:0:)'; BLANK = ' ';
VAR  N,I:INTEGER;
INT:ARRAY(.1..7) OF CHAR;
BEGIN
  INT (.6.) := BLANK; INT(.7.) := NULLCHAR;
  N := NUMBER; I := 6;
  REPEAT
    I := I -1;
    INT(.I.) := CHR(N MOD 10 + ORD('0'));
    N := N DIV 10;
  UNTIL N = 0;
  WRITETEXT (@INT(.I.));
END;
```

36) KERNELERROR - comanda a impressão, através das duas rotinas anteriores, de uma mensagem completa de erro e

termina o processamento (HALT) ou desvia para o depurador, dependendo da opção na compilação.

Parâmetro: RESULT - endereço da mensagem de erro.

Rotina:

```
PROCEDURE KERNELERROR (RESULT:STRING);
CONST SYSTEM = '(:13:)SYSTEM LINE (:0:)';
NLBEL = '(:13:)(:7:)(:0:)';
BEGIN
  WRITETEXT(@SYSTEM);
  WRITEINT(RUNNING.HEAD.OPLINE);
  WRITETEXT(RESULT);
  WRITETEXT(@NLBEL);
  CYCLE  END; "HALT"
END;
```

37) KERNELREADY - manda imprimir, via WRITETEXT, uma mensagem indicando que o sistema está pronto.

Parâmetros: nenhum.

Rotina:

```
PROCEDURE KERNELREADY;
CONST READYMESSAGE = '(:13:)SYSTEM READY(:13:)(:7:)(:0:)';
BEGIN
  WRITETEXT(@READYMESSAGE);
END;
```

5.3 Primitivas

As funções do "kernel" que podem ser chamadas diretamente pelo programa concorrente (via interpretador) são denominadas, genericamente, de primitivas. Cada primitiva executa uma função básica do sistema, de forma indivisível.

Para executar uma primitiva, um programa deve usar uma instrução virtual que faça com que o interpretador execute a pseudo-instrução "kerneltrap" (KTRAP). Esta instrução simula

a ocorrência de uma interrupção ("trap") ocasionada por "software", e assim, dá ao programa, acesso ao "kernel" para entrada em uma de suas primitivas. A instrução KTRAP possui 5 argumentos que são interpretados pelo "kernel" (passados pelo descritor do processo):

- HEAD.OPCODE - código de operação da primitiva desejada;
- HEAD.PARAM - até 4 parâmetros para a execução da primitiva.

Existem 4 grupos de primitivas, segundo as funções que realizam:

- primitivas para controle de processos;
- primitivas para implementação de monitores;
- primitiva para entrada e saída;
- primitivas auxiliares.

As primitivas de cada grupo, implementadas no "kernel", são descritas a seguir com as seguintes especificações:

- função realizada pela primitiva;
- nome interno da primitiva, composto do nome da classe a que pertence e do nome da rotina na classe (quando não pertence a nenhuma classe, só o nome da rotina é indicado);
- parâmetros da primitiva (argumentos da instrução KTRAP);
- funções internas chamadas;
- rotina que implementa a primitiva.

A seguinte rotina mostra como ocorre uma entrada no "kernel", para executar uma primitiva, e saída do mesmo, após

ser concluida a operação solicitada.

```
PROCEDURE KERNELCALL;
BEGIN
  WHIT RUNNING.HEAD
  DO CASE OPCODE OF
       0: CLOCK.WAIT;
       1: PARAM(.1.) := CLOCK.REALTIME;
       2: RUNNING.SYSTEMERROR;
       3: INITPROCESS(PARAM(.1.),PARAM(.2.),
                      PARAM(.3.),PARAM(.4.));
       4: ENDPROCESS;
       5: STOPJOB(PARAM(.1.),PARAM(.2.));
       6: GATE.ENTER(PARAM(.1.));
       7: GATE.LEAVE(PARAM(.1.));
       8: GATE.DELAY(PARAM(.1.).PARAM(.2.));
       9: GATE.CONTINUE(PARAM(.1.),PARAM(.2.));
      10: INITGATE(PARAM(.1.));
      11: IO(PARAM(.1.),PARAM(.2),PARAM(.3.));
      END;
IF RUNNING.USER = NIL
THEN READY.SELECT;
"KERNELEXIT" END;
```

5.3.1 Primitivas para Controle de Processos

Existem cinco primitivas que controlam a execução dos processos de um programa concorrente:

- INITPROCESS;
- ENDPROCESS;
- STOPJOB;
- WAIT e
- SYSTEMERROR.

Suas funções são iniciar a execução de um processo (disputa pela UCP com outros processos) e parar sua execução, definitiva ou temporariamente.

a) INITPROCESS (inicializa processo) - esta primitiva tem por função, criar, para o "kernel", um novo processo, o que, para o sistema operacional, significa inicializar um processo do programa concorrente e iniciar a sua execução. Na inicialização de um processo, o "kernel" aloca e inicializa um registro descritor para o mesmo, coloca sua identificação na lista de processos já inicializados, aloca um segmento privado de dados para o processo e passa para o mesmo ("processo filho") os parâmetros indicados pelo processo que chamou a primitiva ("processo pai"). Finalmente, o "kernel" decide se inicia a execução do processo pela UCP, ou se o põe na fila "ready" por existirem processos com maior prioridade esperando. Todo processo é inicializado com prioridade 2 (mínima). Para a realização destas operações, a primitiva chama a função interna INITCHILD.

Nome interno: INITPROCESS.

Parâmetros da primitiva:

PARAMLENGTH - tamanho da área de parâmetros passados pelo "processo pai".

VARLENGTH - tamanho da área de variáveis permanentes requerida pelo processo.

STACKLENGTH - tamanho máximo da pilha requerida pelo processo.

QVALUE - endereço inicial do código virtual do processo.

Funções internas chamadas: RUNNING.POPPARAM, RUNNING.PREEMPTED, READY.ENTER, RUNNING.INITCHILD.

Rotina:

```
PROCEDURE INITPROCESS(PARAMLENGTH,VARLENGTH,
                     STACKLENGTH,QVALUE:INTEGER);
BEGIN
  WITH RUNNING
  DO BEGIN
     IF PARAMLENGTH <> 0
     THEN POPPARAM(PARAMLENGTH);
     READY.ENTER(PREEMPTED);
     INITCHILD(PARAMLENGTH,VARLENGTH,STACKLENGTH,QVALUE);
     END;
END "KERNEL EXIT";
```

b) ENDPROCESS (termina a execução do processo) - o processo que chamou a primitiva perde a posse da UCP e deixa de concorrer pela mesma, não sendo inserido em fila alguma. O segmento de dados alocado para o processo continua existindo e não pode ser re-usado, até que o sistema seja reinicializado.

Nome interno: ENDPROCESS.

Parâmetros da primitiva: nenhum.

Função interna chamada: RUNNING.PREEMPTED.

Rotina:

```
PROCEDURE ENDPROCESS;
VAR  P : PROCESSREF;
BEGIN
  P := RUNNING.PREEMPTED;
END "KERNEL EXIT";
```

c) STOPJOB (cancela a execucão de um programa seqüencial) - esta primitiva tem especial aplicação na quebra de laços sem fim ("loops") em programas seqüenciais, pois o cancelamento da tarefa só ocorre no fim do programa, ou quando é executada uma instrução de desvio condicional (FALSEJUMP) no programa. Especificamente, sua função é descontinuar o programa seqüencial sendo executado pelo processo indicado, atribuindo-lhe o resultado desejado conforme o motivo do cancelamento.

Nome interno: STOPJOB.

Parâmetros da primitiva:

P - índice da identificação do processo que executa a tarefa, na lista de processos já inicializados pelo "kernel".

WHY - motivo do cancelamento (resultado atribuido à tarefa).

Funções internas chamadas: nenhuma.

Rotina:

```
PROCEDURE STOPJOB(P, WHY:INTEGER);
VAR  PROC:PROCESSREF;
     HEAD:@HEADTYPE;
BEGIN
  PROC:=RUNNING.PROCESSID(.P.);
  IF PROC = RUNNING.USER
  THEN HEAD:=@RUNNING.HEAD
  ELSE HEAD:=@PROC@.HEAD;
  HEAD@.RESULT:=WHY;
  HEAD@.CONTINUE:=FALSE;
END "KERNEL EXIT";
```

d) WAIT (bloqueia o processo por pequeno intervalo de tempo) - quando a primitiva WAIT é executada, o processo que a chamou, libera a UCP e deixa de concorrer pela mesma, até que o relógio complete mais um segundo (incremente o correspondente contador).

Nome interno: CLOCK.WAIT.

Parâmetros da primitiva: nenhum.

Função interna chamada: SIGNAL.AWAIT.

Rotina:

```
PROCEDURE ENTRY WAIT;
BEGIN
  NEXTTIME.AWAIT;
END "KERNEL EXIT";
```

e) SYSTEMERROR (cancela a execução do programa concorrente) - quando ocorre um erro na execução de um dos

processos do programa concorrente (sistema operacional), esta primitiva é chamada para abortar o sistema, após imprimir, no terminal mestre, uma mensagem indicando o erro ocorrido.

Nome interno: RUNNING.SYSTEMERROR.

Parâmetros da primitiva: nenhum.

Função interna chamada: KERNELERROR.

Rotina:

```
PROCEDURE ENTRY SYSTEMERROR;
CONST  TERMINATED='TERMINATED(:0:)';
       OVERFLOWERR='OVERFLOW ERROR(:0:)';
       POINTERERR='POINTER ERROR(:0:)';
       RANGEERR='RANGE ERROR(:0:)';
       VARIANTERR='VARIANT ERROR(:0:)';
       HEAPLIMIT='HEAP LIMIT(:0:)';
       STACKLIMIT='STACK LIMIT(:0:)';
VAR  TEXT:STRING;
BEGIN
  CASE HEAD.RESULT OF
    0:TEXT:=@TERMINATED;
    1:TEXT:=@OVERFLOWERR;
    2:TEXT:=@POINTERERR;
    3:TEXT:=@RANGEERR;
    4:TEXT:=@VARIANTERR;
    5:TEXT:=@HEAPLIMIT;
    6:TEXT:=@STACKLIMIT;
    END;
  KERNELERROR(TEXT);
END;
```

5.3.2 Primitivas para Implementação de Monitores

As primitivas que implementam monitores são, basicamente, as rotinas da classe GATE. Elas controlam a entrada em cada uma das portas do "kernel" criadas para monitores, e executam as operações definidas sobre variáveis do tipo QUEUE do Pascal Concorrente.

Fazem parte deste grupo, as primitivas, que são descritas a seguir:

- INITGATE;
- ENTERGATE;
- LEAVEGATE;
- DELAYGATE e
- CONTINUEGATE.

a) INITGATE (inicializa porta) - aloca, na memória disponível ("heap") do "kernel", uma porta para representar o monitor sendo inicializado. Esta porta é, então, inicializada (pela classe GATE) com sua fila de espera vazia e permanecendo "fechada" (sem permitir a entrada de outro processo no mesmo monitor) enquanto é executada a rotina de inicialização do monitor. O endereço da porta é passado como parâmetro de retorno, ao processo que chamou a primitiva.

Nome interno: INITGATE.

Parâmetro da primitiva:

G - variável que receberá o endereço da porta.

Funcão interna chamada: NEWCORE.NEW.

Rotina:

```
PROCEDURE INITGATE(VAR G:@GATE);
BEGIN
  G:=NEWCORE.NEW(GATEREC);
  INIT G@;
END "KERNEL EXIT";
```

b) ENTERGATE (solicita acesso exclusivo para entrar em monitor) - se a porta está aberta, o processo entra e fecha-a; em caso contrário, o processo libera a UCP e espera, na fila de entrada da porta, a sua vez de entrar. O processo que chamou a primitiva tem seu contador de chamadas embutidas

de monitores incrementado de uma unidade, e sua prioridade aumentada para o mais alto nível (0).

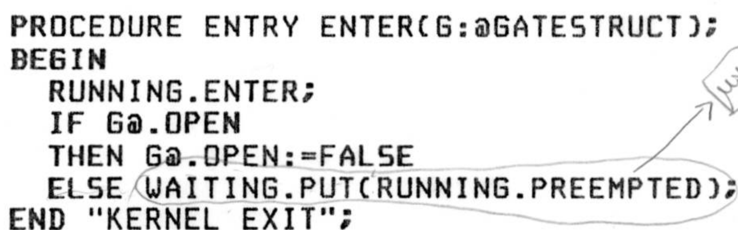
Nome interno: GATE.ENTER.

Parâmetro da primitiva:

G - endereço da porta.

Funções internas chamadas: RUNNING.ENTER, RUNNING.PREEMPTED, PROCESSQUEUE.PUT.

Rotina:



```
PROCEDURE ENTRY ENTER(G:@GATESTRUCT);
BEGIN
  RUNNING.ENTER;
  IF G@.OPEN
  THEN G@.OPEN:=FALSE
  ELSE WAITING.PUT(RUNNING.PREEMPTED);
END "KERNEL EXIT";
```

c) LEAVEGATE (libera a porta de acesso a um monitor) - se nenhum processo está esperando na fila de entrada da porta, esta fica aberta; em caso contrário, o primeiro processo da fila é transferido para a fila "ready", entrando, assim, na porta, que permanece fechada aos demais. Um processo chama esta primitiva quando deixa um monitor, perdendo o acesso a este e tendo seu contador de chamadas embutidas de monitores decrementado de uma unidade. Ainda, sua prioridade é reduzida para o nível mais baixo (2), quando não fica dentro de monitor algum.

Nome interno: GATE.LEAVE.

Parâmetro da primitiva:

G - endereço da porta.

Funções internas chamadas: PROCESSQUEUE.ANY, PROCESSQUEUE.GET, READY.ENTER, RUNNING.LEAVE.

Rotina:

```
PROCEDURE "ENTRY" LEAVE( G:@GATESTRUCT);
VAR  P:PROCESSREF;
BEGIN
  G@.OPEN:=TRUE;
  IF G@.WAITING.ANY
  THEN BEGIN
       G@.OPEN:=FALSE;
       P:=G@.WAITING.GET;
       READY.ENTER(P);
       END;
  RUNNING.LEAVE;
END "KERNEL EXIT";
```

d) DELAYGATE (retarda a execução de um processo dentro de um monitor e libera sua porta) - esta primitiva só pode ser chamada dentro de um monitor. Sua função é transferir o processo "running" para uma fila de um só processo (do tipo QUEUE) do monitor, e liberar o acesso ao mesmo (através de sua porta) para outros processos.

Nome interno: GATE.DELAY.

Parâmetros da primitiva:

G - endereço da porta.

Q - variável que apontará para o descritor do processo retardado.

Funções internas chamadas: RUNNING.PREEMPTED, PROCESSQUEUE.ANY, PROCESSQUEUE.GET, READY.ENTER.

Rotina:

```
PROCEDURE ENTRY DELAY( G:@GATESTRUCT;
                       VAR Q:PROCESSREF);
VAR  P:PROCESSREF;
BEGIN
  Q:=RUNNING.PREEMPTED;
  G@.OPEN:=TRUE;
  IF G@.WAITING.ANY
  THEN BEGIN
       G@.OPEN:=FALSE;
       P:=G@.WAITING.GET;
       READY.ENTER(P);
       END;
END "KERNEL EXIT";
```

e) CONTINUEGATE (continua a execução de um processo bloqueado numa QUEUE dentro de um monitor) - esta primitiva também só pode ser chamada dentro de um monitor. O processo sai do monitor liberando a porta de acesso ao mesmo. Caso haja algum processo na fila indicada, ele é transferido para a fila "ready", a fim de continuar sua execução a partir do ponto em que havia sido bloqueado, e a porta é fechada; senão, é executado o procedimento LEAVEGATE descrito anteriormente no ítem c, podendo a porta do monitor permanecer aberta ou não.

Nome interno: GATE.CONTINUE.

Parâmetros da primitiva:

G - endereço da porta.

Q - variável do tipo QUEUE sobre a qual é realizada a operação.

Funções internas chamadas: READY.ENTER, RUNNING.LEAVE.

Rotina:

```
PROCEDURE ENTRY CONTINUE(G:@GATESTRUCT;
                         VAR Q:PROCESSREF);
VAR  P:PROCESSREF;
BEGIN
  P:=Q;
  IF P=NIL
  THEN LEAVE(G)
  ELSE BEGIN
       Q:=NIL;
       READY.ENTER(P);
       RUNNING.LEAVE;
       END;
END "KERNEL EXIT";
```

5.3.3 Primitiva para Entrada e Saída

Um comando IO do Pascal Concorrente é traduzido, pelo compilador, na chamada de uma rotina do "kernel" que inicia a execução da operação requisitada. Esta rotina é a primitiva IO.

A primitiva IO determina que a prioridade do processo que a chamou passe para nível médio (1), isto se o comando IO não está dentro de um monitor (se a execução está dentro de um monitor, a prioridade permanece no nível zero, o mais alto). Antes de chamar a função interna que executa a operação INITIO para o periférico solicitado, é verificada a conexão do periférico no sistema. Se não foi conectado, a primitiva retorna, de imediato, ao processo, com o estado "failure" para a operação requisitada.

Nome interno : IO.

Parâmetros da primitiva:

BUFFER - endereço do "buffer" para transferência de dados.

PARAM - endereço do registro (RECORD) contendo as especificações da operação requisitada.

DEVICE - código do dispositivo de entrada/saída com o qual a operação deve ser realizada.

Funções internas chamadas: RUNNING.STARTIO, IOFAIL e as funções INITIO dos periféricos programados.

Rotina:

```
PROCEDURE IO( VAR BUFFER:BUFFERTYPE;
              VAR PARAM:@IOPARAM; DEVICE:IODEVICE);
VAR  DEVICECON:BOOLEAN;
```

```
BEGIN
  RUNNING.STARTIO;
  CASE DEVICE OF
    ALMO:DEVICECON:=ALMDTERMINAL.CONNECTED;
    CDISK:DEVICECON:=CARTDISK.CONNECTED;
    TAPEO:DEVICECON:=TAPE1600.CONNECTED;
    PRINTERO:DEVICECON:=SERIALPRINT.CONNECTED;
    CARDREADER:DEVICECON:=CARDREADER.CONNECTED;
    END;
  IF DEVICECON
  THEN CASE DEVICE OF
          ALMO:ALMDTERMINAL.INITIO(BUFFER,PARAM);
          CDISK:CARTDISK.INITIO(BUFFER,PARAM);
          TAPEO:TAPE1600.INITIO(BUFFER,PARAM);
          PRINTERO:SERIALPRINT.INITIO(BUFFER,PARAM);
          CARDREADER:CARDREADER.INITIO(BUFFER,PARAM);
          END
  ELSE IOFAIL(PARAM);
END "KERNEL EXIT";
```

5.3.4 Primitivas Auxiliares

Pertencem a este grupo as primitivas que não se enquadram nos grupos anteriores. Atualmente, existe somente uma primitiva implementada que faz parte deste grupo: REALTIME.

A primitiva REALTIME tem por função passar ao processo "running" o valor correspondente ao tempo transcorrido, em segundos, desde a inicialização do sistema.

Nome interno: CLOCK.REALTIME.

Parâmetros da primitiva: nenhum.

Funções internas chamadas: nenhuma.

Rotina:

```
FUNCTION ENTRY REALTIME:INTEGER;
BEGIN
  REALTIME:=NOW.SEC;
END "KERNEL EXIT";
```

5.4 Tratamento de Interrupções

Além das primitivas, uma outra forma de entrar no "kernel" é através de interrupções geradas pelo "hardware" (ver seção 3.1).

O presente trabalho, numa etapa inicial, considera os três primeiros tipos de interrupções (falta de energia, atraso no tempo de resposta e erro de paridade) como erros fatais, que abortam o sistema. As demais interrupções possíveis são as de relógio e de periféricos. As interrupcões de periféricos não programados (interrupcões imprevistas) também provocam a necessidade de uma nova carga do sistema, como nos erros fatais.

O corpo de cada rotina de tratamento de interrupções de periféricos é apresentado, como função interna, na descrição da classe que implementa o periférico. Nesta seção, estas interrupções recebem um tratamento apenas superficial e quase totalmente fictício, já que o corpo da rotina é endereçado diretamente, quando ocorre a interrupção.

Existem, portanto, basicamente, quatro tipos de interrupções, sob o ponto de vista do sistema:

- erros fatais;
- interrupções imprevistas;
- interrupção de relógio e
- interrupções de periféricos conectados.

a) Erros fatais - indica o erro ocorrido e desativa o sistema.

Rotina:

```
PROCEDURE FATALERROR(INTNUMBER:1..3);
CONST  F='(:13:)F-ERROR.';
       P='(:13:)P-ERROR:';
       T='(:13:)T-ERROR:';
VAR  ADDR:INTEGER;
BEGIN
  IF INTNUMBER=1
  THEN WRITETEXT(@F)
  ELSE BEGIN
         ADDR:=ERRORREG "INSTRUCAO INP 2";
         IF INTNUMBER=2
         THEN WRITETEXT(@P)
         ELSE WRITETEXT(@T);
         WRITEINT(ADDR);
       END;
  CYCLE   END;   "HALT"
"SYSTEMOFF" END;
```

b) Interrupções imprevistas - indica o número da interrupção e desativa o sistema.

Rotina:

```
PROCEDURE XXINTERRUPT(INTNUMBER:0..60);
CONST INT = '(:13:)INT';
BEGIN
  WRITETEXT(@INT);
  WRITEINT(INTNUMBER);
  CYCLE   END;    "HALT"
"SYSTEMOFF" END;
```

c) Interrupção de relógio - ocorre a cada 20 ms, já que o relógio está programado com a freqüência de 50 Hz. São atualizados o período no TIMER e o tempo real no CLOCK, e verificadas as conseqüências destas atualizações para os processos. Por último é chamado o "scheduler" para decidir se o processo "running" continua a ser executado, ou libera a UCP para outros.

Rotina:

```
PROCEDURE CLOCKINT;
VAR  INTERVAL:INTEGER;
BEGIN
  TIMER.TICK(INTERVAL);
  CLOCK.INCREMENT(INTERVAL);
  READY.RESCHEDULE;
  IF RUNNING.USER = NIL
  THEN READY.SELECT;
"KERNELEXIT" END;
```

d) Interrupções de periféricos conectados - chama a rotina da classe correspondente ao periférico, que trata a interrupção ocorrida. Após tratada a interrupção, a saída do "kernel" é feita da mesma maneira como nas primitivas e na interrupção de relógio.

Rotina:

```
PROCEDURE INTERRUPT(DEVICE:IODEVICE);
BEGIN
  CASE DEVICE OF
    ALMO:WHIT ALMDTERMINAL
         DO CASE STATE OF
              PASSIVE,READING:READINT;
              WRITING:WRITEINT;
              END;
    CDISK:CARTDISK.INTERRUPT;
    TAPEO:TAPE1600.INTERRUPT;
    PRINTERO:SERIALPRINT.INTERRUPT;
    END;
  IF RUNNING.USER = NIL
  THEN READY.SELECT;
"KERNELEXIT" END;
```

5.5 Periféricos

Cada periférico da máquina virtual é representado, no "kernel", por uma classe que simula suas operações nos periféricos do computador utilizado - o LABO-8034.

Neste trabalho, estão implementados os seguintes periféricos:

- discos magnéticos (cartucho);
- terminais vídeo-teclado;
- impressora de linhas (serial).

O periférico fita magnética é considerado como parte do sistema, mas, na realidade, está sendo implementado separadamente, e, posteriormente, será integrado ao sistema.

As operações definidas pelas classes dos periféricos são, basicamente, duas:

- INITIO, cuja função é iniciar a execução da operação de IO requisitada e, se isto implica em esperar uma resposta do periférico, liberar a UCP, bloqueando o processo que a solicitou, até que ocorra a interrupção correspondente;
- INTERRUPT, que trata as interrupções sinalizadas pelo periférico, ou seja, se algum processo está a espera de tal interrupção, ele torna-se, novamente, apto a concorrer pela UCP, podendo ser selecionado para execução imediatamente, de acordo com a decisão do "scheduler".

Após concluida uma operação de IO, o estado ("status") resultante de sua execução deve ser indicado ao processo. Inicialmente, ainda na operação INITIO, o estado é suposto "complete", mas se um erro ocorrer na tranferência de dados, o estado é trocado convenientemente, na rotina INTERRUPT.

Alguns periféricos (aqui considerados como os controladores) controlam (isto é, são compostos por) mais de uma unidade de dispositivos de entrada e saída.

Somente um processo de cada vez pode utilizar um periférico (ou unidade do mesmo). Isto deve ser garantido pelo sistema operacional, e não pelo "kernel", cuja função principal é uniformizar as operações de entrada e saída dos periféricos e seus resultados.

5.5.1 Discos magnéticos

A classe que implementa o controlador de discos magnéticos permite a utilização de até 8 discos do tipo cartucho (ver capítulo 3). No entanto, a transferência de dados só pode ser feita com um disco de cada vez.

As operações de IO definidas sobre este periférico são:

a) INPUT - ler um bloco do disco;

b) OUTPUT - gravar um bloco no disco;

c) CONTROL - reinicializar o "kernel", carregando o sistema da área de disco cujo endereço é indicado.

As duas primeiras operações bloqueiam o processo e a terceira cancela a execução do sistema imediatamente.

Os possíveis estados indicados ao processo quando um erro ocorre na execução das operações INPUT e OUTPUT são: "failure", "intervention", e "transmission".

```
VAR  CARTDISK: CLASS;
CONST IOCOUNT = 512 "BYTES"; MAXDISKS = 8;
TYPE IOFUNCTION = ( NOP, RESTORE, SEEK, NOTUSED, READ,
                    WRITE, PROOFREAD, INITIALIZE );

VAR  USER: PROCESSREF;
     ADDR: @IORESULT;
```

```
VAR ENTRY  CONNECTED: BOOLEAN;
VAR "HARDWARE"
     CONTROLSTATUS: PACKED RECORD
                      CLEARED, PARITYERROR, LOSTDATA,
                      NONEXISTMEM, BUSY, NOTUSED1,
                      NOTUSED2, INTERRUPT: BOOLEAN
                    END;

     DRIVESTATUS: PACKED RECORD
                    DRIVEERROR, SEEKERROR, NONEXISTCYL,
                    SEEK, NOTREADY, NOTUSED1, NOTUSED2,
                    CARTRIDGE: BOOLEAN
                  END;

     ERRORREG: ( NOERROR, 1, 2, 3, 4, 5, 6, NONEXISTSECT,
                 8, 9, 10, OVERRUN, 12, NONEXISTDISK, 14 );

     OPCODE: IOFUNCTION;

     CONTROLCOMM: PACKED RECORD
                    CLEARCONTROL, CLEARFLAGS: BOOLEAN;
                    NOTUSED: ARRAY (.1..4.) OF BOOLEAN;
                    CLEARINT: BOOLEAN
                  END;

     DISKADDR: PACKED RECORD
                 DRIVE: 0..3;
                 F: BOOLEAN "DEFECTIVE TRACK";
                 HEAD: 0..3;
                 CYLINDER: 0..407;
                 SECTOR: 0..11;
               END;

     MEMORYADDR: ADDRESS; "EM BYTES"
     PREFIXADDR: 0..15;
     COUNTREG: INTEGER; "EM BYTES"

PROCEDURE ENTRY INITIO( VAR BUF: BUFFERTYPE;
                        VAR COM: @IOPARAM );
VAR  OPONDISK: PACKED RECORD
                 OP: IOFUNCTION; DISK: 0..7
               END;
     IOARG, PREFIX: INTEGER; RESTADDR: ADDRESS;
BEGIN
   COM@.STATUS:=COMPLETE; "A CONFIRMAR-SE ..."
   ADDR:=@COM@.STATUS; "PARA RETIFICAR O STATUS"
   CASE COM@.OPERATION OF
     INPUT, OUTPUT: WITH OPONDISK, DISKADDR DO BEGIN
       IOARG:=COM@.MODIFIER;
       IF COM@.UNIT <> -1
       THEN DISK:=COM@.UNIT MOD MAXDISKS
       ELSE DISK:=SYSTEMUNIT;
       IF COM@.OPERATION = INPUT
       THEN OP:=READ
       ELSE OP:=WRITE;
```

```
      VIRTUAL.REALADDRESS( BUF, PREFIX, RESTADDR );
      MEMORYADDR:=RESTADDR "OUT 605";
      PREFIXADDR:=PREFIX; SECTOR:=IOARG MOD 12 "OUT 604";
      HEAD:= "FACE" ( IOARG DIV 12 + 1 ) MOD 2 +
             2*( "FIX/REM DISK" DISK MOD 2 );
      F:=FALSE; CYLINDER:=IOARG DIV 24 "OUT 603";
      COUNTREG:=IOCOUNT "OUT 606";
      OPCODE:=OP; DRIVE:=DISK DIV 2 "OUT 601";
      USER:=RUNNING.PREEMPTED;
      END;
    MOVE: ;
    CONTROL: INIT KERNEL ( COM@.MODIFIER );
  END;
END "KERNEL EXIT";


PROCEDURE ENTRY INTERRUPT;
VAR  RESULT: IORESULT;
BEGIN
  IF USER <> NIL
  THEN WITH CONTROLSTATUS, DRIVESTATUS, CONTROLCOMM
       DO BEGIN
          RESULT:=COMPLETE;
          IF ERRORREG <> NOERROR   "INP 604"
          THEN BEGIN
               IF ERRORREG = NONEXISTSECT OR
                  ERRORREG = NONEXISTDISK OR
                  ERRORREG = OVERRUN
               THEN RESULT:=FAILURE
               ELSE RESULT:=TRANSMISSION;
               OPCODE:=NOP;
               END
          ELSE IF NONEXISTCYL      "INP 603"
               THEN RESULT:=FAILURE
               ELSE IF DRIVEERROR OR SEEKERROR OR
                       SEEK OR NOTREADY
                    THEN RESULT:=INTERVENTION
                    ELSE IF NONEXISTMEM "INP 602"
                         THEN RESULT:=FAILURE
                         ELSE IF CLEARED OR BUSY OR
                                 PARITYERROR OR LOSTDATA
                              THEN RESULT:=TRANSMISSION;
          IF RESULT <> COMPLETE
          THEN BEGIN
               CLEARFLAGS:=TRUE; CLEARINT:=TRUE
                                 "OUT 602";
               VIRTUAL.GETMAP( USER );
               ADDR@:=RESULT;
               END;
          READY.ENTER( USER );
          USER:=NIL;
          READY.ENDIO;
          END;
END "KERNEL EXIT";
```

```
PROCEDURE INITIALIZE( PRESENTDEV: BOOLEAN );
BEGIN
  CONNECTED:=PRESENTDEV;
  IF CONNECTED
  THEN WITH CONTROLCOMM
       DO BEGIN
          CLEARFLAGS:=TRUE;
          CLEARINT:=TRUE; "OUT 602"
          USER:=NIL;
          END;
END;

BEGIN
  INITIALIZE( DEVICEPRESENT( @CONTROLCOMM ) );
END;
```

5.5.2 Terminais vídeo-teclado

Um controlador de terminais vídeo-teclado sem capacidade de processamento remoto é implementado, no "kernel", por uma classe que controla a execução de comandos de IO, paralelamente, para até 4 terminais (capacidade máxima do controlador).

As operações de IO definidas para terminais são:

a) INPUT - ler um caracter digitado no teclado;

b) OUTPUT - mostrar um caracter no vídeo;

c) MOVE - executar uma função de controle do vídeo (seqüência de caracteres);

d) CONTROL - bloquear o processo "running" até a entrada de um caracter BEL (control G), e liberar a UCP.

Qualquer das operações bloqueia o processo até a completa execução da operação (INPUT, OUTPUT, MOVE) ou a entrada do caracter BEL no teclado do terminal mestre (CONTROL).

Cada terminal (canal do controlador) deve sempre estar em uma, e só uma, das três situações possíveis: PASSIVE, READING e WRITING. Todos os erros que possam ocorrer, durante a execução de uma operação requisitada, são tratados dentro da classe ("kernel"); portanto, o único estado indicado como resposta ao processo que a requisitou é "complete".

Todos os caracteres digitados no teclado, com exceção de BEL para o terminal mestre, são ecoados no vídeo do terminal, se o mesmo está na situação READING, ou ignorados, em caso contrário. Qualquer caracter digitado, só é transferido ao processo depois de ser ecoado. Um caracter CR é transferido ao processo como LF, na entrada; e um LF, na saída, é transformado em CR.

As funções de controle do vídeo, que podem ser solicitadas com a operação MOVE, são mostradas na tabela 1. Nessa tabela são indicados os nomes das funções e os valores correspondentes que devem ser passados ao "kernel" no campo ARG do parâmetro de IO. Qualquer valor diferente dos indicados poderá causar um erro no terminal.

A classe SIGNAL é utilizada para implementar a fila de processos bloqueados a espera da entrada de um caracter BEL.

Função	IOPARAM.ARG
Tabulação (coluna X; linha Y)	Y*256 + X (X=1..80;Y=1..25)
Marcar a posição corrente do cursor	9
Retornar o cursor para a última posição marcada	10
Cursor na linha 0 e coluna 0 (CURSOR HOME)	18
Eliminar a linha em que está o cursor (LINE DELETION)	19
Iniciar campo com iluminação reduzida	25
Inserir uma linha em branco na posição corrente do cursor	26
Limpar o vídeo (CLEAR SCREEN)	28
Limpar todos os campos com iluminação total	29
Iniciar campo com iluminação total	31

Tabela 1: Funções de controle de vídeo e códigos.

```
VAR  ALMDTERMINAL:CLASS;

CONST  NT=4;
       BELCHAR='(:7:)';
       LFCHAR='(:10:)';
       CRCHAR='(:13:)';

TYPE  TERMSTATE=(PASSIVE,READING,WRITING);
      TERMUNITS=0..NT-1;

VAR  TERMINAL:TERMUNITS;
     USER:ARRAY (.TERMUNITS.) OF PROCESSREF;
     CONTROL:ARRAY (.TERMUNITS.) OF INTEGER;
     CHARADDR:ARRAY (.TERMUNITS.) OF BUFFERTYPE;
```

```
     INPUTCHAR:ARRAY (.TERMUNITS.) OF CHAR;
     ECHO:ARRAY (.TERMUNITS.) OF BOOLEAN;
VAR  BELL: SIGNAL;
VAR ENTRY  STATE:ARRAY (.TERMUNITS.) OF TERMSTATE;
           CONNECTED:BOOLEAN;


VAR "HARDWARE"
     RECEIVECHAR:CHAR;
     CONTROLSTATUS:PACKED RECORD
                     CLEARED,PARITYERROR:BOOLEAN;
                     NOTUSED:ARRAY (.1..5.) OF BOOLEAN;
                     INTERRUPT:BOOLEAN
                   END;
     CHANELSTATUS:PACKED RECORD
                    M1,M2,M3,NOTUSED,RECEIVEERROR,XMITBUFEMPTY,
                    SHFTREGEMPTY,RECBUFFULL:BOOLEAN
                  END;
     INTCHANNEL:TERMUNITS;
     INTSELECT:PACKED RECORD
                 CHANNEL:TERMUNITS;
                 LINE:(REC,XMIT);
                 INTERRUPT:(DISABLE,ENABLE)
               END;
     TRANSMITCHAR:CHAR;
     CONTROLCOMM:PACKED RECORD
                   CLEARCONTROL,CLEARFLAGS,NOTUSED1,NOTUSED2
                   CHANNELRESET,CLEARECBFULL,NOTUSED3,
                   INTDISABLE:BOOLEAN
                 END;
     CHANELSELECT:TERMUNITS;

PROCEDURE ENTRY INITIO (VAR BUF:BUFFERTYPE;VAR COM:@IOPARAM);
CONST  LEADIN='(:126:)';
BEGIN
  COM@.STATUS:=COMPLETE;
  TERMINAL:=COM@.UNIT MOD NT;
  CASE COM@.OPERATION OF
    INPUT:BEGIN
          STATE(.TERMINAL.):=READING;
          CHARADDR(.TERMINAL.):=BUF;
          END;
    OUTPUT,MOVE:BEGIN
          STATE(.TERMINAL.):=WRITING;
          IF COM@.OPERATION=OUTPUT
          THEN WRITECHAR(BUF@,TERMINAL)
          ELSE BEGIN
               WRITECHAR(LEADIN,TERMINAL);
               CONTROL(.TERMINAL.):=COM@.MODIFIER;
               END;
          END;
    CONTROL:BELL.AWAIT;
    END;
  IF COM@.OPERATION <> CONTROL
  THEN USER(.TERMINAL.) := RUNNING.PREEMPTED;
END "KERNEL EXIT";
```

```
PROCEDURE ENTRY WRITEINT;
BEGIN
  TERMINAL:=INTCHANNEL  "INP 144";
  IF CONTROL(.TERMINAL.) <> 0
  THEN CONTROLCHAR(@CONTROL(.TERMINAL.),TERMINAL)
  ELSE BEGIN
       WITH INTSELECT
       DO BEGIN
          CHANNEL:=TERMINAL;
          LINE:=XMIT;
          INTERRUPT:=DISABLE  "OUT 140";
          END;
       IF STATE(.TERMINAL.)=WRITING
       THEN BEGIN
            STATE(.TERMINAL.):=PASSIVE;
            IF ECHO(.TERMINAL.)
            THEN BEGIN
                 ECHO(.TERMINAL.):=FALSE;
                 VIRTUAL.GETMAP(USER(.TERMINAL.));
                 CHARADDR(.TERMINAL.)@:=INPUTCHAR(.TERMINAL.);
                 END;
            READY.ENTER(USER(.TERMINAL.));
            USER(.TERMINAL.):=NIL;
            READY.ENDIO;
            END;
       END;
END "KERNEL EXIT";


PROCEDURE ENTRY READINT;
VAR  CH:CHAR;
BEGIN
  TERMINAL:=INTCHANNEL  "INP 144";
  CHANNELSELECT:=TERMINAL  "INP 143";
  IF NOT CHANELSTATUS.RECEIVEERROR  "INP 143"
  THEN BEGIN
       CH:=RECEIVECHAR  "INP 141";
       IF TERMINAL=0 AND CH=BELCHAR
       THEN BELL.SEND
       ELSE IF STATE(.TERMINAL.)=READING
            THEN BEGIN
                 STATE(.TERMINAL.):=WRITING;
                 IF CH=CRCHAR
                 THEN CH:=LFCHAR;
                 INPUTCHAR(.TERMINAL.):=CH;
                 ECHO(.TERMINAL.):=TRUE;
                 WRITECHAR(CH,TERMINAL);
                 END;
       END
  ELSE BEGIN
       CONTROLCOMM.CHANNELRESET:=TRUE  "OUT 142";
       IF STATE(.TERMINAL.) <> WRITING
       THEN WRITECHAR(BELCHAR,TERMINAL);
       END;
END "KERNEL EXIT";
```

```
PROCEDURE ENTRY KERNELWRITE(CH:CHAR);
BEGIN
  CHANELSELECT:=0  "OUT 143";
  REPEAT UNTIL CHANELSTATUS.XMITBUFEMPTY  "INP 143";
  TRANSMITCHAR:=CH  "OUT 141";
  STATE(.0.):=PASSIVE;
END;

PROCEDURE INITIALIZE(PRESENTDEV:BOOLEAN);
VAR  I:TERMUNITS;
BEGIN
  CONNECTED:=PRESENTDEV;
  IF CONNECTED
  THEN BEGIN
       INIT BELL;
       CONTROLCOMM.CLEARFLAGS:=TRUE  "OUT 142";
       FOR I:=0 TO 3
       DO WITH INTSELECT
          DO BEGIN
             CHANNEL:=I;LINE:=REC;
             INTERRUPT:=ENABLE  "OUT 140";
             USER(.I.):=NIL;STATE(.I.):=PASSIVE;
             ECHO(.I.):=FALSE;CONTROL(.I.):=0;
             END;
       END;
END;

PROCEDURE WRITECHAR(CH:CHAR;TERM:TERMUNITS);
BEGIN
  CHANELSELECT:=TERM  "OUT 143";
  WITH INTSELECT
  DO BEGIN
     CHANNEL:=TERM;
     LINE:=XMIT;
     INTERRUPT:=ENABLE  "OUT 140";
     END;
  IF CH=LFCHAR
  THEN TRANSMITCHAR:=CRCHAR
  ELSE TRANSMITCHAR:=CH  "OUT 141";
END;

PROCEDURE CONTROLCHAR(VAR CTL:@INTEGER;TERM:TERMUNITS);
CONST  TABCHAR='(:17:)';
VAR  CH:CHAR; CTLCH:INTEGER;
BEGIN
  CHANELSELECT:=TERM  "OUT 143";
  CTLCH:=CTL@;
  IF CTLCH > 255
  THEN BEGIN
       CTL@:=-CTLCH;
       CH:=TABCHAR;
       END
  ELSE BEGIN
       IF CTLCH < 0
       THEN CTLCH := -CTLCH - 1;
```

```
      CH := CTLCH MOD 256;
      CTL@ := -(CTLCH DIV 256);
      END;
  TRANSMITCHAR:=CH  "OUT 141";
END;

BEGIN
  INITIALIZE(DEVICEPRESENT(@CONTROLCOMM));
END;
```

5.5.3 Impressora

Uma impressora de linhas é simulada por uma classe que implementa o controlador de uma impressora serial.

A impressora serial do LABO-8034 possui dois tracionadores de papel, o que permite a impressão simultânea de dois relatórios distintos, embora só possua um cabeçote de impressão dos caracteres. Entretanto, neste trabalho, somente o tracionador da esquerda é programado, o que simplifica o algorítmo da implementação e torna mais rápida a impressão dos relatórios.

As operações de IO definidas para a impressora de linhas, simulada por esta classe, são:

a) OUTPUT - imprimir uma linha com uma seqüência de caracteres, cujo endereço inicial é indicado, e cujo final é determinado por um caracter NUL ou pelo preenchimento total da linha (132 caracteres);

b) CONTROL - trocar o tamanho da página para o indicado no campo ARG do parâmetro de IO (RECORD); inicialmente, o tamanho é de 66 linhas;

c) INPUT - ler para o campo ARG do parâmetro de IO o tamanho da página correntemente usado pela classe.

A execução da função CONTROL é imediata e o processo recebe do "kernel" (classe) o estado "complete", quando a operação é executada normalmente, ou "failure", quando a impressora está em atividade e, então, a operação não é executada. Também a função INPUT tem execução imediata, e sempre com estado "complete".

Se durante a execução da função OUTPUT, um erro é detectado, a operação é cancelada e o estado "intervention" é indicado ao processo.

A UCP só é liberada (e o processo bloqueado) se a execução da função, pela classe, determinar alguma operação na impressora serial.

A execução da operação OUTPUT envolve as seguintes atividades:

- copiar a linha para o "buffer" da classe (máximo de 132 caracteres), determinando a posição do primeiro caracter imprimível e o número deles na linha;

- copiar o controle de carro indicado no campo ARG do parâmetro de IO, para salto de linhas após a impressão do "buffer";

- se há algum caracter a ser impresso, posicionar o cabeçote e indicar o sentido do movimento, de acordo com a posição atual do mesmo;

- imprimir caracter por caracter na impressora serial, até que todos tenham sido impressos ou ocorra um erro;

- processar o controle de carro e liberar o processo para execução (inserir na fila "ready").

O controle de carro contém duas informações que são processadas nesta ordem:

1) Salto de página - salta para o topo da próxima página, se o "byte" mais à esquerda do controle de carro é diferente de zero;

2) Salto de linhas - salta o número de linhas indicado no "byte" mais à direita (0 a 255).

```
VAR  SERIALPRINT : CLASS ;

CONST  PAGSZ = 66 "LINHAS";
       LINSZ = 132 "CARACTERES";
       SPACE = ' ';
       NUL = '(:0:)' "TERMINO DE LINHA INCOMPLETA";
VAR  BUFFER : ARRAY (.0..LINSZ-1.) OF CHAR;
     ADDR : @IORESULT;
     USER : PROCESSREF;
VAR  NCHARS,
     POSITION,
     CONTROL,SKIP : INTEGER ;
     DIRECTION:(REVERSE,FORWARD);"REV=-1;FOR=+1"
     LINESPERPAGE,
     FREELINES,
     FIRSTPOS,
     LASTPOS : INTEGER;
     BASEINDEX : -1..0;
VAR ENTRY  CONNECTED : BOOLEAN;

VAR "HARDWARE"
     CONTROLSTATUS : PACKED RECORD
                        CLEARED, PARITYERROR, ENDSWITCH,
                        CARRTRANSERROR, LIDOPEN, NOTUSED,
                        PRINTERINT, CONTROLINT : BOOLEAN
                     END;
     PRINTERSTATUS : PACKED RECORD
                        DEVICEERROR, HOMEPOSITION,NOTUSED,
                        READYFORPRINT,PRINTFLAG,TABFLAG,
                        ENDOFPAPER : BOOLEAN
                     END;
     LEPSTATUS : PACKED RECORD
                    LEPCLEARED,LPARITYERROR,LEP1INT,LEP2INT,
                    LEPTRANSCLOCK,LEP1FEEDFLAG,LEP2TRANSCLOCK,
                    LEP2FEEDFLAG : BOOLEAN
                 END;
```

```
      CARRPOSITION : 0..180;
      PRINTCHAR : CHAR;
      FUNCTIONCODE : PACKED RECORD
                        CASE TAG : (CTL,FLG,DIR,INT) OF
                        CTL : (CLEARCONTROL : BOOLEAN);
                        FLG : (CLEAR : (FLAGS,INT));
                        DIR : (PRINT : (REVERSE,FORWARD));
                        INT : (INTERRUPT : (ENABLE,DISABLE))
                     END;
      LEPFUNCTCODE : (CLEARLEPLOGIC,CLEARFLAGS,CLEARINTLEP1,
                      CLEARINTLEP2);
      PAPERFEED : PACKED RECORD
                     LINECOUNT : 0..63;
                     LEP : (LEP1,LEP2)
                   END;


PROCEDURE ENTRY INITIO (VAR BUF : BUFFERTYPE;
                        VAR COM:@IOPARAM);
CONST  LINELENGTH=LINSZ;
VAR  I:INTEGER;
     CH:CHAR;
     FREE:BOOLEAN;
BEGIN
  IF PRINTERSTATUS.DEVICEERROR OR PRINTERSTATUS.ENDOFPAPER
                                          "INP 101"
  THEN BEGIN
       ERROR;
       COM@.STATUS:=INTERVENTION;
       END
  ELSE BEGIN
       COM@.STATUS:=COMPLETE;
       CASE COM@.OPERATION OF
         OUTPUT:BEGIN
                ADDR:=@COM@.STATUS;
                CONTROL:=COM@.ARG;
                NCHARS:=-1;
                I:=0;
                REPEAT
                  CH:=BUF@(.I.);
                  BUFFER(.I.):=CH;
                  IF CH>SPACE
                  THEN BEGIN
                       NCHARS:=NCHARS+1;
                       IF NCHARS=0
                       THEN FIRSTPOS:=I;
                       END;
                  I:=I+1;
                UNTIL (CH=NUL) OR (I=LINELENGTH);
                LASTPOS:=I;
                WITH LEPSTATUS
                DO FREE:=NOT(LEP1INT OR LEP1FEEDFLAG);
                                         "INP 104;INP 105"
                NCHARS:=NCHARS+1;
```

```
                IF NCHARS>0
                THEN BEGIN
                     IF FREE
                     THEN INITWRITE;
                     USER:=RUNNING.PREEMPTED;
                     END
                ELSE BEGIN
                     LINECONTROL;
                     IF FREE
                     THEN SKIPCOMMAND;
                     END;
                END;
         INPUT:COM@.ARG:=LINESPERPAGE;
         CONTROL:IF (SKIP=0) AND (NCHARS=0)
                 THEN BEGIN
                      LINESPERPAGE:=COM@.ARG;
                      FREELINES:=LINESPERPAGE;
                      END
                 ELSE COM@.STATUS:=FAILURE;
         MOVE:;
         END;
       END;
END "KERNEL EXIT";


PROCEDURE ENTRY INTERRUPT;
VAR  DONE:BOOLEAN;
BEGIN
  WITH CONTROLSTATUS, PRINTERSTATUS, LEPSTATUS, FUNCTIONCODE
  DO BEGIN
     IF PRINTERINT  "INP 102"
     THEN BEGIN
          TAG:=FLG;CLEAR:=INT;  "OUT 102"
          IF CLEARED OR PARITYERROR OR ENDSWITCH OR
             CARRTRANSERROR OR LIDOPEN
          THEN BEGIN
               ERROR;
               VIRTUAL.GETMAP(USER);
               ADDR@:=INTERVENTION;
               DONE:=TRUE;
               END
          ELSE IF NCHARS>0
               THEN BEGIN WRITECHAR;DONE:=FALSE;END
                                                 "INP 101"
               ELSE IF NOT PRINTFLAG
                    THEN BEGIN
                         LINECONTROL;
                         SKIPCOMMAND;
                         DONE:=TRUE;
                         END;
```

```
            IF DONE
            THEN BEGIN
                 READY.ENTER(USER);
                 USER:=NIL;
                 READY.ENDIO;
                 END;
            END;
       IF LEP2INT OR LPARITYERROR  "INP 104"
       THEN ERROR;
       IF LEP1INT
       THEN BEGIN
            LEPFUNCTIONCODE:=CLEARINTLEP1;  "OUT 104"
            IF NOT PRINTFLAG                "INP 101"
            THEN BEGIN
                 SKIPCOMMAND;
                 IF NOT LEP1FEEDPAPER AND (NCHAR>0)  "INP 105"
                 THEN INITWRITE;
                 END;
            END;
       END;
END "KERNEL EXIT;


PROCEDURE ERROR;
BEGIN
  LEPFUNCTCODE:=CLEARINTLEP2;  "OUT 104"
  LEPFUNCTCODE:=CLEARFLAGS;    "OUT 104"
  WITH FUNCTIONCODE
  DO BEGIN
     TAG:=FLG;CLEAR:=FLAGS;      "OUT 102"
     TAG:=INT;INTERRUPT:=ENABLE;"OUT 102"
     END;
END;


PROCEDURE INITIALIZE(PRESENTDEV:BOOLEAN);
BEGIN
  CONNECTED:=PRESENTDEV;
  IF CONNECTED
  THEN BEGIN
       LEPFUNCTCODE:=CLEARLEPLOGIC;  "OUT 104"
       WITH FUNCTIONCODE
       DO BEGIN
          TAG:=FLG;CLEAR:=FLAGS;      "OUT 102"
          TAG:=INT;INTERRUPT:=ENABLE;"OUT 102"
          END;
       USER:=NIL;
       NCHARS:=0;
       LINESPERPAGE:=PAGSZ;
       FREELINES:=LINESPERPAGE;
       SKIP:=0;
       END;
END;
```

```
PROCEDURE INITWRITE;
VAR  I:INTEGER;
BEGIN
  I:=CARRPOSITION;
  IF I-FIRSTPOS <= LASTPOS-I    "INP 103"
  THEN BEGIN
       POSITION:=FIRSTPOS;
       DIRECTION:=FORWARD;
       BASEINDEX:=0;
       END
  ELSE BEGIN
       POSITION:=LASTPOSITION;
       DIRECTION:=REVERSE;
       BASEINDEX:=-1;
       END;
  WITH FUNCTIONCODE
  DO BEGIN
     TAG:=DIR;PRINT:=DIRECTION;
     END;
  CARRPOSITION:=POSITION;
  WRITECHAR;
END;


PROCEDURE LINECONTROL;
VAR  LINES:0..255;
BEGIN
  IF CONTROL DIV 256 <> 0
  THEN BEGIN
       SKIP:=SKIP+FREELINES;
       FREELINES:=LINESPERPAGE;
       END;
  LINES:=CONTROL MOD 256;
  SKIP:=SKIP+LINES;
  FREELINES:=FREELINES - LINES;
  WHILE FREELINES < 0
  DO FREELINES:=FREELINES+LINESPERPAGE;
END;


PROCEDURE SKIPCOMMAND;
CONST  MAXSKIP=63  "LINES";
VAR  LINES:0..MAXSKIP;
BEGIN
  LINES:=SKIP;
  IF LINES > 0
  THEN BEGIN
       IF LINES > MAXSKIP
       THEN LINES:=MAXSKIP;
       SKIP:=SKIP-LINES;
       PAPERFEED.LINECOUNT:=LINES;
       PAPERFEED.LEP:=LEP1;         "OUT 105"
       END;
END;
```

```
PROCEDURE WRITECHAR;
VAR  CH:CHAR;
BEGIN
  WHILE PRINTERSTATUS.READYFORPRINT AND (NCHARS>0)  "INP 101"
  DO BEGIN
     CH:=BUFFER(.BASEINDEX+POSITION.);
     PRINTCHAR:=CH;                                  "OUT 101"
     IF DIRECTION=FORWARD
     THEN POSITION:=POSITION+1
     ELSE POSITION:=POSITION-1;
     IF CH <> SPACE
     THEN NCHARS:=NCHAR-1;
     END;
END;


BEGIN
  FUNCTIONCODE.TAG:=CTL;
  INITIALIZE(DEVICEPRESENT(@FUNCTIONCODE));
END;
```

6 INTERPRETADOR

O interpretador da máquina virtual é um conjunto de rotinas que executam, no LABO-8034, as instruções virtuais daquela máquina hipotética. Sua programação baseia-se na mesma técnica utilizada por Brinch Hansen[1] na versão original - "threaded code".[11]

6.1 A Máquina Virtual

A máquina virtual é constituida por um processador e alguns periféricos virtuais. Os periféricos estão implementados no "kernel" e descritos no capítulo dedicado ao mesmo (seção 5.5). O processador virtual consiste de um conjunto de registradores e do interpretador das instruções virtuais.

Alguns registradores da máquina virtual são implementados pelos próprios registradores do computador (R), outros são palavras de memória usadas para este fim específico (MREG). A tabela 2 mostra os registradores e suas funções.

Os registradores que tem funções fixas durante a execução de um processo são também mostrados nas figuras 1, 8 e 9. Os outros são registradores de trabalho. O "heap top" H, que aparece na figura 1, não é considerado aqui como um registrador, e sim, como um apontador que faz parte do descritor do processo.

Registrador	Implementação	Função
W	R1	Acumulador
X	R3	Acumulador e indexador
Q	R0	PC da máquina virtual
S	R2	"stack pointer"(topo da pilha)
P	PC	PC da máquina real
B	MREG+0	Registrador base local
G	MREG+1	Registrador base global
TA	MREG+2	Registrador temporário A
TB	MREG+3	Registrador temporário B
TC	MREG+4	Registrador temporário C
FE	MREG+5	Registrador de expoente (PF)
FM	MREG+6	Registrador de mantissa (PF)
Y	R2	Acumulador e indexador
Z	R0	Acumulador

Tabela 2: Registradores da máquina virtual.

Para usar R0 e R2 como Z e Y, é necessário salvar os valores de Q e S, neles armazenados, nos registradores temporários, e restaurá-los posteriormente.

O interpretador consiste das rotinas que implementam as instruções virtuais e de uma tabela com o endereço inicial de cada rotina.

Para utilizar a técnica do "threaded code" de Bell[11], define-se uma pseudo-instrução NEXT, que deve ser a última

instrução de cada rotina. NEXT é definida como uma MACRO em Assembler:

```
MACRO NEXT
MOV   Q,X
LDA   X,0,X
INC   Q,Q
JMP   @TABLE-1,X
MFIM;
```

onde TABLE é o endereço inicial da tabela de endereços. Sua ação é a seguinte:

- busca no código virtual uma nova instrução, usando Q como indicador;
- incrementa Q;
- usando o código da instrução como índice, toma, da tabela, o endereço da rotina correspondente e põe em P (instrução JMP).

O processador da máquina virtual é orientado para utilização de pilha ("stack"). Isto significa que os operandos das instruções, na sua maioria, são retirados da pilha, processados e o resultado inserido no topo da pilha. Assim, uma operação ADDWORD, por exemplo, é processada da seguinte forma:

- retira do topo da pilha uma palavra, que é utilizada como primeiro operando - o apontador do topo é incrementado (a pilha cresce em sentido contrário ao crescimento dos endereços de memória);
- retira a palavra seguinte no topo da pilha, para ser usada como segundo operando - o apontador do topo é novamente incrementado;
- soma os valores dos dois operandos;

- insere o resultado no topo da pilha, decrementando o apontador do topo.

Para implementar estas operações de inserção (PUSH) e retirada (POP) numa pilha, com decrementos e incrementos automáticos do registrador que aponta para o topo da mesma, é necessário definir algumas pseudo-instruções (macros, no Assembler), pois o computador não possui instruções específicas para este fim, nem facilidades para o processamento de pilha.

```
MACRO  PUSH AC
NEG  S,S
COM  S,S
STA  AC,0,S
MFIM

MACRO  POP  AC
LDA    AC,0,S
INC    S,S
MFIM
```

Também são definidas instruções para leitura (SREAD) e gravação (SWRIT) de uma palavra, numa posição qualquer da pilha, relativa ao topo.

```
SREAD  AC,PR = LDA  AC,PR,S

SWRIT  AC,PR = STA  AC,PR,S
```

Os símbolos AC e PR são, respectivamente, o acumulador usado como portador da informação a ser transferida para ou da pilha, e a posição relativa ao topo da pilha a ser processada.

As operações de multiplicação e divisão são também, em virtude da simplicidade do processador do LABO, implementadas através de macros.

```
MACRO   MUL AC1,AC2,AC3,N
CLRA    AC3,AC3
SHLR    AC2,AC2,SZC
ADDZ    AC1,AC3
SKNZ    AC2
JMP     .+3+N
SHLL    AC1,AC1,SNC
JMP     .-5
MFIM
```

```
MACRO   DIV AC1,AC2,AC3,ACT
LDA     ACT,DN016  ;(DN016  é  -16)
SHLL    AC1,AC3
CLRA    AC1,AC1
SHRL    AC1,AC1
CSGR    AC2,AC1
SUB     AC2,AC1
SHRL    AC3,AC3
ISZA    ACT,ACT
JMP     .-5
MFIM
```

Na multiplicação (MUL), AC1, AC2, e AC3 são os acumuladores usados, respectivamente, como multiplicando, multiplicador e produto. Tanto os fatores quanto o produto são de 16 bits. Com isto, é possível a ocorrência de "overflow". Quando ele ocorre, a instrução seguinte a chamada de MUL é executada. Se a operação é executada sem "overflow", N indica o número de palavras a serem saltadas depois da chamada de MUL. Na divisão (DIV), divide-se uma palavra (dividendo AC1) por uma palavra (divisor AC2), para resultar um quociente de uma palavra (AC3) e um resto (AC1) menor que o divisor. ACT é um acumulador usado para trabalho. Todos os operandos são considerados sem sinal. Os sinais e a divisão por zero devem ser analisados nas rotinas que usam as macros MUL e DIV.

6.2 As Instruções Virtuais

Cada instrução tem um código fixo, entre 1 e 112. Algumas tem um ou mais operandos, cada um representado numa palavra do código virtual. A instrução CASEJUMP tem número variável de operandos.

Uma lista de todas as instruções com os respectivos códigos encontra-se no Apêndice A. São sinalizadas as instruções que envolvem comunicação com o "kernel" e aquelas só geradas pelo Pascal Seqüencial ou só pelo Pascal Concorrente.

A comunicação com o "kernel" - chamada de primitivas - é feita através de uma MACRO que copia em OPLINE (no descritor do processo) o número da linha corrente do programa fonte e executa a instrução KTRAP do Assembler, que simula a sinalização de uma interrupção por "software" (ver seção 5.3).

Para cada processo que é criado, o "kernel" inicializa o registrador P com o endereço inicial do interpretador. Neste endereço, STARTADDRESS, há, simplesmente, uma pseudo-instrução NEXT, que toma a primeira instrução virtual do código objeto do processo e desvia para a rotina que executa a instrução. Daí em diante, cada rotina termina, de maneira semelhante, com um desvio para a rotina que executa a próxima instrução virtual, ou para a rotina que analisa uma condição de erro ou o término de programa seqüencial.

As instruções da máquina virtual são apresentadas a seguir, de maneira sucinta, dando, em alguns casos, detalhes que as relacionam com os comandos do Pascal. Os operandos das

instruções são indicados entre parênteses.

a) CONSTANTES - as constantes de uma só palavra são diretamente colocadas na pilha pela instrução

PUSHCONST (VALOR);

para os outros tipos de constantes, apenas os endereços iniciais das mesmas são empilhados pela instrução

CONSTADDR (DESL),

para posterior processamento por outras instruções (PUSHREAL, PUSHSET). No segundo caso, DESL é um deslocamento em relação ao endereço inicial da área de constantes do programa concorrente, ou do seqüencial sendo executado por um de seus processos.

b) ENDEREÇOS RELATIVOS - endereços de variáveis locais, globais e de rotinas do tipo "process entry" são, respectivamente, empilhados pelas instruções

LOCALADDR (DESL),

GLOBALADDR (DESL) e

PUSHLABEL (DIST).

Os operandos são endereços relativos, respectivamente, aos registradores B, G e Q.

c) ENDEREÇOS RELATIVOS INDIRETOS - os valores armazenados nos endereços a que se referem as duas primeiras instruções do ítem anterior são, respectivamente, empilhados pelas instruções

PUSHLOCAL (DESL) e

PUSHGLOBAL (DESL).

d) ENDEREÇOS NA PILHA - o valor armazenado no topo da pilha é retirado da mesma e usado como endereço absoluto

pelas instruções

PUSHIND,

PUSHBYTE,

PUSHREAL e

PUSHSET,

para empilhar, respectivamente, um endereço indireto ou valor de uma variável, um caracter (colocado no "byte" da direita de uma palavra), um valor do tipo REAL e um conjunto (estrutura do tipo SET).

e) COMPONENTES DE ESTRUTURAS - os endereços de elementos de estruturas dos tipos ARRAY e RECORD são colocados na pilha, respectivamente, pelas instruções

INDEX (MIN,MAX-MIN,COMPRB) e

FIELD (DESL),

depois de calculados, utilizando, da pilha, o endereço da estrutura e, dos operandos, COMPRB (comprimento, em bytes, de cada componente do ARRAY) ou DESL (endereço do componente, relativo ao início do RECORD). No primeiro caso, o índice é retirado da pilha e, após verificados seus limites, usado no cálculo do endereço.

f) COMPATIBILIDADE E CONSISTÊNCIA DE TIPOS - estas verificações são feitas pelas instruções

POINTER,

RANGE (MIN,MAX) e

VARIANT (DESL,TAGSET),

que, respectivamente, verificam se um "pointer" no topo da pilha é igual a zero (POINTERERROR) ou não, se um valor no topo da pilha está no intervalo da enumeração indicada nos operandos

ou não (RANGEERROR), e se o valor do "tag" e variante assumida são consistentes ou não (VARIANTERROR).

g) ATRIBUIÇÕES - para as atribuições, são tomados do topo da pilha, nesta ordem, o valor a ser atribuido e o endereço da variável que o recebe. As atribuições para os diversos tipos de variáveis são feitas pelas instruções

COPYBYTE, que altera só o "byte" indicado,

COPYWORD,

COPYREAL,

COPYSET,

COPYTAG (COMPRW), que faz também a inicialização do componente variante,

COPYSTRUC (COMPRW), para a qual o primeiro operando na pilha é o endereço da estrutura fonte, e não o conteúdo da mesma.

h) ALOCAÇÃO DO "HEAP" - a alocação de variáveis referenciadas por "pointers" é feita pelas instruções

NEW (STACKLENGTH+COMPRW,COMPRW) e

NEWINIT (STACKLENGTH+COMPRW,COMPRW),

que, após verificada a existência de espaço suficiente no "heap", aloca nele COMPRW palavras, atribuindo o endereço desta área à variável ("pointer") cujo endereço está no topo da pilha e atualizando o HEAPTOP. A segunda instrução inicializa a área com zeros.

i) OPERAÇÕES ARITMÉTICAS - as operações aritméticas definidas sobre um ou dois operandos, ambos inteiros ou reais, no topo da pilha, são executadas pelas instruções

NEGWORD, NEGREAL,

ADDWORD, ADDREAL,

SUBWORD, SUBREAL,

MULWORD, MULREAL,

DIVWORD, DIVREAL,

MODWORD,

cujos resultados, do mesmo tipo dos operandos, são colocados na pilha. As condições de OVERFLOWERROR são testadas nas instruções em que podem ocorrer.

j) OPERAÇÕES RELACIONAIS - dois operandos do mesmo tipo (enumerações, reais) ou os endereços de duas estruturas compatíveis (ARRAY, RECORD) são retirados da pilha para comparação de seus valores pelas instruções

LSWORD, LSREAL,

EQWORD, EQREAL,

GRWORD, GRREAL,

NLWORD, NLREAL,

NEWORD, NEREAL,

NGWORD, NGREAL,

EQSTRUCT (COMPRW), NESTRUCT(COMPRW),

e ainda, só para "strings" de caracteres,

LSSTRUCT (COMPRW), GRSTRUCT (COMPRW)

NLSTRUCT (COMPRW), NGSTRUCT (COMPRW).

O resultado é um valor booleano, que vai para a pilha.

l) OPERAÇÕES LÓGICAS - um ou dois valores booleanos - 0("false") ou 1("true") - representados em palavras, no topo da pilha, são substituidos pelo valor booleano resultante da execução sobre eles de uma das instruções:

NOT, ANDWORD e ORWORD.

m) OPERAÇÕES SOBRE CONJUNTOS - na tabela 3 são mostradas as instruções que operam sobre conjuntos e seus elementos, os quais são substituidos na pilha pelo conjunto ou valor booleano resultante da operação.

Instrução	Operação	Operandos (na pilha)	Resultado (na pilha)	Erro Possível
BUILDSET	inclusão de elemento	elem/conj	conj	RANGEERROR
ANDSET	intersecção	conj/conj	conj	-
ORSET	união	conj/conj	conj	-
SUBSET	diferença	conj/conj	conj	-
INSET	pertence	conj/elem	booleano	RANGEERROR
EQSET	igual	conj/conj	booleano	-
NLSET	contém	conj/conj	booleano	-
NESET	diferente	conj/conj	booleano	-
NGSET	está contido	conj/conj	booleano	-

Tabela 3: Operações sobre conjuntos.

n) DESVIOS LOCAIS - a seqüência da execução das instruções virtuais pode ser alterada pelas instruções

JUMP (DIST),

FALSEJUMP (DIST) e

CASEJUMP (MIN,MAX-MIN,DISTmin,...,DISTmax),

que executam desvios incondicional, condicional com o valor booleano no topo da pilha (desviando se o valor é "false") e controlado pelo elemento de uma enumeração no topo da pilha, respectivamente. O endereço de desvio é calculado com a

distância DIST relativa ao valor do registrador Q.

o) CHAMADAS DE ROTINAS - as chamadas de programas seqüenciais, subprogramas, rotinas de inicialização e ativação de processos são executadas pelas instruções da tabela 4. ENTRY é um deslocamento relativo ao topo da pilha do processo, usado para obter o endereço absoluto da rotina. INITCLASS e INITMON copiam os parâmetros para a área de variáveis da classe ou monitor que está sendo inicializado, e INITPROC chama o "kernel", que inicializa o processo, copiando os parâmetros em sua área. PARAMLENGTH, VARLENGTH e STACKLENGTH são usados, pelo "kernel", para calcular o tamanho do segmento privado de dados do processo. As cinco primeiras instruções salvam, na pilha, o endereço de retorno (valor corrente de Q). CALLPROG ainda salva, na pilha, o endereço da área de constantes do programa.

Instrução	Chamada de	Operandos da instrução	Operandos no topo da pilha
CALL	subprograma do programa	DIST	-
CALLSYS	"process entry"	ENTRY-2	-
CALLPROG	programa seqüencial	-	endereço do código
INITCLASS	rotina inicial de classe	PARAMLENGTH DIST	parâmetros e endereço da classe
INITMON	rotina inicial de monitor	PARAMLENGTH DIST	parâmetros e endereço do monitor
INITPROC	processo (ativação)	PARAMLENGTH VARLENGTH STACKLENGTH DIST	parâmetros (processados no "kernel")

Tabela 4: Chamadas de rotinas.

p) INICIALIZAÇÃO DE VARIÁVEIS LOCAIS - a área de variáveis locais de programas e subprogramas Pascal Seqüencial pode ser inicializada com zeros pela instrução

INITVAR (COMPRW).

Antes de um subprograma do tipo FUNCTION ser chamado, é necessário alocar um espaco na pilha, para receber o valor da função, determinado dentro do subprograma. O tamanho do espaço alocado depende do tipo da função (INTEGER ou REAL, em função simples, ou em função que seja entrada de classe ou monitor). Isto é feito pela instrução

FUNCVALUE (TIPO),

onde TIPO é 0, 1, 2 ou 3, conforme os tipos acima descritos.

q) ENTRADA EM ROTINAS - a primeira instrução de qualquer rotina que pode ser chamada por uma das instruções do ítem 'o', deve ser uma das seguintes:

ENTER (STACKLENGTH,POPLENGTH,LINHA,VARLENGTH),

ENTERCLASS (STACKLENGTH,POPLENGTH,LINHA,VARLENGTH),

ENTERMON (STACKLENGTH,POPLENGTH,LINHA,VARLENGTH),

para as chamadas por CALL;

ENTERPROC (STACKLENGTH,POPLENGTH,LINHA,VARLENGTH),

para as chamadas por CALLSYS;

ENTERPROG (POPLENGTH,LINHA,STACKLENGTH,VARLENGTH),

para os programas seqüenciais, chamados por CALLPROG;

BEGINCLASS (STACKLENGTH,5,LINHA,0),

BEGINMON (STACKLENGTH,5,LINHA,0),

para rotinas de inicialização de classes e monitores, chamadas, respectivamente, por INITCLASS e INITMON;

BEGINPROC (LINHA),

para processos de programas concorrentes, ativados (inicializados) pela instrução INITPROC.

Os parâmetros são usados para a determinação do espaço de dados necessário à rotina e para manter o número da linha do programa fonte onde a mesma inicia, o que auxilia na localização de erros. Todas as instruções, exceto BEGINPROC, executam as seguintes tarefas:

- verificar existência de espaço suficiente na pilha (se não há , ocorre erro STACKLIMIT);
- salvar, na pilha, os valores dos registradores G, B e S antes da chamada, para restaurá-los no retorno;
- estabelecer os novos valores dos registradores B, S e G (G permanece o mesmo com ENTER).

ENTERPROG, além disto, sinaliza, no componente JOB do registro descritor do processo, a entrada num programa seqüencial; ENTERPROC, ao contrário, sinaliza a saída.

BEGINPROC simplesmente troca o número da linha do programa fonte, para indicar a linha inicial do processo.

r) SAÍDA DE ROTINAS - as rotinas referenciadas no ítem anterior terminam, respectivamente, pelas instruções:

EXIT,

EXITCLASS,

EXITMON,

EXITPROC,

EXITPROG,

ENDCLASS,

ENDMON e

ENDPROC.

ENDPROC simplesmente termina o processo; as outras restauram os registradores B, G, S e Q para os valores que possuiam antes da chamada, retornando, assim, para a rotina que chamou a que está sendo deixada. EXITPROG sinaliza, em JOB, a saída do programa seqüencial e EXITPROC, ao contrário, sinaliza o retorno ao programa seqüencial que chamou a "process entry".

s) OUTRAS OPERAÇÕES - as operações apresentadas na tabela 5 são usadas pelos compiladores Pascal Concorrente e Seqüencial como funções pré-definidas, com exceção das duas primeiras, que são usadas na geração de código dos comandos FOR. Em ABSWORD e TRUNCREAL é detectada a ocorrência de OVERFLOWERROR.

Instrução	Operando na pilha	Resultado na pilha	Operação
INCRWORD	endereço da palavra	-	incrementa o conteúdo da palavra endereçada
DECRWORD	"	-	decrementa o conteúdo da palavra endereçada
ABSWORD	INTEGER	INTEGER	valor absoluto inteiro
ABSREAL	REAL	REAL	valor absoluto real
TRUNCREAL	REAL	INTEGER	real --> inteiro
CONVWORD	INTEGER	REAL	inteiro --> real
SUCCWORD	enumeração	enumeração	I:=I+1
PREDWORD	"	"	I:=I-1
EMPTY	QUEUE	booleano	(valor = 0)

Tabela 5: Operações especiais.

t) INSTRUÇÕES DE CONTROLE DO SISTEMA - neste grupo, estão instruções que, em conjunto com o "kernel", controlam o estado do sistema e possibilitam a troca de informações entre programas e "kernel" (através dos registros descritores de processos, ou periféricos). As instruções são as seguintes:

- ATTRIBUTE - um atributo do descritor do processo corrente, cujo número de ordem está na pilha, é lido e seu valor colocado na pilha;

- REALTIME - copia, da classe CLOCK do "kernel" para o topo da pilha, o tempo decorrido, em segundos, desde a inicialização do sistema;

- NEWLINE(NÚMERO) - marca no descritor do processo o número de uma nova linha do programa fonte;

- SETHEAP - troca o HEAPTOP do processo para o novo endereço que é tomado do topo da pilha (HEAPTOP pode ser lido através de ATTRIBUTE);

- STOP - estabelece um resultado com o qual, o programa seqüencial chamado por um processo, será abortado na execução da próxima instrução FALSEJUMP (o resultado a ser atribuido e a identificação do processo, obtida por ATTRIBUTE previamente, são retirados da pilha);

- START - evita o cancelamento automático da execução do próximo programa seqüencial pelo processo corrente, devido a execução prévia de uma instrução STOP para este processo;

- WAIT - retarda a execução do processo corrente até o próximo sinal de 1 segundo no marcador de tempo real do sistema;

- DELAY - retarda a execução do processo corrente, dentro de um monitor, colocando uma referência ao processo numa variável tipo QUEUE e liberando o monitor, até que outro processo, entrando neste monitor, execute a instrução CONTINUE sobre a mesma variável (o endereço da porta que implementa o monitor está na pilha, apontado pelo registrador G, e o endereço da variável QUEUE está no topo da pilha);

- CONTINUE - reinicia a execução, dentro do monitor indicado por G, do processo que foi retardado por uma instrução DELAY sobre uma variável tipo QUEUE, cujo endereço está no topo da pilha (se nenhum processo foi colocado naquela QUEUE, o monitor é deixado livre);

- IO - esta é a única instrução de entrada e saída da máquina virtual; seus operandos estão no topo da pilha e representam o seguinte, na ordem de retirada: uma constante indicando o periférico com o qual deve ser realizada a operação, o endereço de um registro que especifica a operação e o endereço da variável ("buffer") utilizada para a transferência dos dados (o registro da operação contém informações sobre a operação a ser realizada, o estado resultante, a unidade do periférico e um argumento, que são utilizadas pelo "kernel", dependendo do periférico);

- POP(COMPRW) - ajusta o comprimento da pilha, retirando da mesma informações que não são mais necessárias e que não foram retiradas automaticamente no decorrer de seu uso; o registrador S tem seu valor adicionado de COMPRW.

7 INICIALIZAÇÃO DO SISTEMA

O sistema é carregado na memória em duas fases. Na primeira, são carregados o "kernel" e o interpretador, partindo da posição zero da memória. Daí, o controle é passado ao "kernel" que chama uma rotina de carga do sistema operacional. Após a carga do sistema operacional, o "kernel" passa a executar os procedimentos de inicialização de seus componentes e criação do processo inicial do programa.

O carregador do "kernel" é um pequeno programa, em linguagem de máquina, gravado no bloco zero do disco do sistema. Ele é chamado pelo "Bootstrap Loader" do computador, que é carregado na memória principal automaticamente quando a máquina é ligada, ou por uma rotina do "kernel" que reinicializa a máquina virtual.

7.1 Carga do Sistema

O carregador é lido, do bloco zero do disco do sistema em uso, para a posição 26000_8 da memória, durante a execução do "Bootstrap Loader" (IPL) ou na reinicialização do sistema (operação "control" para disco). Este carregador tem dois endereços de entrada alternativos. O primeiro (26000_8) é assumido durante um IPL e o segundo (26005_8), nas chamadas para reinicialização. A diferença está na determinação de dois indicadores: a unidade de disco e o segmento de disco do sistema operacional a ser carregado. No caso de IPL, o primeiro

(indicador de unidade de disco) é copiado da área de variáveis do "bootstrap" e o segundo, por falta, indica o segmento do sistema operacional permanente (primeiro sistema - endereço 24). No caso de reinicialização, estes indicadores são passados como parâmetros ao carregador.

Ao receber o controle, o carregador lê, do disco para a memória (endereco 0), o código de máquina do "kernel" e do interpretador, e verifica os conteúdos de duas posições da memória, para uma conferência grosseira da carga do "kernel". A seguir, o número da unidade de disco e o endereço de disco do sistema operacional são passados ao "kernel". Se ocorre um erro na carga ou se os endereços testados não conferem com o esperado, é mostrada, no console, a mensagem

KERNEL LOAD ERROR

e provocado um novo IPL. Na ausência de erro, o carregador passa a fazer o teste e inicialização da memória acima de 64 KB, determinando a capacidade máxima de memória sem defeito. Este valor é passado ao "kernel", como tamanho total da memória disponível.

Após a carga dos programas que implementam a máquina virtual, o carregador passa o controle ao "kernel", desviando para o endereço zero da memória. Neste endereço existe uma instrução de desvio para uma rotina que faz a carga do programa do sistema (sistema operacional). O endereço da área do disco onde se encontra o programa é determinado na carga do "kernel", e, na memória, o programa é colocado a partir da primeira posição disponível depois do interpretador. Se ocorre um erro na carga do sistema operacional, a seguinte

mensagem será mostrada no console (neste caso, a inicialização deve ser recomeçada):

SYSTEM LINE 0 SYSTEM LOAD ERROR.

7.2 Inicialização

As primeiras tarefas, antes da inicialização do "kernel" e do sistema, são:

- inicializar (zerar) a memória restante dentro dos primeiros 64 KB, que já havia sido testada durante o último IPL (a memória acima de 64 KB é testada e inicializada no carregador do bloco zero); e

- ligar (disparar) o relógio interno do computador, na freqüência de 50 Hz, o que deve provocar uma interrupção a cada 20 ms.

As demais tarefas executadas são a inicialização das classes que compõem o "kernel" e a saída de uma mensagem, no console, avisando que o "kernel" está pronto e que o processo inicial do programa do sistema começa a ser executado.

Usando a mesma notação com que são apresentadas as rotinas do "kernel" (ver capítulo 5), apresenta-se a seguir sua rotina de inicialização. As três primeiras rotinas chamadas são os procedimentos descritos neste capítulo.

Rotina:

```
VAR  KERNELLENGTH, INTERPRETERLENGTH: INTEGER;
BEGIN
  KERNELLENGTH := ...;
  INTERPRETERLENGTH := ...;
```

```
"AS TRES PRIMEIRAS ROTINAS SAO EXECUTADAS DIRETAMENTE
 ATRAVES DE UMA INSTRUCAO 'JUMP' NO ENDERECO ZERO."

LOADVIRTUALMACHINE;
LOADSYSTEMPROGRAM;
COREANDCLOCKINITIALIZE;

"INICIALIZA O KERNEL"

INIT
  NEWCORE,
  TIMER,CLOCK,
  CORE,VIRTUAL,
  RUNNING,READY,
  TAPE1600,CARTDISK,
  ALMDTERMINAL,SERIALPRINT,CARDREADER;

"INICIALIZA O PROCESSO INICIAL"

RUNNING.INITPARENT(KERNELLENGTH+INTERPRETERLENGTH);
KERNELREADY;
END.
```

8 CONCLUSÕES

Os objetivos do trabalho foram plenamente alcançados. O sistema está implementado já há algum tempo e sendo usado por outros pesquisadores para a realização de seus trabalhos. As expectativas quanto ao desempenho do sistema, com o uso do sistema operacional SOLO, foram, inclusive, superadas, pois o mesmo aproxima-se muito do desempenho obtido por Brinch Hansen com sua implementação no PDP-11/45. Na presente implementação, o sistema - "kernel" e interpretador - foi bastante otimizado em relação ao original, procurando obter um maior grau de eficiência, em termos de alocação de memória, tempo de interpretação e velocidade de execução dentro do "kernel".

Comparando a implementação no LABO com a de Brinch Hansen, foi verificado que o tempo de compilação do programa concorrente SOLO é, praticamente, o mesmo nos dois computadores: 2 min e 30 seg no PDP[1] e 2 min e 40 seg no LABO. Deve ser lembrado que o compilador Pascal Concorrente (bem como o Seqüencial) faz todo o processamento dos números presentes no programa fonte, utilizando variáveis e constantes do tipo REAL (ponto flutuante). Isto é uma desvantagem, na comparação, para a presente implementação.

A melhor implementação do sistema no LABO pode ser observada, ao verificar que a performance nos dois equipamentos é muito semelhante, embora o PDP seja mais adequado para a simulação da máquina virtual, possuindo muitas vantagens sobre o LABO, entre as quais estão:

- arquitetura mais próxima da máquina virtual (conjunto de registradores maior e de uso geral, processamento de "stack", conjunto de instruções mais poderoso e versátil, etc);

- processamento de ponto flutuante por "hardware";

- detecção de "overflow", por "hardware", nas operações aritméticas em complemento de 2;

- maior facilidade na troca de memória virtual (multiplexação de processos).

O "kernel" foi testado com o auxílio de um programa depurador, simulando as chamadas das primitivas e verificando os resultados obtidos em cada caminho percorrido dentro do "kernel". Para testar o interpretador e o "kernel" em conjunto, primeiro foi criado um disco com o sistema e, então, o próprio programa concorrente SOLO, e seus programas seqüenciais, foram utilizados juntamente com o depurador. Depois de testadas todas as rotinas, individualmente, poucos testes do sistema, como um todo, permitiram considerá-lo como funcional e, praticamente, correto, considerando a variedade dos programas utilizados nos testes.

Os dois programas, "kernel" e interpretador, ocupam, cada um, 5 KB de memória, aproximadamente. Estes tamanhos podem ser considerados como equiparáveis, sem desvantagens, aos tamanhos destes programas implementados no PDP (6 KB e 2 KB, respectivamente). Basta que se considere a maior dificuldade de programação, no LABO, de algumas estruturas da máquina virtual, principalmente, no que se refere ao processamento de pilhas, o que determinou esta diferença

tão significativa no tamanho do interpretador.

Outro elemento de comparação que pode ser utilizado, é um programa escrito em Pascal Seqüencial, do qual se possui os tempos de execução em outros equipamentos. O programa utilizado deve dispor 8 peças em um tabuleiro de xadrez, de tal forma que nenhuma das linhas, colunas e diagonais contenha mais de uma peça. A tabela 6 mostra os tempos de execução do programa, nos diversos computadores, incluindo a implementação no LABO. Alguns sistemas são compilativos e outros interpretativos. Uma comparação mais realística deve ser feita com os últimos e, dessa forma, pode ser verificado que o tempo de execução do programa, no LABO (Pascal), está bem abaixo dos tempos obtidos nos outros sistemas do mesmo tipo. Inclusive, alguns sistemas compilativos (Z80-PASCAL) não apresentam uma performance muito melhor (9,9 segundos) do que a presente implementação (12 segundos), a qual envolve interpretação. Por outro lado, o mesmo programa, reescrito na linguagem Business Basic, rodando sozinho no sistema operacional NIROS do LABO-8034, é executado em 165 segundos.

Assim, analisando-se os resultados obtidos, em termos de desempenho e utilidade, conclui-se que o sistema implementado é utilizável na prática, e que seu desenvolvimento e aplicação em minicomputadores é bastante conveniente.

O transporte de "software", da maneira realizada, é uma metodologia bastante adequada para a implementação de sistemas de programação deste tipo.

Sistema	Tempo(seg)	Sistema	Tempo(seg)
DIEHL DS2000	41 +	LSI 11 PASCAL	4 *
DIETZ 621 PASCAL E	68 #	DIETZ 621 FULL-PASCAL	2 *
GA 216/10	43 #	Z80-PASCAL	9,9 *
SIEMENS SCHULPASCAL	50 #	SIEMENS 330/6.620	5 *
LABO-8034 SOLO	12		

(*) com compilação
(#) com interpretação
(+) com compilação e interpretação

Tabela 6: Tempos de execução de um mesmo programa em diversos sistemas.

Como sugestões para a continuação deste trabalho, podem ser relacionadas a implementação de novos sistemas operacionais utilizando Pascal Concorrente, ou a modificação do sistema SOLO para transformá-lo em um sistema multi-usuário, e a implementação deste sistema de programação - Pascal Concorrente - em outros minicomputadores nacionais.

APÊNDICE A

Instruções da Máquina Virtual

Código		Instrução	Código		Instrução
1		CONSTADDR	2		LOCALADDR
3		GLOBALADDR	4		PUSHCONST
5		PUSHLOCAL	6		PUSHGLOBAL
7		PUSHIND	8		PUSHBYTE
9		PUSHREAL	10		PUSHSET
11		FIELD	12		INDEX
13	S	POINTER	14		VARIANT
15		RANGE	16		COPYBYTE
17		COPYWORD	18		COPYREAL
19		COPYSET	20		COPYTAG
21		COPYSTRUC	22	S	NEW
23	S	NEWINIT	24		NOT
25		ANDWORD	26		ANDSET
27		ORWORD	28		ORSET
29		NEGWORD	30		NEGREAL
31		ADDWORD	32		ADDREAL
33		SUBWORD	34		SUBREAL
35		SUBSET	36		MULWORD
37		MULREAL	38		DIVWORD
39		DIVREAL	40		MODWORD
41		BUILDSET	42		INSET
43		LSWORD	44		EQWORD
45		GRWORD	46		NLWORD

Código		Instrução	Código		Instrução
47		NEWORD	48		NGWORD
49		LSREAL	50		EQREAL
51		GRREAL	52		NLREAL
53		NEREAL	54		NGREAL
55		EQSET	56		NLSET
57		NESET	58		NGSET
59		LSSTRUCT	60		EQSTRUCT
61		GRSTRUCT	62		NLSTRUCT
63		NESTRUCT	64		NGSTRUCT
65		FUNCVALUE	66		JUMP
67		FALSEJUMP	68		CASEJUMP
69	S	INITVAR	70		CALL
71	S	CALLSYS	72		ENTER
73		EXIT	74	S	ENTERPROG
75	S	EXITPROG	76	C	BEGINCLASS
77	C	ENDCLASS	78	C	ENTERCLASS
79	C	EXITCLASS	80	CK	BEGINMON
81	CK	ENDMON	82	CK	ENTERMON
83	CK	EXITMON	84	C	BEGINPROC
85	CK	ENDPROC	86	C	ENTERPROC
87	C	EXITPROC	88		POP
89		NEWLINE	90		INCRWORD
91		DECRWORD	92	C	INITCLASS
93	C	INITMON	94	CK	INITPROC
95	C	PUSHLABEL	96	C	CALLPROG
97		TRUNCREAL	98		ABSWORD
99		ABSREAL	100		SUCCWORD

Código		Instrução	Código		Instrução
101		PREDWORD	102		CONVWORD
103	C	EMPTY	104	C	ATTRIBUTE
105	CK	REALTIME	106	CK	DELAY
107	CK	CONTINUE	108	CK	IO
109	C	START	110	CK	STOP
111	C	SETHEAP	112	CK	WAIT

As letras C, S e K são usadas, entre as colunas de código e nome, para marcar as instruções que:

- só são utilizadas pelo Pascal Concorrente (C);
- só são utilizadas pelo Pascal Seqüencial (S);
- envolvem comunicação com o "kernel" (K).

BIBLIOGRAFIA

1. BRINCH HANSEN, P. The architeture of concurrent programs. Englewood Cliffs, Prentice-Hall, 1977.

2. HARTMANN, Al C. A Concurrent Pascal compiler for minicomputers. Berlin, Springer-Verlag, 1977. (Lecture Notes in Computer Science, 50)

3. SEIDEL, H.A. & GREBE, G. A kernel for a Concurrent Pascal operating system. In: INTERNATIONAL CONFERENCE on OPERATING SYSTEMS, 2. Rocquencourt, IRIA, 1978.

4. HAMMERL, L.; LÖHR, K.-P. & PÜTZ, J. Concurrent Pascal in virtual machines. In: WIPPERMANN, H.-W. TAGUNG in KAISERSLAUTERN,2., Feb.16-17, 1979. Pascal. Stuttgart, B. G. Teubner, 1979. p.109-21.

5. HOLT, R.C. et alii. Structured concurrent programming with operating systems applications. Reading, Addison-Wesley, 1978.

6. BRINCH HANSEN, P. A programming methodology for operating system design. In: INFORMATION PROCESSING 74. S. l., North-Holland, 1974.

7. HOARE, C.A.R. Monitors: an operating system structuring concept. Communications of the ACM, New York, 17(10): 549-57, 1974.

8. MEDEIROS, Gil Carlos R. _Características funcionais do computador LABO-8034._ Porto Alegre, Pós-Graduação em Ciência da Computação da UFRGS, 1980.

9. NIXDORF COMPUTER. _Data processing system 8870/1 - system description._ Dec. 1977.

10. NIXDORF COMPUTER. _System 8870 - field engineering reference manual._ Sept. 1978.

11. BELL, J.R. Threaded code. _Communications of the ACM,_ New York, _16_(6):370-72, 1973.

12. KRÖNIG, D. & HOFFMANN, H.-J. Eine Zusammenstellung von Eigenschaften der bei der Tagung vorgestellten PASCAL-Implementierungen. In: WIPPERMANN, H.-W. TAGUNG in KAISERSLAUTERN,2., Feb.16-17, 1979. _Pascal._ Stuttgart, B. G. Teubner, 1979. p.185-201.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Pós-Graduação em Ciência da Computação

Implementação do Sistema

Pascal Concorrente no

Computador LABO-8034

TESE APRESENTADA AOS SRS.

Visto e permitido a impressão

Porto Alegre,..../...../.....

Coordenador do Curso de Pós-Graduação
em Ciência da Computação