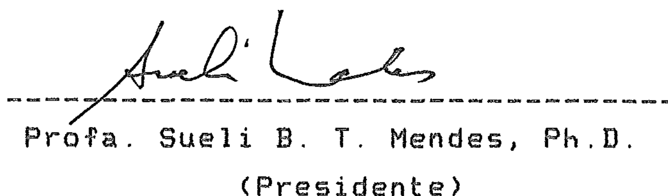
PORTOS-TF:
SISTEMA OPERACIONAL PORTÁTIL DE TEMPO-REAL COM PRIMITIVAS DE TOLERÂNCIA A FALHAS

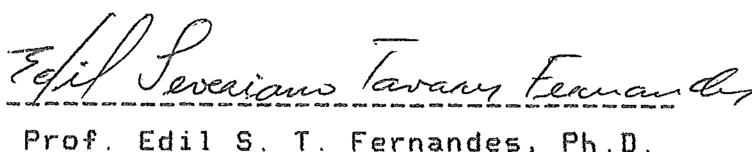
Wanderley Lobianco Júnior.

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

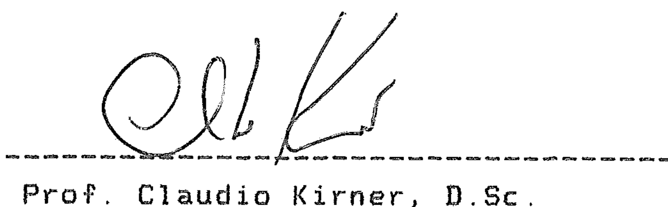
Aprovada por:



Profa. Sueli B. T. Mendes, Ph.D.
(Presidente)



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



Prof. Claudio Kirner, D.Sc.

RIO DE JANEIRO, RJ - BRASIL
JUNHO DE 1989

LOBIANCO JÚNIOR, WANDERLEY

PORTOS-TF: Sistema Operacional Portátil de Tempo-Real com Primitivas de Tolerância a Falhas [Rio de Janeiro] 1989.

xv, 205 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 1989)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Sistemas Operacionais I. COPPE/UFRJ II. Título (série).

a minha noiva (e futura esposa), que sempre me motivou e apoiou no mestrado e na vida;

a minha mãe, primeira e eterna professora;

a meu pai e a meu irmão, meus maiores amigos;

a meus avós, de quem sempre sentirei saudades e muito carinho;

a Deus, pois sem Ele nada seria possível.

AGRADECIMENTOS:

À Profa. Sueli B. T. Mendes, pela forma amiga com que me orientou neste trabalho.

Aos professores Edil S. T. Fernandes e Cláudio Kirner, pela honra que me dão ao integrarem minha banca.

Ao Prof. Wilson V. Ruggiero, pela gentileza de ter me ajudado no início dos meus trabalhos de tese com orientações muito importantes.

A meus amigos Luiz Carlos de Souza Jr., Fernando C. L. Vilela de Abreu e Ronaldo P. Thompson, com quem muito aprendi e tive o prazer de desenvolver meu primeiro trabalho na área de sistemas operacionais.

A meus amigos Roberto L. C. Ramos, Carlos Eduardo W. Ratto e Sérgio C. Andrade, que contribuiram com idéias e opiniões valiosas para a elaboração desta tese.

A minha noiva e a minha mãe, pelo paciente trabalho de elaboração das figuras.

A Mário Roberto, Maria Helena, Mário e Ivone Benevides e Capello Ivo e Wanda Folhadella, pelo amor e carinho que sempre me deram.

A Sônia Alves de Freitas, pela atenção com que me orientou e ajudou nos aspectos formais da tese.

A meus professores da COPPE, que me ajudaram a adquirir novos conhecimentos.

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

PORTOS-TF:
SISTEMA OPERACIONAL PORTÁTIL DE TEMPO-REAL
COM PRIMITIVAS DE TOLERÂNCIA A FALHAS

Wanderley Lobianco Júnior

junho, 1989

Orientadora: Sueli B. T. Mendes

Programa: Engenharia de Sistemas e Computação

Este trabalho aborda a descrição, projeto e implementação de um sistema operacional portátil, voltado para aplicações em tempo-real e com primitivas de tolerância a falhas: o PORTOS-TF. São discutidos aspectos relativos a sistemas operacionais, portabilidade, tempo-real e tolerância a falhas. Após a apresentação de questões teóricas encontradas na literatura, são comentadas as opções de projeto adotadas.

O PORTOS-TF apresenta um núcleo multitarefa que permite a execução de processos concorrentes com prioridades e tamanhos de fatias de tempo distintos. A fim de tornar o sistema aplicativo mais versátil e modular, a comunicação e sincronização entre processos se dá através de troca de mensagens com endereçamento baseado em portos. Para permitir sua aplicação em ambientes de alta

confiabilidade, o PORTOS-TF incorpora primitivas de tolerância a falhas que criam um mecanismo de N-versões.

Constituem-se nas maiores contribuições desta tese a reunião, em um único texto, de técnicas e conceitos relativos à tolerância a falhas, em especial a falhas de software, e a discussão das questões ligadas à implementação de um sistema operacional portátil de tempo-real com primitivas de tolerância a falhas.

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

PORTOS-TF:
PORTABLE REAL-TIME OPERATING SYSTEM WITH FAULT-TOLERANCE PRIMITIVES

Wanderley Lobianco Júnior.

June, 1989

Advisor: Sueli B. T. Mendes

Department: Systems Engineering and Computing

This work describes the design and implementation of an operating system for real-time applications with fault-tolerance primitives: PORTOS-TF. Aspects related to operating systems, portability, real-time constraints and fault-tolerance are discussed. The adopted choices related to the operating system design are presented after theoretical considerations.

PORTOS-TF creates a multitask kernel which allows the execution of concurrent processes with different priorities and time-slices. It implements a message switching mechanism for communication and synchronisation between processes. Addressing politics are based on ports. Fault-tolerance primitives support a N-version mechanism for highly reliable applications.

The great contributions provided by this thesis consist of a study of different fault-tolerance concepts and technics (mainly related to software) in the same text and the discussion of implementation aspects in a portable real-time operating system with fault-tolerance primitives.

ÍNDICE:

CAPÍTULO I : INTRODUÇÃO 1
I.1. Motivação 1
I.1.1. Razões para ser Portátil 2
I.1.2. Razões para ser Tolerante a Falhas 4
I.2. Objetivos 5
I.3. Metodologia 6
I.4. Características Gerais 7
I.5. Estrutura de Capítulos 10

CAPÍTULO II : PORTABILIDADE E REUTILIZAÇÃO DE SOFTWARE 13
II.1. Introdução 13
II.2. Critérios de Portabilidade 14
II.3. Questões Relativas à Portabilidade do Sistema Operacional 15
II.3.1. Características da Máquina Padrão 16
II.4. Procedimentos de Transporte do PORTOS-TF 18

CAPÍTULO III : NÚCLEO DO SISTEMA OPERACIONAL 21
III.1. Introdução 21
III.2. Estruturas de Dados do Núcleo 24
III.2.1. Lista de Descritores de Processos 24
III.2.2. Lista de Processos no Estado "Pronto" 25
III.2.3. Lista de Processos Bloqueados com Cláusula de Tempo 26
III.3. Instalação e Acesso ao Núcleo 28
III.4. Características dos processos do PORTOS-TF 29
III.5. Funções do Núcleo 30
III.5.1. Criação de Processos 30
III.5.2. Destruição de Processos 34
III.5.3. Obtenção do Identificador 34
III.5.4. Bloqueio de Processos 35
III.5.5. Liberação de Processos 36
III.5.6. Obtenção da Prioridade 37

III.5.7. Alteração da Prioridade 37
III.5.8. Obtenção da Fatia de Tempo 37
III.5.9. Alteração da Fatia de Tempo 38
III.5.10. Congelamento da Fatia de Tempo 38
III.5.11. Descongelamento da Fatia de Tempo 38
III.5.12. Habilitação de Geração de Exceções 38
III.5.13. Desabilitação de Geração de Exceções 39
III.5.14. Habilitação de Escalonamento 39
III.5.15. Desabilitação de Escalonamento 39
III.5.16. Escalonamento do Processo Corrente 40
III.5.17. Obtenção do Estado do Sistema Operacional 40

CAPÍTULO IV : COMUNICAÇÃO E SINCRONIZAÇÃO ENTRE PROCESSOS 41
IV.1. Introdução 41
IV.2. Mecanismos de Comunicação e Sincronização entre Processos 41
IV.2.1. Endereçamento em Sistemas Distribuídos 43
IV.2.2. Mecanismos para Sistemas com Memória Distribuída 44
IV.2.2.1. Troca de Mensagens 45
IV.2.2.2. "Rendez-Vous" 49
IV.2.2.3. Chamada Remota de Procedimentos 50
IV.2.3. Portos 51
IV.3. Configuração Dinâmica de Sistemas 56
IV.4. Implementação de um Mecanismo de Comunicação e Sincronização entre Processos Baseado em Portos 57
IV.4.1. Primitivas para o Manuseio de Portos 58
IV.4.2. Primitivas de Comunicação e Sincronização 59
IV.4.2.1. Envia 60
IV.4.2.2. Recebe 61
IV.4.2.3. Selec 62
IV.4.2.4. Fim_Selec 65
IV.4.2.5. Porto_Ultima_Mensagem 65

IV.4.2.6. Responde 66

CAPÍTULO V: GERÊNCIA DE MEMÓRIA 67
V.1. Introdução 67
V.2. Opções de Projeto 68
V.3. Aspectos da Implementação 70

CAPÍTULO VI : ESTRUTURAS DE DADOS 74
VI.1. Introdução 74
VI.2. Opções de Projeto 78
VI.2.1. "Listas" do Sistema Operacional 79
VI.2.2. O Descritor de Processos 82
VI.3. Aspectos da Implementação 84
VI.3.1. Algoritmo de Balanceamento de Árvore Binária 89

CAPÍTULO VII : TOLERÂNCIA A FALHAS 94
VII.1. Introdução 94
VII.2. Terminologia 96
VII.2.1. Termos Gerais 96
VII.2.2. Termos Específicos 98
VII.3. Classificação das Falhas 101
VII.4. Tipos de Tratamentos de Faltas, Erros e de Falhas 103
VII.5. Princípios de Sistemas Tolerantes a Falhas ... 104
VII.6. Técnicas de Tolerância a Falhas 107
VII.6.1. Técnicas de Redundância Temporal 108
VII.6.1.1. Ações Atômicas 108
VII.6.1.2. Exceções 110
VII.6.1.3. Recuperação de Erros 112
VII.6.1.3.1. "Backward Error Recovery" 113
VII.6.1.3.1.1. Técnicas de Redundância Passiva 113
VII.6.1.3.1.1.1. Suporte para Mecanismos de "Hot Stand-by" 116
VII.6.1.3.1.2. Blocos de Recuperação ("Recovery Blocks") 123

VII.6.1.3.1.2.1. As Alternativas 127
VII.6.1.3.1.2.2. O Teste de Aceitação 128
VII.6.1.3.1.2.3. Restauração do Estado Inicial 130
VII.6.1.3.1.2.4. Recuperação de Erros em Processos Comunicantes ... 131
VII.6.1.3.2. "Forward Error Recovery" 136
VII.6.1.3.3. "Backward Error Recovery", "Forward Error Recovery" e Exceções 138
VII.6.2. Técnicas de Redundância de Recursos 140
VII.6.2.1. Os Módulos 142
VII.6.2.2. Os Votadores 144
VII.6.2.3. Primitivas do Sistema Operacional 147
VII.6.3. Comparação entre Redundância Temporal e Redundância de Recursos 148
VII.7. Implementação de um Mecanismo de N-Versões ... 150

CAPÍTULO VIII : CONCLUSÕES 154

REFERÊNCIAS BIBLIOGRÁFICAS 157

APÊNDICE A : IMPLEMENTAÇÃO EM EQUIPAMENTOS DO TIPO IBM-PC 165
A.1. Introdução 165
A.2. Operação em Conjunto com o DOS 166
A.3. Aspectos da Implementação 168
A.3.1. O Relógio de Tempo-Real 168
A.3.2. Troca de Contexto 169
A.3.3. Uso de Ponteiros e Endereços 170
A.3.4. O "Buffer Pool" do Sistema 171
A.3.5. Escrita em Vídeo 172

APÊNDICE B : PRIMITIVAS DO NÚCLEO DO PORTOS-TF 174
B.1. Explicação dos Tipos não-padrões Empregados 174
B.2. Protótipos da Primitivas do Núcleo 177

APÊNDICE C : COMENTÁRIOS SOBRE AS REFERÊNCIAS BIBLIOGRÁFICAS 189
C.1. Textos Relativos a Sistemas Operacionais 189
C.2. Textos Ligados a Tolerância a Falhas 196
C.3. Outras Referências 204

ÍNDICE DE FIGURAS:

CAPÍTULO III

III.1. Esquema de um nó da Lista de DPs 25

III.2. Esquema de um nó da Lista dos Prontos ou da Lista dos Bloqueados 26

CAPÍTULO IV

IV.1. Esquema de um mecanismo de troca de mensagens 45

IV.2. Esquema de difusão de mensagens 47

IV.3. Esquema de recepção de mensagem de qualquer remetente 48

IV.4. Tipos de portos e possibilidades de ligação ... 54

IV.5. Opções de ligação de portos no PORTOS-TF 58

IV.6. Exemplo de estrutura de seleção baseada em construções de linguagem 64

IV.7. Exemplo de estrutura de seleção no PORTOS-TF .. 64

CAPÍTULO V

V.1. Exemplo de alocação de áreas de memória 69

V.2. Esquema de um elemento do "buffer pool" 71

V.3. Esquema de desmembramento de bloco de memória .. 71

V.4. Esquema de fusão de blocos de memória 72

CAPÍTULO VI

VI.1. Exemplo de tabela 74

VI.2. Exemplo de lista encadeada 75

VI.3. Exemplo de árvore 77

VI.4. Exemplo de árvore binária 78

VI.5. Estrutura de um nó das árvores empregadas no PORTOS-TF 80

VI.6. Estrutura geral das "listas" do PORTOS-TF 81

VI.7. Esquema de um nó de "lista" do PORTOS-TF com apenas um elemento 85
VI.8. Possibilidades de substituição de raiz de árvore binária 88
VI.9. Esquema de uma árvore binária desbalanceada à direita 91
VI.10. Opções de balanceamento da árvore da figura VI.9 92

CAPÍTULO VII

VII.1. Componente ideal tolerante a falhas 111
VII.2. Exemplo de utilização de pontos de checagem .. 119
VII.3. Sintaxe de um bloco de recuperação 124
VII.4. Exemplo de um bloco de recuperação simples ... 125
VII.5. Bloco de recuperação mais complexo 126
VII.6. Um bloco de recuperação com alternativas que chegam a resultados diferentes mas, ainda assim, aceitáveis, apesar de menos desejáveis 128
VII.7. Um programa de ordenação tolerante a falhas .. 129
VII.8. Estrutura ilustrativa do "efeito dominó" 133
VII.9. Transações entre processos através de conversações 135
VII.10. Procedimento de emergência para aviões leves 138
VII.11. Implementação de bloco de recuperação através de gerenciamento de exceções 139
VII.12. Esquema de redundância modular tripla 141

CAPÍTULO I

INTRODUÇÃO

"Um Sistema Operacional pode ser encarado como um conjunto de programas, implementados por software ou firmware, que tornam o hardware utilizável. O hardware provê o 'potencial bruto de computação'. Os sistemas operacionais tornam esse potencial convenientemente disponível aos usuários." É assim que DEITEL (1984) define sistemas operacionais. A literatura sobre essa área da Ciência da Computação é vasta (BRINCH HANSEN, 1973; MADNICK & DONOVAN, 1974; TSICHRITZIS & BERNSTEIN, 1974; HOLT & outros, 1978; DEITEL, 1984; KIRNER & MENDES, 1988; etc.).

I.1. Motivação:

Sistemas operacionais de tempo-real não são um assunto novo. Vários trabalhos já foram escritos a seu respeito (FERREIRA, 1985; SANTOS, 1987; SCHWARZ, 1981). Todavia, existe uma peculiaridade nesse tipo de sistemas operacionais que os difere dos demais: tempo de resposta.

Em sistemas de tempo-real, solicitações do mundo exterior devem ser atendidas em um tempo finito e bem determinado. Isso faz com que seja desejável o conhecimento detalhado do sistema operacional.

Na área de microcomputadores não são muitos os sistemas operacionais de tempo-real comerciais. Pode-se citar o QNX (da Quantum Software Systems Ltd., Canadá) (1988) e o SC (da FDTE/USP) (SANTOS, 1987). Esses sistemas operacionais, contudo, não são portáteis para equipamentos dedicados (do tipo unidade remota de processamento).

Um sistema distribuído de tempo-real é composto de várias estações (computadores) interligadas por um sub-sistema de comunicação. As estações que compõem esse sistema distribuído não são necessáriamente iguais. De um modo geral, sistemas distribuídos são compostos de vários tipos de equipamentos. A escolha do tipo de estação é feita tomando-se por base as funções por ela desempenhada. Como a portabilidade do software aplicativo de tempo real é desejável, deve-se ter o mesmo sistema operacional no maior número de estações possíveis.

Um outro fator muito importante em sistemas de tempo-real é a capacidade de continuarem em operação após a ocorrência de uma falha, mesmo que de forma degradada.

A motivação desse trabalho é, portanto, a elaboração de um sistema operacional portátil de tempo-real com primitivas de tolerância a falhas, particularmente falhas de software.

I.1.1. Razões para ser Portátil:

Software vem se tornando, com o passar dos anos, o componente mais caro de um sistema de computação. Com o avanço na área de circuitos integrados, os equipamentos projetados têm sido mais poderosos e flexíveis, podendo ser utilizados em uma vasta gama de aplicações. O hardware provê a infraestrutura sobre a qual se apoia a aplicação. Cabe, pois, ao software "vestir" este hardware de modo a construir um sistema de computação capaz de desempenhar sua função da melhor forma possível.

Quando se desenvolve um software para uma determinada aplicação, deve-se preocupar em fazê-lo de modo a que, senão todo, pelo menos parte deste software possa ser reaproveitado em uma nova situação. Isso se deve,

ainda em boa parte, pela forma "artesanal" com que se faz software. Como forma de tornar o desenvolvimento de software uma atividade disciplinada, visando minimizar os esforços para sua concepção, implementação, manutenção e alteração, surgiu um ramo na Ciência da Computação denominado Engenharia de Software. Tratar software como um trabalho de engenharia implica na adoção de técnicas e procedimentos formais que levam a implementações mais confiáveis em menor tempo.

Contudo, a adoção de técnicas de engenharia de software não resolve o problema de reaproveitamento do software. Por mais bem regrado que seja seu desenvolvimento, ainda existe a questão da diversidade de ambientes operacionais nos quais o software é executado. Esse ambiente é composto por vários fatores como tipo e número dos processadores, arquitetura da máquina, sistema operacional, linguagem de programação, etc. Diante dessa situação, muitas vezes a única parte de um software que pode ser reaproveitada é sua idéia.

"Portabilidade" é a palavra chave dessa questão. Um software deve ter um grau de portabilidade o maior possível. Um software totalmente portátil é aquele que não necessita de nenhuma modificação para ser executado em um ambiente operacional diverso daquele no qual foi originalmente desenvolvido.

Na área de aplicações em tempo-real (como a área de controle de processos), é comum a existência de vários equipamentos de processamento de dados, como por exemplo CPs (controladores programáveis), CNCs (comandos numéricos computadorizados), unidades remotas de processamento (que são computadores dedicados), além de microcomputadores, minicomputadores e até mesmo computadores de grande porte de propósito genérico. Como um sistema de tempo-real engloba, em muitos casos, mais de

um desses componentes, torna-se necessário o desenvolvimento de software para mais de um tipo de máquina hospedeira.

Ocorre, porém, que ter que considerar, no desenvolvimento do software aplicativo, as particularidades de cada uma das estações que compõem esse sistema distribuído, é uma restrição extremamente desconfortável para o projetista. Dependendo da configuração do sistema, algumas tarefas podem ser executadas em um ou outro tipo de estação.

A existência de um sistema operacional que seja comum aos diversos tipos de computadores empregados aumenta a portabilidade do software aplicativo. Em alguns casos, o aplicativo pode ser totalmente portátil ou necessitar apenas de ser recompilado para ser expresso em termos do código do novo processador.

Todavia, como o aplicativo necessita do sistema operacional, este também deve ser o mais portátil possível. Caso contrário, as dificuldades encontradas para transportar o sistema operacional para uma nova máquina terão reflexos na portabilidade do sistema de tempo-real como um todo.

I.1.2. Razões para ser Tolerante a Falhas:

Sistemas de tempo-real são encontrados em diversos setores da atividade humana. São, em sua grande maioria, sistemas de controle. Esses sistemas têm aplicação nas áreas de manufatura, geração de energia, militar, médica, aero-espacial, etc.

Por mais bem especificados e testados que sejam, não há como garantir que um sistema de controle em tempo-real seja totalmente confiável, isto é, que esteja

sempre em atividade e se comportando de acordo com suas especificações. Defeitos em componentes, bem como erros de projeto e de implementação são sempre possíveis de ocorrer.

Uma falha em um sistema de controle pode ter conseqüências de dimensões variáveis, de acordo com o tipo de sistema sobre o qual atua. Uma suspensão no bombeamento automático de água para um reservatório pode causar apenas o desconforto de ter-se que realizar essa operação manualmente. Por outro lado, uma falha em um sistema de controle de uma nave espacial pode significar o desperdício de uma grande soma de dinheiro, sem ser levada em consideração a perda de vidas humanas, caso o vôo seja tripulado.

Uma falha pode interromper o funcionamento de um sistema de controle. Apesar de indesejável, essa condição é muitas vezes preferível ao mau funcionamento do sistema.

De qualquer forma, independentemente do tipo de falha, de sua conseqüência e gravidade, é importante que um sistema de tempo-real disponha de algum mecanismo que impeça que os efeitos do mau funcionamento de uma das suas partes (seja de software ou hardware) seja sentido pelo sistema como um todo. É igualmente relevante que o sistema operacional disponha de primitivas que permitam a implementação de mecanismos próprios de tolerância a falhas para que a aplicação não tenha que se preocupar em criá-los.

I.2. Objetivos:

Como já mencionado anteriormente, o objetivo geral desse trabalho é o de elaborar, implementar e discutir um sistema operacional, voltado para um ambiente

distribuído, com primitivas de tolerância a falhas e estruturado de forma a ser o mais portátil possível. A atenção maior dessa tese, contudo, está nos mecanismos de tolerância a falhas, em especial falhas de software.

Visa-se, principalmente, estruturar o sistema operacional de forma a ser o mais portátil possível e estudar e discutir conceitos e mecanismos de tolerância a falhas, implementando, por fim, um desses mecanismos. O mecanismo de tolerância a falhas escolhido foi o de N-versões. A implementação desse sistema operacional em um microcomputador do tipo IBM-PC ilustra aplicações desses conceitos.

Dada à portabilidade do sistema operacional, os resultados obtidos podem ser extrapolados ou mesmo verificados em outros equipamentos com capacidade de processamento semelhante à do empregado nessa tese. A portabilidade para máquinas de maior porte é também um objetivo perseguido, mas que será deixado para trabalhos posteriores.

I.3. Metodologia:

A metodologia de trabalho empregada consistiu em efetuar um estudo global sobre sistemas operacionais, sistemas de tempo-real e confiabilidade de sistemas.

A partir desse estudo foram elaboradas propostas para a implementação do sistema operacional, de acordo com os objetivos expostos no ítem anterior. Foram definidos os "limites" da tese, ou seja, quais tópicos seriam abordados e quais não. Esse é um momento importante pois uma definição mal elaborada do contexto estudado pode dificultar um acabamento adequado do

trabalho.

A implementação foi feita de forma gradual e modular. Cada módulo foi idealizado, projetado, implementado e depurado até apresentar um comportamento com um certo grau de confiabilidade. Mesmo não garantindo a eliminação total dos erros, esse tipo de procedimento facilita o seu isolamento permitindo, dessa forma, diminuir o tempo de implementação.

A documentação da tese foi sendo gerada em paralelo com a implementação. Alguns capítulos chegaram a ser escritos antes mesmo da implementação. Isso permitiu que o texto elaborado fosse o mais fiel possível ao trabalho prático.

Por fim, foram relacionadas algumas críticas e sugestões. Essa parte visa salientar os pontos fortes e fracos do trabalho, para que possam servir como experiência prévia para trabalhos futuros. São, ainda, sugeridos temas que possam dar continuação a essa tese.

I.4. Características Gerais:

O sistema operacional projetado e implementado nesse trabalho foi batizado com o nome PORTOS-TF. Ele tem as seguintes características:

. multitarefa;
. monoprocessado;
. voltado para um ambiente distribuído;
. com mecanismos de comunicação e sincronização entre processos baseados em troca de mensagens.
. endereçamento por portos.
. escrito em C;
. dividido em módulos dependentes e independentes do hardware empregado e

. com mecanismo de N-versões para tolerância a falhas de software.

A capacidade multitarefa se faz necessária pela própria característica de paralelismo entre eventos existente na área de controle em tempo-real.

Dada à descentralização do sistema como um todo, não se faz necessário o emprego de estações com grande capacidade de processamento. Dessa forma, computadores com um único processador atendem muito bem às necessidades do sistema.

A escolha de mecanismos de comunicação e sincronização entre processos baseados em troca de mensagens é a opção mais simples e natural para um sistema distribuído. Mecanismos de chamada remota de procedimentos são mais adequados a nível de linguagens e não de sistema operacional. Técnicas como semáforos ou monitores foram descartadas por necessitarem de memória compartilhada.

O endereçamento para comunicação via portos tem como objetivo permitir a configuração dinâmica dos processos do sistema. Essa facilidade é de grande importância para as primitivas de tolerância a falhas implementados.

A linguagem C foi escolhida para implementar o PORTOS-TF por ser uma linguagem bastante difundida e bem padronizada. Isso torna os programas escritos nessa linguagem quase que totalmente portáteis, isto é, independentes do compilador usado.

A estruturação dos programas fonte em dependentes e independentes do hardware utilizado permite transportar o sistema operacional com um mínimo de esforço. Desde que haja compiladores "C" para o processador destino

e uma forma de carregar esse código na memória do computador, basta recompilar os módulos independentes do hardware. Os módulos dependentes, contudo deverão ser reescritos pois estão vinculados à estrutura de memória, número e tipo dos registradores, etc.

A existência de um mecanismo de tolerância a falhas implementado pelo sistema operacional é uma característica muito desejável em sistemas de controle. Dessa forma, o software aplicativo não necessita de se preocupar explicitamente com a elaboração desses mecanismos.

O PORTOS-TF não provê nenhum sistema de gerenciamento de arquivos. Ele assume que todo o software, básico e aplicativo, está carregado na memória primária da estação. Essa opção de projeto, apesar de à primeira vista parecer comprometer a aplicabilidade do sistema operacional, não o torna desinteressante, como pode ser visto a seguir.

Existem duas classes de máquinas onde o PORTOS-TF pode ser empregado: máquinas de uso geral e máquinas dedicadas. Nas máquinas de uso geral, isto é, máquinas comerciais usadas para uma grande variedade de aplicações, existe sempre um sistema operacional dito "nativo" que é utilizado para prover um ambiente de propósito genérico. Esses sistemas operacionais nativos apresentam, de um modo geral, um sistema de gerenciamento de arquivos. Para esse tipo de computador, é conveniente utilizar-se o PORTOS-TF em conjunto com o sistema operacional nativo, desde que, é claro, não haja conflitos irremediáveis entre os dois sistemas operacionais. Com isso, conjugam-se as funções do PORTOS-TF com a do sistema operacional nativo (SOUZA, 1987). Uma das vantagens desse tipo de associação é o uso de softwares desenvolvidos para o sistema operacional nativo como sendo um dos processos

gerenciados pelo PORTOS-TF. O sistema de gerenciamento de arquivos, nessa situação é, então, provido pelo sistema operacional nativo. Em alguns sistemas operacionais nativos poderá ser necessária alguma alteração no sistema de gerenciamento de arquivos a fim de adaptá-los a um ambiente concorrente e de tempo-real. Como tais modificações são muito dependentes da máquina e do sistema de gerenciamento de arquivos original, elas não foram consideradas nesse trabalho.

Em computadores dedicados, que não possuem sistema operacional nativo, é comum também não se encontrar dispositivo algum de memória de massa. Nessas máquinas, o software fica todo gravado em memórias não voláteis (do tipo EPROM) ou então é carregado remotamente de um computador com esse tipo de dispositivo. Nos casos, contudo, em que esses computadores possuam dispositivos de memória secundária e desejem utilizá-la, deve-se desenvolver um novo módulo, dependente da arquitetura da máquina e do dispositivo, a fim de suprir os mecanismos de gerenciamento de arquivos. Esse módulo pode se constituir de um processo para o sistema operacional ou então ser encarado como um controlador de dispositivo ("device driver"). A primeira opção tem a vantagem de lidar mais facilmente com os assincronismos do dispositivo físico sem ocupar a UCP na tarefa de aguardar o atendimento de um pedido.

I.5. Estrutura de Capítulos:

Neste capítulo I, é feita uma introdução aos temas estudados na tese: sistemas operacionais de tempo-real, portabilidade e tolerância a falhas. São apresentadas as razões que levaram à elaboração de um trabalho nessa área, o que se pretende com ele, de que forma o problema foi abordado e que características tem sua

implementação.

No capítulo II, são estudadas questões gerais sobre "portabilidade" de sistemas, bem como aspectos particulares relativos a sistemas operacionais.

O capítulo III é dedicado ao núcleo do sistema operacional. Nele, são apresentados, de forma resumida, alguns conceitos básicos na área de sistemas operacionais. Em seguida são relacionadas as opções de projeto com a exposição dos motivos que as levaram a serem adotadas. As estruturas de dados empregadas na construção do núcleo são descritas com o propósito de auxiliar no entendimento da lógica de suas funções. Essas funções são listadas no último ítem do capítulo.

O capítulo IV é reservado para um tema bastante estudado na área de sistemas operacionais: mecanismos de comunicação e sincronização entre processos. É dada uma visão geral sobre os mecanismos existentes e onde mais se aplicam. Em seguida são mostradas as opções de projeto feitas e suas justificativas. Por último, as primitivas de comunicação e sincronização são comentadas.

No capítulo V, é discutida a idéia e a implementação de um "pool" de memória com o objetivo de permitir ao sistema operacional alocar dinamicamente suas estruturas.

O capítulo VI discute as estruturas de dados que são usadas no PORTOS-TF. Esse assunto mereceu um capítulo à parte por constituir-se em um tópico de grande importância na eficiência do sistema operacional.

O capítulo VII fala sobre o que pretende ser a maior contribuição desse trabalho: um estudo de técnicas de tolerância a falhas, em especial falhas de projeto de

software. Nesse capítulo serão formuladas e comentadas primitivas que implementam um mecanismo de N-versões para tolerância a falhas.

O capítulo VIII traz as conclusões dessa tese, resumindo o que foi implementado e sugerindo alguns tópicos que podem ser abordados em trabalhos futuros, pois o tema abordado é vasto e está longe de se esgotar.

Terminado o texto principal, apresenta-se a bibliografia utilizada na elaboração desse trabalho. Associado a cada título, vem um breve comentário sobre o que é abordado pelo texto. Pretende-se com isso auxiliar aos que se utilizarão da tese como referência durante a seleção de textos para maior aprofundamento em algum tópico. Algumas das referências bibliográficas, mesmo não tendo contribuído diretamente para a elaboração da tese, são relacionados com o objetivo de servir de referência para o detalhamento maior em alguma área específica.

O apêndice A discute as particularidades da implementação do PORTOS-TF em um computador do tipo IBM-PC. Nessa implementação é comentada a forma como o PORTOS-TF utiliza o DOS como sistema operacional nativo da estação.

Por fim, são descritas, no apêndice B, os protótipos das primitivas do PORTOS-TF.

CAPÍTULO II

PORTABILIDADE E REUTILIZAÇÃO DE SOFTWARE

II.1. Introdução:

Portabilidade é uma característica desejável em qualquer software. Isso se deve ao fato de que, cada vez mais, o custo do software tem aumentado em relação ao do hardware.

Um software deve ser projetado de forma a que possa ser reaproveitado. Quando se concebe um programa, é comum existirem algumas funções cujo uso não se restringe à aplicação para a qual se destinam. Em muitos casos, porém, a forma com que são implementadas faz com que seja impossível seu reaproveitamento em outros programas, pelo menos sem nenhum tipo de modificação. Do mesmo modo, em sistemas multitarefa e distribuídos, a implementação despreocupada de um módulo potencialmente reutilizável pode inviabilizar sua aplicação em outros sistemas.

A generalidade deve, pois, ser buscada. Essa busca, porém, pode levar à ineficiência do módulo. Existe, então, como em qualquer problema de engenharia, uma relação custo X benefício, que deve ser muito bem considerada.

O que foi discutido anteriormente se refere ao reaproveitamento de módulos de software para aplicações diferentes. Existe, contudo, a situação na qual a aplicação é a mesma. O que muda é o ambiente computacional (computador, sistema operacional, etc.). O problema então está em conseguir executar o mesmo software em ambientes computacionais diversos.

Diferenças como o processador utilizado, o tamanho da memória e o sistema operacional (sistema de arquivos e funções disponíveis) podem fazer com que uma mesma aplicação tenha que ser toda reescrita. O uso de linguagens de alto nível reduz a dependência do processador na medida em que se disponha de compiladores para os processadores empregados. Uma outra forma de facilitar o transporte de um programa de um ambiente computacional para outro está na padronização do sistema operacional.

Sistemas operacionais que seguem um mesmo padrão apresentam suas funções e interfaces de acordo com o especificado nesse padrão. Suas implementações, contudo, podem ser completamente independentes e distintas.

Ocorre que um sistema operacional escrito para um tipo de computador também deve ser portátil na medida em que se deseje servir também para outros tipos de equipamentos. É nesse conceito que se coloca o PORTOS-TF.

II.2. Critérios de Portabilidade:

CHERITON & outros (1979) dizem que "um programa é portátil para um conjunto de máquinas caso custe significativamente menos modificá-lo para cada máquina do que implementá-lo e mantê-lo separadamente". Esse custo inclui, entre outros fatores, o desempenho do programa. Assim sendo, um programa fácil de ser convertido para um novo computador mas que se mostre extremamente ineficiente nesse novo hardware não pode ser considerado portátil.

Um software que não necessita de nenhuma alteração para ser transportado de um computador para outras máquinas é dito "independente da máquina".

Um software portátil é vantajoso sob o ponto

de vista de seu desenvolvimento, manutenção e evolução. É normalmente mais econômico desenvolver-se um software para vários computadores do que confeccionar um para cada tipo de máquina. A maior área de aplicação do programa justifica sua melhor documentação e um projeto mais bem elaborado. Por fim, é também mais fácil a manutenção e evolução de um programa bem projetado e documentado do que muitos que, invariavelmente, não terão a mesma qualidade desse único.

II.3. Questões Relativas à Portabilidade do Sistema Operacional:

A primeira questão que surge quando se discute a portabilidade de um sistema operacional é a definição das máquinas para as quais ele é portátil. Um sistema operacional tem que se basear na existência de um conjunto de estruturas providas pela arquitetura da máquina de modo a construir sobre ela sua própria arquitetura. Isso faz com que tenha que ser definida uma máquina padrão, que é uma abstração das máquinas reais que se pretende atender.

Um outro ponto a ser discutido é a forma de se estruturar o sistema operacional a fim de minimizar o trabalho de transporte entre máquinas diferentes. Mesmo que escrito em uma linguagem de alto nível (como, por exemplo, a linguagem "C"), alguns pontos do programa têm que ser amarrados de acordo com as características específicas do hardware. Esse é o caso, por exemplo, da entrada e saída nas primitivas do sistema operacional.

Os códigos fontes do PORTOS-TF foram divididos em módulos. Essa divisão obedeceu a dois critérios: funcionalidade e portabilidade. Os arquivos do tipo cabeçalho, isto é, arquivos de definições, estruturas,

variáveis e protótipos de funções, foram separados. As funções foram agrupadas de acordo com o assunto que tratam. Assim, por exemplo, as funções do núcleo constituem um arquivo, as funções de inicialização constituem outro e assim por diante. A partir dessa divisão funcional, os elementos (definições, estruturas, funções, etc.) de um mesmo grupo foram desmembrados em arquivos diferentes de acordo com a dependência que apresentavam em relação ao hardware. Dessa forma, cada grupo de funções associadas a uma determinada divisão lógica do sistema operacional ficou separado em dois arquivos: um contendo apenas elementos independentes do hardware e outro contendo elementos dependentes. Existem alguns casos, contudo, que não há elementos dependentes do hardware e outros onde só há elementos desse tipo. Nesses casos, só existe um único arquivo para o grupo lógico. Apenas como ilustração, tem-se que arquivos que contêm elementos dependentes do hardware têm o mesmo nome de seus pares precedido apenas por um cifrão ($).

II.3.1. Características da Máquina Padrão:

A máquina padrão do PORTOS-TF tem as seguintes características:

. mecanismo interrupção de relógio de tempo-real;
. registros de segmento, pelo menos para código e dados;
. um único processador e
. um compilador "C" para seu processador.

Dizer que o PORTOS-TF é portátil para todas as máquinas que tiverem essas características é uma afirmação muito forte, pois algumas delas podem apresentar particularidades que impeçam o seu transporte de maneira simples. Todavia, pode-se afirmar que máquinas que tiverem essas características são fortes candidatas a pertencerem

ao "domínio do PORTOS-TF".

É imprescindível para um sistema operacional de tempo-real que o hardware no qual se baseia tenha a capacidade de gerar interrupções em intervalos de tempo constante, ou seja, uma interrupção de relógio de tempo-real. Essa interrupção é usada para o escalonamento por tempo e para o controle das temporizações do sistema.

Existem situações em que, no sistema aplicativo, várias atividades iguais ocorrem em paralelo. Dessa forma, o código executado em cada uma delas é o mesmo. O que difere é o estado das variáveis, registradores e da pilha do processo. Assim sendo, de modo a economizar memória, a área que armazena esse código pode ser compartilhada entre esses processos. Cada um deles, contudo, deve ter sua própria área de dados e de pilha. Logo, é necessária a existência de registradores de segmento (ou registradores de base) de forma a ser possível a criação de várias instâncias de um mesmo processo. O sistema operacional efetua esse compartilhamento de forma automática: quando os processos "gêmeos" são criados, o pai informa a mesma área de código para todos, mas áreas de dado e de pilha diferentes.

Um núcleo multitarefa pode gerenciar um ou mais processadores. Contudo, no caso do processador não ser único, deve-se ter a preocupação adicional de se prevenir contra eventuais interrupções de hardware. Sabe-se que o núcleo de um sistema operacional deve ser executado com interrupções inibidas. Para que isso seja garantido em um processador, basta que as interrupções de hardware desse processador estejam desabilitadas. Isto pode ser obtido automaticamente, em um mecanismo de interrupção de software, ou explicitamente, a partir de instruções de máquina. O PORTOS-TF parte da suposição de que não há mais de um processador em cada estação em que

está sendo executado. Dessa forma, não tem qualquer mecanismo de gerenciamento de múltiplas UCPs.

Pelo fato do sistema operacional estar todo escrito na linguagem C, é imperativo que exista um compilador dessa linguagem que gere código de máquina para o processador para o qual se deseja transportar o PORTOS-TF. Esse compilador deve seguir os padrões de KERNIGHAN & RITCHIE (1978) e ANSI e permitir chamadas recursivas de procedimentos, pois há diversas funções do sistema operacional que se valem dessa característica.

II.4. Procedimentos de Transporte do PORTOS-TF:

Um sistema operacional não pode ser "independente da máquina". Como já mencionado anteriormente, há pontos que estão intimamante ligados à sua arquitetura, como o salvamento e a restauração do contexto, por exemplo (vide capítulo III). Dado que o sistema operacional pode ser portátil para equipamentos de arquiteturas diferentes, algumas alterações devem ser promovidas para o seu completo transporte.

De acordo com o exposto no ítem II.3, os fontes do PORTOS-TF estão separados em arquivos dependentes e independentes da arquitetura da máquina. Desde que haja um compilador C que gere um código executável para a máquina destino (isto é, aquela para onde se está transportando o sistema operacional), basta que esses módulos sejam compilados. Os arquivos dependentes da arquitetura da máquina destino devem, contudo, ser reescritos.

As partes do PORTOS-TF que são dependentes da arquitetura da máquina destino estão ligadas aos seguintes tópicos:

. salvamento e restauração de contexto;
. iniciação de vetores de interrupção de software;
. interceptação da rotina de tratamento do relógio de tempo-real;
. rotinas de escrita em vídeo e
. definições de algumas constantes e estruturas.

A primeira implementação do PORTOS-TF foi realizada em equipamentos do tipo IBM-PC (vide apêndice A). Para outros tipos de computadores, os módulos dependentes da máquina devem ser parcialmente ou, em alguns casos, totalmente alterados. Apesar dessa primeira implementação ter sido toda feita em "C", nada impede que se use outras linguagens, incluindo Assembly, na reconstrução dos módulos dependentes da máquina.

Em alguns tipos de arquitetura, onde não exista a estrutura de interrupção de software, as chamadas ao núcleo do sistema operacional podem ser feitas através de um vetor de "jumps". Nesse esquema, é realizada uma chamada a um endereço de memória absoluto onde será escrita uma rotina que selecionará e encaminhará a chamada à função pertinente. Todavia, como nessa situação não há mascaramento nem desmascaramento automático das interrupções de hardware, ao contrário do que ocorre em um esquema de interrupção de software, deve-se garantir que as interrupções de hardware ficarão desabilitadas enquanto não retornarem da chamada. Isso pode ser alcançado através da utilização de intruções de habilitação e desabilitação de interrupções na entrada e na saída, respectivamente, de cada uma dessas rotinas.

Alguns computadores possuem relógios de tempo-real (condição necessária para pertencerem ao domínio do PORTOS-TF). Porém nem todos eles têm uma rotina original de tratamento de interrupção de relógio (que pode, por exemplo, vir gravada em uma EPROM). Esse é o caso de

alguns computadores dedicados. Nessa situação, basta implementar-se a rotina de tratamento de interrupção de relógio de tempo-real do PORTOS-TF. Todavia, em máquinas que possuam uma rotina original de tratamento do relógio, deve ser feita uma interceptação de sua chamada a fim de introduzir a rotina equivalente do PORTOS-TF. No apêndice A, existe um ítem relacionado com esse assunto.

Como o mapeamento da memória de vídeo é particular a cada computador, as funções associadas ao seu uso devem ser reescritas para cada tipo de equipamento distinto.

Algumas das constantes usadas pelo sistema operacional são dadas por particularidades do hardware. É o caso de, por exemplo, o número de pulsos do relógio que totalizem um segundo, ou então o número da interrupção de software onde pode ser colocada a rotina de entrada para o núcleo do sistema operacional. Elas devem, pois, ser redefinidas para cada equipamento diferente.

Alguns tipos de variáveis também podem depender da arquitetura do processador da estação. O conceito de ponteiro longo ("far pointer") que existe na família de processadores do 8088 (usada nos equipamentos do tipo IBM-PC) por exemplo, pode não fazer sentido em outros tipos de UCPs.

CAPÍTULO III

NÚCLEO DO SISTEMA OPERACIONAL

III.1. Introdução:

A arquitetura de um sistema operacional muitas vezes confronta dois fatores importantes em um projeto de software: simplicidade X eficácia. Implementar mecanismos de forma simples e direta é uma preocupação que deve estar sempre na mente de qualquer projetista. Quanto menor a complexidade, menor a probabilidade de ocorrência de erros na implementação e mais fácil e rápida a manutenção. Existem situações, contudo, onde estruturas simples não conseguem desempenhar as tarefas envolvidas de forma eficaz. Isso, porém, não implica que o projeto tenha que se tornar extremamente complexo. Apenas, em tais situações, opções mais elaboradas devem ser adotadas.

A primeira questão a ser abordada no projeto de um sistema operacional diz respeito à sua estruturação. Segundo HOLT & outros (1978), há duas formas básicas de se organizar um sistema operacional: através de um monitor monolítico e através de um núcleo.

O PORTOS-TF é estruturado a partir de um núcleo pois, desta forma, apenas as operações críticas são executadas com interrupções desabilitadas, ao contrário do que ocorre em um monitor monolítico.

Cabe ao núcleo do sistema operacional gerenciar o processador, suprir mecanismos de comunicação e sincronização entre processos (vide capítulo IV) e oferecer primitivas que implementem, ou sobre as quais sejam implementados, mecanismos de tolerância a falhas (vide capítulo VII). O gerenciamento do processador entre os

diversos processos que competem por esse recurso é bastante ilustrado na literatura (DEITEL, 1984; HOLT & outros, 1978; MADNICK & DONOVAN, 1974). As demais funções do sistema operacional ficam fora do núcleo, dispostas em entidades de software, assíncronas e interrompíveis chamadas "processos".

Cada recurso de hardware pode ser controlado por um processo, batizado de "processo gerente" do recurso. Dessa forma, o núcleo não necessitará se preocupar com o gerenciamento dos recursos periféricos de hardware, podendo assim dedicar-se às atividades já mencionadas.

Um processo pode estar em três estados: "rodando", "pronto" e "bloqueado". O sistema operacional escalona os processos de acordo com a política mais conveniente para o ambiente onde será empregada. Esta política pode variar de sistema para sistema. DEITEL (1984) dedica um capítulo inteiro a esse tópico.

Há duas formas através das quais um processo pode perder o controle da UCP, ou seja, pelas quais pode haver um escalonamento: por evento e por tempo (HOLT & outros, 1978; DEITEL, 1984; KIRRMAN & KAUFMANN, 1984). No escalonamento por tempo, cada processo recebe uma "fatia de tempo ('time slice')" durante a qual usa o processador. Uma vez expirado esse tempo, o processo será escalonado. Ocorre, porém, que, durante o tempo em que está de posse da UCP, o processo pode efetuar uma chamada ao sistema operacional que o torne "bloqueado". Se isso ocorrer, torna-se um desperdício de tempo aguardar até que a fatia de tempo desse processo expire para promover o escalonamento. Dessa forma, quando um processo fica "bloqueado" por uma chamada ao sistema operacional ele é imediatamente escalonado. Esse tipo de escalonamento recebe o nome de "escalonamento por evento".

Para que o núcleo coordene o escalonamento dos processos é necessário que ele conheça algumas informações sobre cada um deles. Essas informações ficam armazenadas em uma estrutura de dados chamada "descritor de processo (DP)". As informações guardadas nos DPs variam para cada sistema operacional. Algumas informações, contudo, são comuns a todos os sistemas. Dentre elas pode-se citar o identificador do processo, seu estado ("rodando", "pronto" ou "bloqueado") e a localização de sua pilha.

Cada processo deve ter sua própria pilha. O sistema operacional pode possuir pilha própria ou usar a pilha do processo corrente toda vez que for acionado. A primeira opção introduz um processamento extra, para a troca de pilhas na entrada e na saída do sistema operacional. A segunda, por outro lado, obriga os processos a terem uma folga na área de pilha suficiente para que possa ser usada pelos procedimentos do sistema operacional. O PORTOS-TF adotou a primeira opção tendo, assim, sua própria pilha.

Uma outra informação que pode ser levada em consideração pelo sistema operacional e, nesse caso, deve estar nos DPs é a "prioridade" dos processos. A prioridade do processo serve para informar ao sistema operacional qual sua importância em termos de urgência do uso da UCP. Processos mais prioritários têm preferência no uso do processador.

Em sistemas de tempo-real, a distribuição das prioridades por entre os processos é uma questão complexa. Se processos de maior prioridade ficam se alternando no uso do processador, isso pode impedir o uso deste recurso pelos processos de menor prioridade. Esse efeito é conhecido na literatura como adiamento infinito (também conhecido como "starvation", em inglês) (KIRRMANN &

KAUFMANN, 1984).

Prioridades podem ser "estáticas" ou "dinâmicas". No primeiro caso, uma vez atribuída a um processo, a prioridade não é mudada. No segundo, cada processo tem uma prioridade inicial, mas ao longo de sua execução, pode tê-la alterada para um valor mais adequado. Prioridades estáticas são mais simples de serem gerenciadas. Prioridades dinâmicas podem ser adotadas com o propósito de evitar o adiamento infinito.

III.2. Estruturas de Dados do Núcleo:

O núcleo do PORTOS-TF se vale de três estruturas de dados para o gerenciamento dos processos. São elas:

. "lista" de descritores de processos;
. "lista" de processos no estado "pronto" e
. "lista" de processos bloqueados com cláusula de tempo.

Todas as três listas enumeradas anteriormente são implementadas através de uma estrutura mista do tipo árvore binária - lista circular (vide capítulo VI) (daí porque a palavra "lista" esta entre aspas). Essa arquitetura foi adotada por apresentar um tempo médio de busca de um elemento O(log n) em oposição a um tempo O(n) de listas encadeadas.

III.2.1. Lista de Descritores de Processos:

A "lista" de descritores de processos ou, simplesmente, Lista de DPs, tem como objetivo guardar os DPs de todos os processos da estação. A chave de classificação dos elementos nesta lista é o identificador do processo. Como cada processo tem um identificador

diferente dos demais, existe um nó de árvore para cada um dos processos de uma estação. Por esse mesmo motivo, a lista circular associada a cada um dos nós da árvore possui apenas um elemento (figura III.1).

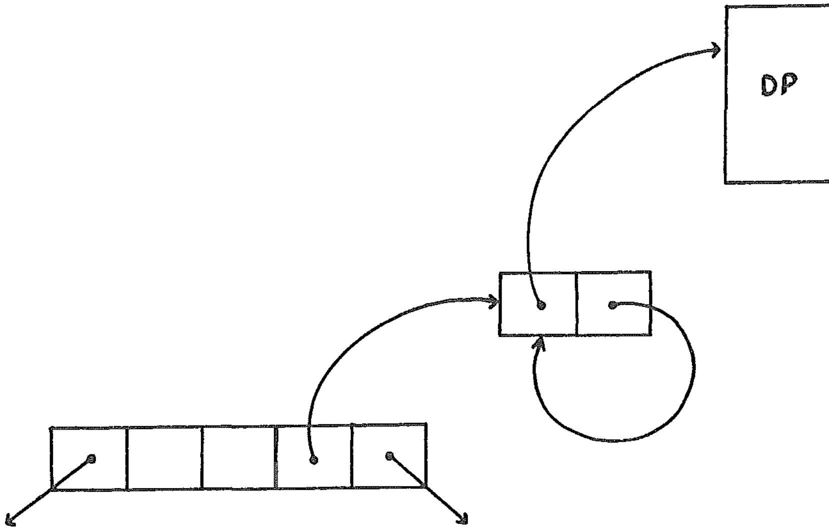


Figura III.1 : Esquema de um nó da Lista de DPs

III.2.2. Lista de Processos no Estado "Pronto":

A "lista" de processos no estado "pronto" ou, simplesmente, Lista dos Prontos, mantém classificados pela prioridade todos os processos no estado pronto da estação. Como vários processos podem ter a mesma prioridade, um mesmo nó de árvore pode servir como entrada para uma lista circular de vários elementos que apontam para os DPs de processos no estado "pronto" de mesma prioridade (figura III.2). O núcleo do sistema operacional se vale dessa lista para escolher que processo ocupará a UCP na ocasião de um escalonamento.

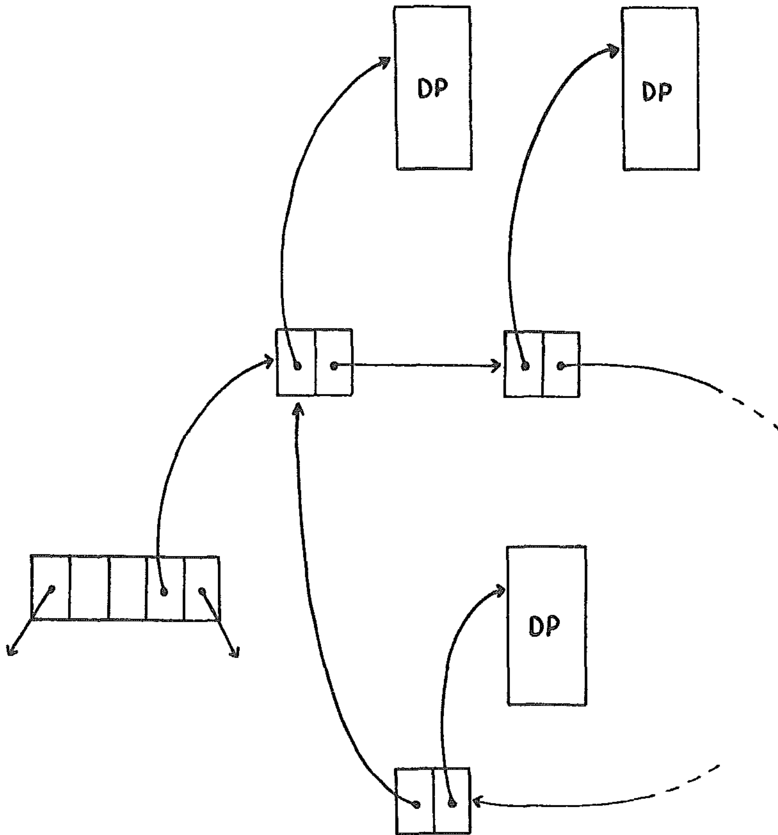


Figura III.2 : Esquema de um nó da Lista dos Prontos
ou da Lista dos Bloqueados

III.2.3. Lista de Processos Bloqueados com Cláusula de Tempo:

A "lista" de processos bloqueados com cláusula de tempo ou, simplesmente, Lista dos Bloqueados, aponta para os DPs dos processos no estado "bloqueado" mas que serão liberados automaticamente após o tempo especificado caso não ocorra nenhum evento que os libere antes desse tempo expirar. Os elementos desta lista têm como chave de classificação o tempo restante para a liberação automática. Assim como a lista dos prontos, a Lista dos Bloqueados tem as características ilustradas na figura (III.2).

Não é necessário que o sistema operacional mantenha uma lista para os processos bloqueados sem cláusula de tempo, uma vez que eles só serão desbloqueados por uma chamada ao núcleo, explícita (pelo processo) ou implícita (por uma outra primitiva).

Para controlar o momento da liberação dos processos bloqueados com cláusula de tempo o sistema operacional poderia, a cada interrupção do relógio, atualizar as chaves da Lista dos Bloqueados. Isso, porém, seria custoso, pois a lista teria que ser toda varrida, o que levaria um tempo O(n).

Foi, então, adotada na implementação uma variação do procedimento citado acima. Nesta alternativa, é empregada uma variável que contém o tempo restante para a liberação do primeiro processo da lista. A cada pulso do relógio, essa variável, e apenas ela, é atualizada. Quando chegar a zero, então os processos indicados no primeiro nó da árvore dos bloqueados serão liberados. A variável receberá então o valor da chave do novo primeiro elemento da Lista dos Bloqueados.

Para esse procedimento funcionar de forma adequada, a chave dos elementos da Lista dos Bloqueados não deve ser o tempo restante para a liberação dos processos indicados em seu respectivo nó da árvore, mas sim a diferença entre esse valor e o tempo restante para a liberação dos processos apontados no primeiro nó da árvore. Pode-se, por conseguinte, observar que a chave do primeiro nó da árvore é nula. A única desvantagem introduzida por essa técnica está no momento em que um processo é bloqueado com uma cláusula de tempo inferior à do primeiro nó da lista. Nesse caso, deve-se incrementar todas as chaves em um valor igual à diferença entre o tempo que falta para desbloquear o primeiro processo da lista e o valor da cláusula de tempo deste novo processo. Uma vez

"deslocadas" as chaves, o processo deve ser inserido na Lista dos Bloqueados com chave zero e a variável de contagem de tempo do sistema operacional alterada para o valor da cláusula de tempo do processo.

III.3. Instalação e Acesso ao Núcleo:

O núcleo do sistema operacional é um módulo de software que fica residente em memória.

As chamadas ao núcleo devem ser feitas através de interrupções de software. Os parâmetros das funções chamadas são passados em uma estrutura alocada pelo processo chamador da função. O endereço dessa estrutura é passado para o núcleo no momento da geração da interrupção de software que transfere o controle da UCP do processo para o núcleo do sistema operacional. A forma com que esse endereço é passado depende da arquitetura da máquina usada. Por esse motivo, ele é um mecanismo implementado em um módulo "dependente da máquina".

Na entrada do núcleo, são empilhados todos os registradores. Em seguida, é salvo, no DP do processo "rodando", o endereço corrente do topo da pilha. O núcleo, então, chaveia para sua pilha. Uma vez que cada processo tem sua própria pilha, os registradores podem ficar guardados nessa pilha. Isso faz com que não seja necessário ocupar-se área do DP para esse fim.

Em um nível mais elementar, constituem o "**contexto**" de um processo os valores dos registradores da máquina no momento do escalonamento, pois supõe-se que cada processo tenha suas prórias áreas de pilha e de dados. Em circunstâncias mais eleboradas, como na restauração de um estado anterior de um processo, alguns valores de variáveis podem também compor o contexto de um processo (vide

capítulo VII).

Na saída do núcleo, a pilha é chaveada para a do processo que deverá ocupar a UCP. Logo após, os registradores são desempihados, e o processamento deixa o núcleo.

Há duas formas do núcleo ser chamado: por um processo (pela evocação de uma primitiva) ou através de uma interrupção do relógio.

Quando a entrada se dá pela chamada de uma de suas primitivas, o núcleo, após efetuar o salvamento do contexto e a troca de pilha, chama a função desejada.

Quando é o relógio que aciona o núcleo, ele verifica se é o momento de liberar algum processo bloqueado com cláusula de tempo. Em caso afirmativo, executa a liberação do processo. Verifica, em seguida, se a fatia do processo que ocupa a UCP não está congelada (ver itens III.5.10 e III.5.11). Se não estiver, decrementa seu valor em uma unidade. Se esse valor chegou a zero, chama a função de escalonamento (ver ítem III.5.16). O escalonamento do processo será, então, promovido desde que não esteja desabilitado (ver ítens III.5.14 e III.5.15). Antes de iniciar o procedimento de retorno, são executadas outras operações como atualizar a hora global do sistema, rearmar o controlador de interrupções, etc.

III.4. Características dos processos do PORTOS-TF:

Processos, na qualidade de entidades autônomas, são executados de forma independente uns dos outros. A cooperação entre eles se dá unicamente através do emprego das primitivas de troca de mensagens (ver capítulo IV). Sua comunicação com o sistema operacional é

feita pela chamada de primitivas do núcleo (ver ítem III.5).

Cada processo tem sua própria área de dados e de pilha. Áreas de código podem ser compartilhadas por mais de um processo. Dessa forma, podem ser criadas em uma estação mais de uma instância de um mesmo processo. Na verdade, para o sistema operacional, cada uma dessas instâncias constitui um novo processo.

III.5. Funções do Núcleo:

O núcleo do PORTOS-TF apresenta as seguintes funções:

. criação e destruição de processos;
. obtenção do identificador do processo corrente;
. bloqueio e liberação de processos;
. obtenção e alteração da prioridade de processos;
. obtenção e alteração da fatia de tempo atribuída aos processos;
. congelamento e descongelamento da fatia de tempo atribuída aos processos;
. habilitação e desabilitação de geração de exceções;
. habilitação e desabilitação de escalonamento;
. escalonamento do processo corrente e
. obtenção do estado do sistema operacional.

III.5.1. Criação de Processos:

A criação de um processo é feita através da execução da primitiva "cria_processo". Qualquer processo pode chamar essa primitiva e, portanto, criar um novo processo.

Em alguns sistemas operacionais como, por exemplo, o UNIX, os processos criadores se tornam "pais"

dos "filhos" criados. Assim sendo, quando o pai é eliminado, os filhos também o são. Não é dessa forma que opera o PORTOS-TF. Nele, não existe nenhuma relação de "parentesco" entre o processo "criador" e o "criado".

Dessa maneira, uma vez criado, um processo desconhece seu criador. O criador, contudo, obtém como código de retorno da primitiva evocada o identificador do processo criado. Com ele, o criador poderá executar todas as operações disponíveis sobre o processo criado. Dessa forma, apesar de não existir uma relação de "pai e filho", no sentido mais comum dos termos, entre esse par de processos fica estabelecida uma hierarquia. O processo criador promove, em muitos casos, a configuração entre os processos por ele criados (vide capítulo IV).

A criação de um processo depende dos seguintes parâmetros:

. endereço inicial do código
. prioridade
. tamanho da fatia de tempo
. endereço inicial da pilha
. endereço inicial da área de dados

A prioridade de um processo pode variar de 1 a 255, sendo 255 a mais baixa e 1 a mais alta. Um processo só ocupa a UCP quando não houver outros de maior prioridade na Lista dos Prontos. Alguns sistemas operacionais como o POOLPO (KIRRMANN & KAUFMANN, 1984) adotam um controle cíclico dos processos. Esse tipo de controle estabelece prioridades iguais para todos os processos uma fatia de tempo grande o suficiente para que cada processo execute uma determinada ação. Processos grandes demais devem ser divididos em vários pequenos processos no momento de sua programação ou através de interrupções do relógio. Essa política evita o adiamento infinito mas, em contrapartida, provoca a subutilização da UCP. O PORTOS-TF adotou o que

KIRRMANN & KAUFMANN (1984) chamam de "escalonamento dirigido por evento" ("event-driven mode") por permitir um melhor aproveitamento do processador. Fica a cargo do projetista do software aplicativo a atribuição adequada das prioridades de seus processos. Caso ele deseje um escalonamento cíclico, basta atribuir a mesma prioridade a todos os processos.

O tamanho da fatia de tempo indica qual o valor da fatia de tempo do processo medido em pulsos do relógio. Quanto maior esse tamanho, mais tempo o processo passará no estado "rodando" antes de ser escalonado por tempo. O tamanho da fatia de tempo de um processo não tem qualquer influência no escalonamento por evento. Essa facilidade do sistema operacional torna viável que processos de mesma prioridade façam um uso diferenciado da UCP em termos de proporção de tempo. Permite, por conseqüência, que no caso de um escalonamento cíclico, as fatias de tempo sejam adequadas a cada processo.

O endereço inicial da pilha é o endereço da posição onde será colocado o primeiro valor a ser empilhado pelo processo. O núcleo do sistema operacional não faz nenhum cheque de estouro de pilha. O processo, ao ser escrito, deve, então, reservar uma área cujo tamanho não seja nunca inferior ao tamanho máximo que a pilha pode alcançar.

No DP do processo, são guardadas apenas as informações relativas à prioridade, tamanho da fatia de tempo, endereço inicial da pilha do processo e a lista dos portos (ver capítulo IV) associados ao processo (na criação, não há portos associados e, portanto, a lista é iniciada vazia). O endereço inicial do código e o endereço inicial da área de dados, são escritos na pilha do processo, no momento de sua criação, sendo de lá retirados e escritos nos registradores pertinentes na hora em que o

processo for despachado. Essa operação é realizada durante o procedimento de saída do núcleo, já descrito anteriormente neste capítulo.

O procedimento de criação de processos desempenha as seguintes tarefas:

. escolhe o identificador do processo;
. cria o DP do processo e insere-o na Lista de DPs;
. deixa-o no estado "bloqueado" e
. prepara a pilha do processo para o primeiro escalonamento.

Este procedimento retorna o identificador do processo. Caso o identificador seja 0 (zero), isso implica que não há mais identificadores disponíveis.

O processo recém criado fica no estado "bloqueado", necessitando ser posteriormente "liberado" para iniciar sua execução. Este procedimento é adotado pois, via de regra, o processo irá se comunicar com os demais. Para tanto, será, ainda, necessário a criação e a ligação dos portos a ele pertencentes (ver capítulo IV). Como a ligação dos portos só pode se dar após a criação dos processos aos quais pertencem, o início da execução destes está, desta maneira, vinculada ao término da etapa de configuração do sistema. Caso começassem a ser executados antes, poderia ocorrer a perda de mensagens.

A carga dos processos para a memória primária do computador, bem como a gerência das áreas alocadas para o código, dados e pilha do processo não são de competência do PORTOS-TF. Como abordado no capítulo I, este sistema operacional é voltado para arquiteturas onde todo o código já esteja em memória (gravados em EPROMs ou carregados em RAM por pequenos módulos de software independentes) ou onde exista um sistema operacional nativo que provenha funções de carga de programas da memória

secundária para a primária. Dessa forma, o procedimento de criação de processos assume que o processo está todo carregado em memória e que suas áreas de dados e de pilha também estão reservadas.

III.5.2. Destruição de Processos:

Um processo é terminado, ou destruído, passando-se como parâmetro seu identificador.

Caso o processo exista (isto é, possua um DP na lista de DPs), ele será "bloqueado" sem cláusula de tempo (veja ítem III.5.4) e seu DP será retirado da Lista de DPs, tendo a área que ocupava liberada.

Como, de acordo com o exposto no ítem III.5.1, o PORTOS-TF não gerencia as áreas alocadas para o código, dados e pilha do processo, nenhuma ação é tomada com relação a essas áreas.

Essa função retorna o valor do identificador passado como parâmetro caso o processo exista. Caso contrário, é retornado o valor nulo.

III.5.3. Obtenção do Identificador:

Essa função não tem parâmetros e, ao ser chamada, retorna o identificador do processo que a chamou.

Como um processo não sabe, ao ser criado, qual identificador lhe foi atribuído, pode ser conveniente que ele evoque esta primitiva para poder efetuar as demais operações no núcleo sobre si mesmo.

III.5.4. Bloqueio de Processos:

Esta primitiva bloqueia um processo (cujo identificador se constitui em um parâmetro da primitiva) por um determinado tempo (um outro parâmetro da função). Caso esse tempo expire antes do processo ser desbloqueado (vide ítem III.5.5), será chamada uma rotina de serviço, cujo endereço também foi passado como parâmetro. Caso o tempo de bloqueio seja infinito (isto é, o processo só será desbloqueado via uma chamada da primitiva de desbloqueio), o parâmetro da rotina de serviço será ignorado.

Uma vez desbloqueado, um processo não tem como saber se o foi por tempo ou por uma atuação da primitiva de desbloqueio. Isso é muito comum durante transações de comunicação. A rotina de serviço tem como função executar as ações desejadas pelo processo que acaba de ser desbloqueado, caso ele o tenha sido por expiração do tempo previsto. Caso não se deseje tomar nenhuma ação no momento da liberação do processo, mas sim algum tempo depois, a rotina de serviço pode ser usada para indicar em uma variável do processo que este foi liberado por um evento e não por tempo. Se, todavia, não for interessante diferenciar a forma pela qual o processo foi desbloqueado, basta que o endereço passado como sendo o da rotina de serviço seja nulo.

Quando é chamada, a primitiva de bloqueio verifica se o processo existe. Caso não exista, retorna o valor nulo. Se existir, retornará o valor do identificador do processo.

Se o identificador do processo for válido, então será efetuada a seguinte seqüência:

. se o processo já estiver bloqueado ele será desbloqueado;

. o processo será retirado da Lista dos Prontos;

. caso haja cláusula de tempo:
  . guarda o endereço da rotina de serviço do DP;
  . põe o processo na Lista dos Bloqueados (ver ítem III.2.3);
. escalona, se estiver no estado rodando.

Se, na chamada da primitiva, o processo que se deseja bloquear já estiver nesse estado, este será desbloqueado para, em seguida, ser bloqueado novamente. Justifica-se essa política a partir do fato de que a última chamada corresponde à necessidade mais atual do sistema. Portanto, o novo estado do sistema deverá corresponder a essa expectativa. Este é o caso quando, por exemplo, um processo que se deseja matar está bloqueado por tempo. Caso ele não passe a ser bloqueado por tempo infinito, ou de outra forma, não seja retirado da lista dos bloqueados, quando o tempo do bloqueio expirar, será acionada uma rotina de serviço e manipulado um DP que não mais existem. Isso terá resultados imprevisíveis mas possivelmente de conseqüências danosas ao sistema como um todo.

III.5.5. Liberação de Processos:

A primitiva que libera, ou desbloqueia, um processo cujo identificador lhe foi passado como parâmetro, retorna esse mesmo identificador caso o processo esteja bloqueado. Caso contrário, o valor retornado será nulo.

O procedimento de liberação de um processo é o seguinte:

. se houver cláusula de tempo, ou seja, o DP estiver na Lista dos Bloqueados, o processo será removido dessa lista;
. o DP será inserido na Lista dos Prontos e
. será efetuado o escalonamento se o processo estiver "rodando".

III.5.6. Obtenção da Prioridade:

Essa primitiva busca o DP do processo cujo identificador é passado como parâmetro e retorna a prioridade alí assinalada. Caso o DP não seja encontrado, será retornada uma prioridade nula.

III.5.7. Alteração da Prioridade:

Para alterar a prioridade de um processo, caso ele exista, a primitiva de alteração bloqueia o processo, caso já não esteja, e altera o campo de seu DP relativo à prioridade. Antes de terminar sua execução, libera o processo, caso este não estivesse bloqueado na entrada da primitiva.

Bloquear e desbloquear o processo é a forma mais simples de mudar a posição do processo na Lista dos Prontos. Convém lembrar que a Lista dos Prontos é ordenada por ordem crescente de valores prioridades.

Esta primitiva recebe como parâmetro o valor do identificador do processo que se deseja alterar a prioridade. Como valor de retorno, devolve a prioridade original do processo, caso ele exista. Caso contrário, devolve o valor nulo.

III.5.8. Obtenção da Fatia de Tempo:

Essa primitiva busca o DP do processo cujo identificador é passado como parâmetro e retorna o valor da fatia de tempo assinalada em seu DP. Caso o DP não seja encontrado, será retornado um valor nulo.

III.5.9. Alteração da Fatia de Tempo:

Uma vez de posse do identificador do processo que se deseja alterar a fatia, passado como parâmetro, seu correspondente DP é procurado na Lista de DPs. Caso não seja encontrado, é retornado o valor nulo. Caso contrário, será retornado, ao fim da primitiva, o valor original da fatia de tempo do processo.

Caso o processo que se está alterando o tamanho da fatia de tempo esteja "rodando", será aplicado o seguinte algoritmo. Se o novo valor, passado como parâmetro, for menor do que o tempo já transcorrido da fatia original, o processo será escalonado. Caso seja maior ou igual, a variável que conta o número de pulsos do relógio que faltam para promover um escalonamento por tempo será atualizada pelo novo valor da fatia.

III.5.10. Congelamento da Fatia de Tempo:

Esta primitiva faz com que não mais se decremente, a cada pulso do relógio, a variável que conta o número de pulsos do relógio que restam para ser efetuado um escalonamento por tempo. O efeito desse procedimento é o "congelamento" da fatia de tempo do processo "rodando".

III.5.11. Descongelamento da Fatia de Tempo:

Esta primitiva cancela o efeito da anterior (III.5.10).

III.5.12. Habilitação de Geração de Exceções:

Essa rotina habilita o campo da máscara de controle de permissão de geração de exceção. A exceção que se vai habilitar é passada como parâmetro para a rotina.

III.5.13. Desabilitação de Geração de Exceções:

Essa rotina desabilita o campo da máscara de controle de permissão de geração de exceção. A exceção que se vai habilitar é passada como parâmetro para a rotina.

III.5.14. Habilitação de Escalonamento:

Através da chamada dessa primitiva, o núcleo fica autorizado a efetuar um novo escalonamento. Ela cancela o efeito da primitiva que desabilita o escalonamento (ver III.5.15).

Se a fatia de tempo do processo corrente já tiver se esgotado (isto é, seu valor chegado a zero), então o processo será escalonado na interrupção seguinte do relógio. Caso contrário, o efeito líquido observado será como se não tivesse ocorrido uma desabilitação de escalonamento.

III.5.15. Desabilitação de Escalonamento:

Esta primitiva liga um "flag" que é testado pela rotina que promove o escalonamento (ver III.5.16). Quando ligado, este "flag" impede que o núcleo execute qualquer escalonamento.

Durante este período de desabilitação, contudo, a fatia de tempo do processo continua a ser decrementada, ao contrário do que ocorre quando é chamada a primitiva de "congelamento da fatia de tempo" (ver III.5.10). Porém, quando sua fatia chega a zero, permanece com esse valor.

III.5.16. Escalonamento do Processo Corrente:

Esta primitiva é a responsável pela eleição do processo que irá ocupar a UCP quando o núcleo terminar sua execução.

Seu algoritmo consiste na rotação da lista circular associada ao primeiro nó da Lista dos Prontos, se este for o processo que estiver ocupando a UCP. Além disso, é feita a atualização das variáveis que indicam (1) o DP corrente, ou seja, o DP do processo no estado "rodando" e (2) quantos pulsos do relógio faltam para terminar a fatia de tempo do processo corrente.

III.5.17. Obtenção do Estado do Sistema Operacional:

Informa quantos processos existem na estação, quantos estão "prontos" e quantos estão "bloqueados".

CAPÍTULO IV

COMUNICAÇÃO E SINCRONIZAÇÃO ENTRE PROCESSOS

IV.1. Introdução:

"Processos concorrentes" são aqueles que existem em um mesmo momento. Tais processos podem ser totalmente independentes ou interagir entre si. No primeiro caso, valem-se apenas da capacidade do sistema operacional em gerenciar os recursos da máquina, em especial a UCP. Porém, quando cooperam entre si, o fazem através de uma comunicação. Ocorre que os processos são assíncronos entre si, isto é, o fluxo de execução de cada um é independente dos demais. Na hora de se comunicar com outro, um processo desconhece em que ponto esse outro está. É, pois, necessário que haja uma forma de se sincronizarem. Desse modo, além do gerenciamento dos recursos físicos e lógicos, o sistema operacional deve prover mecanismos de comunicação e sincronização entre processos.

IV.2. Mecanismos de Comunicação e Sincronização entre Processos:

A comunicação e a sincronização entre processos pode ser obtida através de várias técnicas bastante conhecidas (HOLT & outros, 1978; DEITEL, 1984; MENDES, 1984; SEGRE, 1987; KIRNER & MENDES, 1988).

Existe um elemento decisivo na escolha do mecanismo de comunicação e sincronização: a memória. Arquiteturas monoprocessadas ou de processadores fortemente acoplados (isto é, que compartilham um mesmo barramento interno) dispõem de uma memória comum referenciada por todos os processadores. Sistemas operacionais multitarefa

que são executados sobre essa arquitetura podem assumir que é possível os processos usarem áreas comuns de memória para se comunicar e sincronizar. Por outro lado, arquiteturas baseadas em processadores fracamente acoplados (isto é, que não compartilham um mesmo barramento interno) não dispõem dessa característica.

Um sistema distribuído é composto por várias estações. Nessas estações, independentemente de serem monoprocessadas ou multiprocessadas, existe um ambiente multitarefa. Dessa forma, é possível que os processos de uma mesma máquina compartilhem áreas comuns de memória. Ocorre, porém, que não se pode generalizar essa propriedade para todos os processos do sistema distribuído, uma vez que não existe uma área de memória que seja comum a todos eles.

Pode-se, então, distingüir duas classes de mecanismos de comunicação e sincronização entre processos: os baseados em memória compartilhada e os baseados em troca de mensagens. No primeiro, a mensagem é escrita pelo processo produtor em uma determinada área de memória. (Chama-se de "**processo produtor**" àquele que gera uma mensagem.) Esta área é, então, lida, no momento oportuno, pelo processo consumidor. (Chama-se de "**processo consumidor**" àquele que utiliza mensagens geradas por processos produtores.) Já nos mecanismos baseados em troca de mensagens, os dados a serem enviados se encontram em uma área de memória do processo remetente. Estes dados são então copiados para uma área do processo destinatário. Em algumas situações pode ainda ser necessário o emprego de "buffers" para o armazenamento temporário das mensagens.

A literatura ilustra vários mecanismos de memória compartilhada, como regiões críticas, regiões críticas condicionais, semáforos, monitores, etc. Esses mecanismos, contudo, não se aplicam a sistemas distribuídos, como já visto anteriormente. Já que o

PORTOS-TF segue um modelo distribuído, mesmo quando implementado em uma única estação, as atenções desse texto estarão voltadas para mecanismos de comunicação e sincronização entre processos para sistemas com memória distribuída.

Em sistemas com memória distribuída, um dado compartilhado não pode ser obtido através da leitura direta da posição onde se encontra. Ao invés disso, deve ser transportado, via um subsistema de comunicação, até uma área local onde possa ser endereçada por operações de leitura e escrita do processo.

A questão do endereçamento em sistemas distribuídos é, também, de grande importância. Um dado compartilhado não pode mais ser referenciado por seu endereço de memória. Ao contrário, deve-se identificar um endereço abstrato de onde se possa obter este dado.

IV.2.1. Endereçamento em Sistemas Distribuídos:

Como já mencionado anteriormente, em sistemas distribuídos é necessária a utilização de endereços abstratos que representem os elementos do sistema com os quais se deseja comunicar. Esse endereçamento pode ser implícito ou explícito. O endereçamento explícito pode, ainda, ser direto ou indireto.

Em sistemas com endereçamento implícito como, por exemplo, dutos (ou no inglês, "pipes"), a associação entre os processos que irão se comunicar pode ser estática ou dinâmica. No primeiro caso, ela é estabelecida na criação dos processos e só se desfaz quando eles terminam sua execução. Essa é uma situação que causa bastante limitações ao sistema, uma vez que mantém um elo entre os processos, mesmo em situações em que eles já não estão mais se comunicando. A associação dinâmica entre

processos, isto é, aquela que é mantida apenas no período em que estão interagindo, tem um caráter menos limitante. Segundo SEGRE (1987), "esta solução é apropriada para aplicações nas quais encontra-se um processo trocando mensagens longas com vários outros processos, ou uma seqüência não interrompida de mensagens curtas, com um mesmo processo".

No endereçamento explícito, os processos se comunicam com os demais através de referências a nomes ou endereços. O endereçamento explícito pode, ainda, ser direto ou indireto. No endereçamento direto, os processos com que se deseja comunicar são identificados através de nomes ou endereços globais que lhes são atribuídos. No endereçamento indireto, por outro lado, um processo não associa um nome ou endereço global ao sistema distribuído, mas sim um atributo local. Esses atributos locais, por sua vez, são ligados adequadamente, estabelecendo, assim, um elo de comunicação entre os processos emissor e receptor de mensagens.

A identificação explícita-indireta permite a maior independência entre os módulos de um sistema distribuído. Isso facilita sua reconfiguração em situações de falha, como, por exemplo, no caso da substituição de um modo faltoso por outro de reserva.

IV.2.2. Mecanismos para Sistemas com Memória Distribuída:

Alguns mecanismos de comunicação e sincronização entre processos são:

. troca de mensagens;
. "rendez-vous" e
. chamada remota de procedimentos.

A seguir serão estudados esses mecanismos.

IV.2.2.1. Troca de Mensagens:

Os mecanismos de troca de mensagens baseiam-se em duas primitivas básicas: "envia" e "recebe". A primeira pega uma mensagem existente em um "buffer" local do processo emissor e a "envia" para o processo destinatário, através do subsistema de comunicação. O processo destinatário, por sua vez, deve chamar a primitiva "recebe" para que a mensagem enviada possa ser recebida em um "buffer". A figura (IV.1) ilustra essa situação.

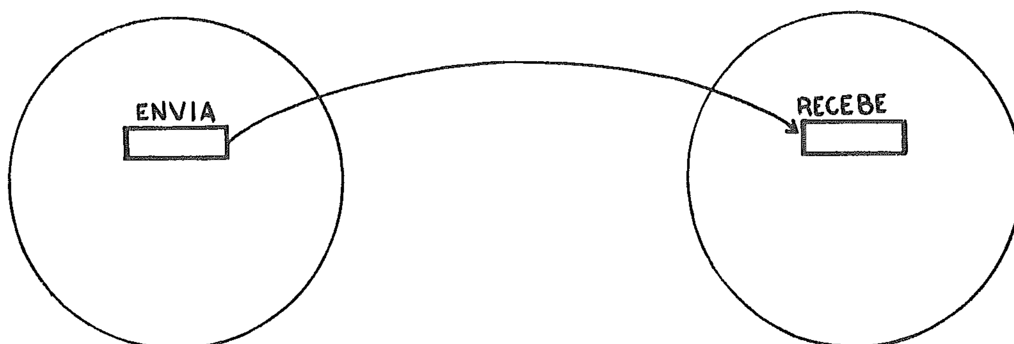


Figura IV.1 : Esquema de um mecanismo de troca de mensagens

Essas primitivas podem ser de dois tipos: bloqueantes e não-bloqueantes. Assim sendo, pode-se ter os seguintes tipos:

. envia não-bloqueante;
. envia bloqueante;
. recebe não-bloqueante e
. recebe bloqueante.

Uma primitiva bloqueante faz com que o processo que a chamou fique "bloqueado" até que a transação seja completada. Entende-se por "transação" a seqüência completa de uma troca de mensagens, ou seja, desde a chamada da primitiva "envia" até o término da execução da primitiva "recebe". (Em alguns casos, a primitiva "envia-

bloqueante" fica aguardando por uma resposta, que é enviada por uma primitiva "responde", chamada pelo processo destinatário.) Dessa forma, a primitiva "envia bloqueante" bloqueia o processo que a chamou até que a mensagem tenha sido recebida pelo processo destinatário (ou até que tenha recebido sua resposta). De forma análoga, uma primitiva "recebe bloqueante" faz com que o processo que a chamou fique "bloqueado" até que uma mensagem seja recebida.

No caso da utilização de primitivas não-bloqueantes, os processos que as evocaram continuam sua execução logo após a chamada, não esperando, assim, a conclusão da transação.

Segundo KIRNER & MENDES (1988), transações que utilizam primitivas bloqueantes são chamadas "totalmente síncronas". Transações que utilizam primitivas não-bloqueantes são chamadas "totalmente assíncronas". Aquelas que utilizam uma primitiva de um tipo e outra de outro são chamadas de "semi-síncronas".

As primitivas de troca de mensagens apresentam, de um modo geral, os seguintes parâmetros:

- identificador dos processos (remetente e destinatário);
- endereço do "buffer" local que contém ou conterá a mensagem e
- tamanho da mensagem.

O identificador do processo destinatário pode não existir em algumas implementações da primitiva "recebe", pois pode ser obtido automaticamente pelo sistema operacional. O mesmo pode acontecer em relação ao parâmetro de identificador do processo remetente quando se fala da primitiva "envia".

O identificador do processo destinatário, na

primitiva "envia" pode endereçar um ou mais processos (figura IV.2). O endereçamento de mais de um processo pode ser útil, ou mesmo necessário, em situações onde uma mesma mensagem deve ser encaminhada a diversos destinatários. Pode ser endereçado um grupo de processos ("multicast") ou todos os processos do sistema (difusão ou "broadcast").

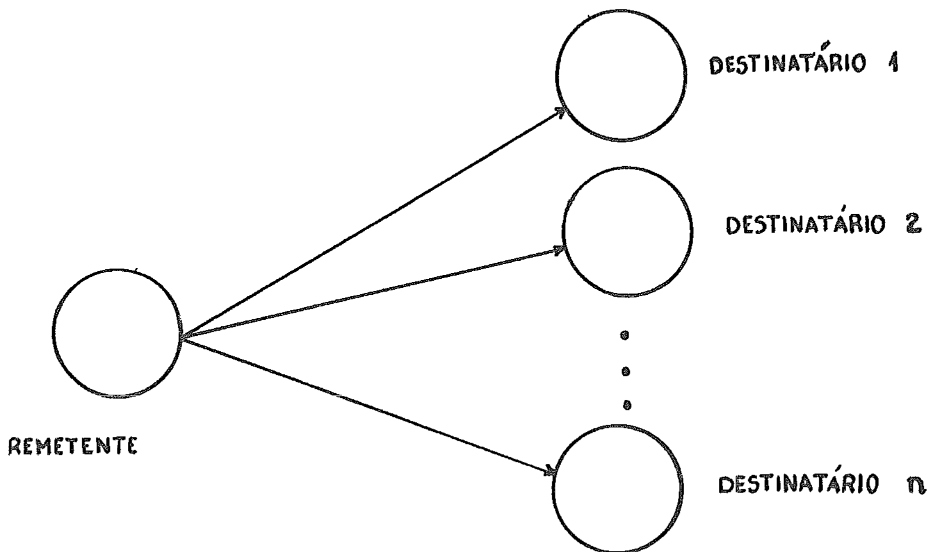


Figura IV.2 : Esquema de difusão de mensagens

O identificador do processo remetente, na primitiva "recebe", pode ser não específico, isto é, pode não endereçar um processo em particular. Esse campo pode ser preenchido com um valor que indique estar apto a receber mensagens de qualquer processo (figura IV.3). Essa situação é muito comum quando o processo receptor não sabe, de ante-mão, qual processo lhe irá enviar uma mensagem. Isso ocorre na maioria dos processos servidores de recursos. Qualquer processo do sistema, a princípio, poderá enviar-lhe uma mensagem solicitando um serviço por ele prestado, sendo, pois, impossível, ou mesmo indesejável, precisar qual será o remetente.

Figura IV.3 : Esquema de recuperação de mensagem de qualquer remetente

Quando um processo chama a primitiva "recebe", ele deve informar o tamanho máximo do "buffer" que ele reservou para a mensagem. Isso evita que o sistema operacional copie para esse "buffer", cujo endereço inicial também é parâmetro da primitiva, uma mensagem de tamanho superior ao seu, que poderia provocar a invasão de áreas reservadas a outras variáveis. No caso da primitiva "envia" o parâmetro de tamanho do "buffer" serve para informar ao sistema operacional qual o tamanho total da mensagem a ser enviada. Esse tamanho serve para o sistema operacional alocar seus "buffers" intermediários, bem como informar seu tamanho às primitivas de envio pelo subsistema de comunicação, na eventualidade de ter que enviá-la para uma outra estação no caso de um sistema distribuído.

Nas primitivas bloqueantes pode, ainda, existir um outro parâmetro que é o "tempo de espera". Esse parâmetro indica qual o tempo máximo que o processo chamador da primitiva ficará bloqueado aguardando o fim da transação. Caso esta se complete antes desse tempo expirar, o processo será desbloqueado e informado de sua conclusão. Se, ao contrário, o tempo se esgotar antes da transação ser terminada, o processo será desbloqueado, mas informado que o tempo limite expirou.

É comum que rotinas que apresentam esse parâmetro tenham um valor especial que indique que o tempo de espera é infinito, ou seja, que o processo deve ficar

bloqueado até que a transação seja completada, não importando quanto tempo levará. Processos servidores, usam freqüentemente esse valor "infinito", enquanto que processos clientes se valem, normalmente, de cláusulas de tempo.

Existe, ainda, a possibilidade de se transformar uma primitiva bloqueante em não bloqueante ao se escolher um valor que indique tempo de espera zero, desde que esse valor seja permitido pela implementação.

É interessante observar que todos os demais mecanismos de comunicação e sincronização entre processos se baseiam, de alguma forma, na troca de mensagens, dando-lhe, contudo, uma forma mais conveniente, de acordo com a aplicação a que se destina.

IV.2.2.2. "Rendez-Vous":

"Rendez-vous" (que quer dizer "encontro", em francês) é um mecanismo síncrono de comunicação e sincronização entre processos. O "rendez-vous" se caracteriza pelo fato de que os processos que se comunicam sincronizam-se em um ponto de "encontro". Caso o processo receptor da mensagem não se encontre no ponto de "rendez-vous", quando da emissão da mensagem, o processo remetente ficará bloqueado até que o receptor atinja o ponto de encontro. O inverso também é verdadeiro. Se o processo receptor atingir o ponto de "rendez-vous" e nenhuma mensagem lhe estiver disponível, ficará, então, bloqueado até que um processo lhe envie uma mensagem.

Uma variação dessa forma de "rendez-vous" é o "rendez-vous" estendido. No "rendez-vous" simples, uma vez efetuada a sincronização, os processos comunicantes seguem seus fluxos de forma independente. No "rendez-vous" estendido, é estabelecida uma transação do tipo "pedido-

resposta" ("request-reply"). Esse tipo de transação começa com uma mensagem de "pedido" e é concluída com uma mensagem, no sentido inverso, do tipo "resposta". A sincronização entre os dois processos se dá da mesma forma que no "rendez-vous" simples. A diferença está no fato de que o processo emissor da mensagem (ou do pedido) permanece bloqueado, enquanto o processo receptor executa um trecho de código, até que lhe este lhe envie uma mensagem de resposta ao pedido. A partir desse ponto, os processos retomam execuções independentes.

Ada é uma linguagem de programação concorrente que utiliza o "rendez-vous" estendido.

IV.2.2.3. Chamada Remota de Procedimentos:

A chamada remota de procedimentos é outra forma de comunicação síncrona entre processos em ambientes distribuídos. Em uma chamada tradicional de um procedimento, o fluxo de execução do programa é desviado para a subrotina e, ao final desta, retornado à instrução seguinte a da chamada com valores de retorno. O mesmo ocorre em uma chamada remota de procedimento. A diferença entre essas duas formas é que, no caso de uma chamada remota, um canal é utilizado para a passagem dos parâmetros de entrada e de saída. Segundo MENDES (1984), "o canal caracteriza o termo remota". Esse canal é, no caso de sistemas distribuídos, o subsistema de comunicação.

De acordo com KIRNER & MENDES (1988), "o processo chamador deverá ficar bloqueado até que o procedimento chamado termine". No caso do procedimento ser reentrante ou não estar sendo executado, será imediatamente acionado. Caso não seja reentrante e já esteja sendo executado, então só será posto em execução, por causa da nova chamada, logo assim que terminar sua execução corrente.

É interessante observar que o mecanismo de chamada remota de procedimento é implementado a nível de linguagem de programa, apoiado em primitivas de troca de mensagens implementadas pelo sistema operacional. Este fato se deve, basicamente, às características de cada um desses níveis: linguagem de programação e sistema operacional. Enquanto um sistema operacional deve ser o mais versátil possível, pois sobre ele podem ser usadas várias linguagens diferentes, uma linguagem de programação concorrente deve se apresentar da forma mais conveniente possível para o usuário, com estruturas as mais próximas daquelas a que ele está acostumado. A troca de mensagens é o mecanismo mais flexível de todos, embora seja também o mais trabalhoso. Já a chamada remota de procedimentos é a mais natural para aqueles que estão acostumados a trabalhar com linguagens procedimentais (ou seja, baseadas em procedimentos). Esse mecanismo, contudo, impõe algumas restrições ao programador. Daí porque a preferência pela troca de mensagens a nível de sistema operacional, deixando-se a chamada remota de procedimento para o nível de linguagem de programação, quando achado conveniente.

IV.2.3. Portos:

Como citado em SEGRE (1987), o conceito de portos (ou portas, como preferem alguns autores) foi introduzido por Balzer e por Walden como forma de identificação indireta durante a troca de mensagens entre processos. A comunicação e sincronização entre processos baseada em portos não é exatamente uma técnica diferente das demais, mas sim uma forma se troca de mensagens com endereçamento indireto.

O termo "porto" vem do inglês "port". Dado a suas características, como será visto adiante nesse ítem, alguns autores, como, por exemplo, SEGRE (1987), adotaram

em português o nome "porta" ao invés de "porto". Nesse trabalho, contudo, será empregado o termo "porto", como em KIRNER & MENDES (1988).

Na comunicação e sincronização entre processos baseados em portos, os processos comunicantes não se endereçam diretamente. Quando um processo deseja enviar uma mensagem, ele a escreve em um porto de saída. Quando, ao contrário, quer ler, vai buscá-la em um porto de entrada. Um processo só conhece os portos a ele pertencentes. Ele ignora os que são dos outros processos. Portanto, sua implementação é independente dos demais. Esse tipo de endereçamento indireto tem a grande vantagem de aumentar a modularidade do sistema distribuído facilitando, assim, seus procedimentos de configuração e reconfiguração.

Como disposto em KRAMER & MAGEE (1985), é amplamente aceito que, no projeto de um grande sistema de software, deve-se decompor o sistema em módulos que possam ser programados, compilados e testados separadamente. Esse procedimento facilita o desenvolvimento do software por lidar com programas menos complexos. O sistema como um todo é, então, composto de uma configuração desses módulos de software. Esses dois níveis de construção de um sistema de software, que claramente se distingüem, são conhecidos na literatura como "programação em pequena escala" (ou "programming-in-the-small") e "programação em larga escala" (ou "programming-in-the-large").

Uma vez que, nesse tipo de comunicação ora discutido, não há referências a elementos do escopo de outros processos, pode-se verificar que a programação em pequena escala é exeqüível.

A programação em larga escala é feita por processos de configuração que ligam portos de saída a

portos de entrada. A transferência das mensagens de um porto de saída para um de entrada é feita pelo sistema operacional.

De acordo com SEGRE (1987), portos podem ser considerados tipos abstratos de dados capazes de ser manipulados por um conjunto específico de funções. Segundo KIRNER & MENDES (1988), um processo pode efetuar as seguintes operações sobre um porto:

. criação;
. eliminação;
. conexão e
. deconexão.

Um processo, para se comunicar com os demais, deve, em primeiro lugar, criar os portos que julgar necessários para essas transações. Esses portos, que são definidos pelos processos criadores, podem ser globais ou locais.

Nas propostas de SILBERSCHATZ (1981), LISKOV (1979) e MAO & YEH (1980), os portos são entidades globais. Na comunicação entre os processos, eles se referenciam a um mesmo porto. Esse tipo de implementação, contudo, compromete a modularidade e, conseqüentemente, a potencialidade de reconfiguração dinâmica do sistema.

Posições mais atuais, como as de REID (1980), JOSEPH (1981), RUGGIERO & BRESSAN (1982), KRAMER & outros (1983) e FANTECHI & outros (1983), utilizam portos como elementos locais a cada processo. Dessa forma, faz-se necessário a ligação, ou conexão, entre eles. É possível, assim, haver a configuração dinâmica do sistema de software.

No que concerne ao sentido das mensagens, os portos podem ser classificados como de entrada, de saída ou

bidirecionais. Enquanto portos bidirecionais (também chamados de portos de difusão (KIRNER & MENDES, 1988)) estão ligados a outros portos também bidirecionais, os portos de saída, que são onde são escritas as mensagens, ficam ligados a portos de entrada, de onde são lidas as mensagens.

Segundo KIRNER & MENDES (1988), portos podem ser ligados de acordo com quatro combinações possíveis, como ilustra a figura (IV.4):

. um para um;
. um para vários;
. vários para um e
. vários para vários.

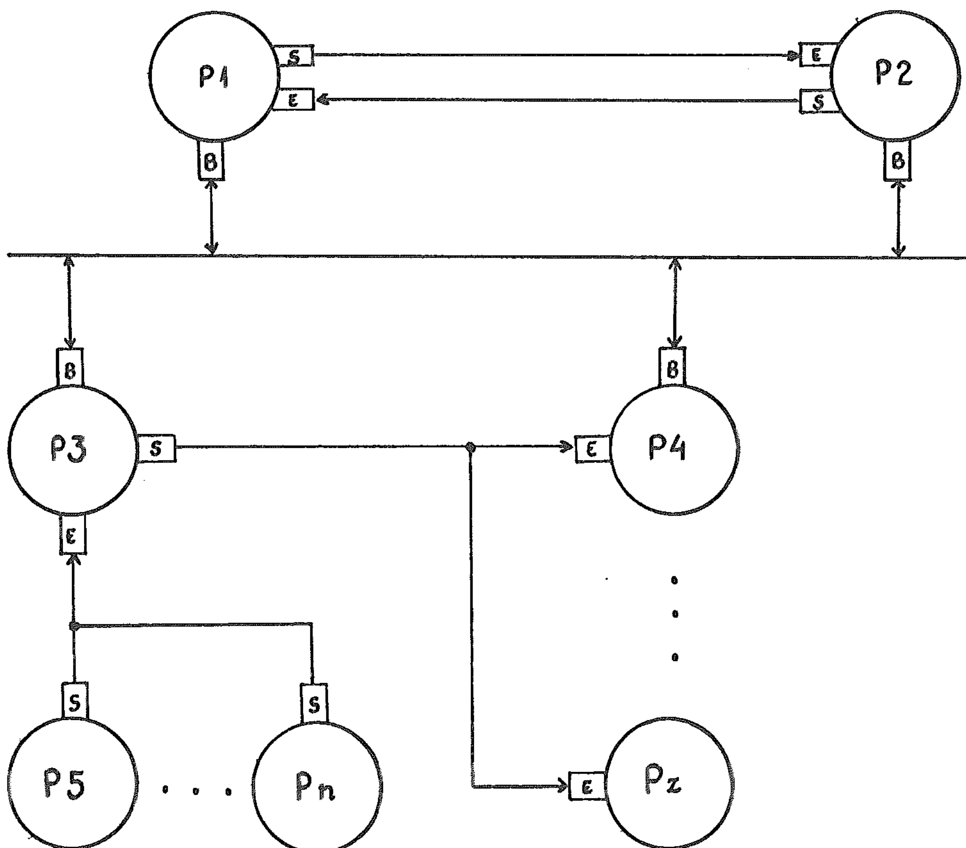


Figura IV.4 : Tipos de portos e possibilidades de ligação

A conexão "um para um" é equivalente ao endereçamento direto de um único processo na troca de mensagens simples. É a forma mais imediata de interligação de portos.

A conexão "um para vários" pode ainda ser do tipo "difusão" ou "conexão múltipla". No primeiro caso, todos os destinatários recebem a mensagem. A difusão pode ser muito útil na transmissão, por exemplo, de um aviso que diga respeito a todos os processos relativos à ligação. A conexão múltipla se caracteriza por deixar a mensagem enviada à disposição de todos os portos pertencentes à ligação até que um processo dono de um desses portos a leia. A partir desse instante, a mensagem não mais estará disponível aos demais. Essa característica é importante quando se deseja que apenas um dentre os possíveis destinatários responda a um pedido.

A conexão do tipo "vários para um" é bastante empregada na comunicação entre clientes e servidor. Nessa forma de ligação, um servidor não tem como saber a princípio qual cliente lhe fará o próximo pedido.

Finalmente, a conexão do tipo "vários para vários" pode ser encarada como sendo uma combinação das ligações "um para vários" e "vários para um".

Na comunicação e sincronização entre processos baseada em portos, existe também a questão do uso de "buffers", assim como na troca de mensagens simples. Como naquele mecanismo, neste também os "buffers" original e final das mensagens são definidos pelos processos remetente e destinatário, respectivamente. No caso, porém, de uma recepção síncrona com uma transmissão assíncrona, quando uma mensagem é escrita em uma porta de saída mas o processo destinatário ainda não executou a instrução de leitura da correspondente porta de entrada, esta deve ser

armazenada em um "buffer" intermediário de forma a liberar o "buffer" do processo remetente sem que com isso, se perca a mensagem enviada.

Além da classificação do porto quanto a seu sentido, pode-se ainda classificá-lo quanto ao tipo de mensagem que por ele passará e ao tipo de sincronismo empregado, isto é se a comunicação será síncrona ou assíncrona.

IV.3. Configuração Dinâmica de Sistemas:

Uma vez que o PORTOS-TF é um sistema operacional que implementa primitivas de tolerância a falhas, ele deve permitir a reconfiguração dinâmica do sistema como um todo. Configuração (ou reconfiguração) dinâmica de sistemas é uma área que tem recebido muita atenção em pesquisas recentes (KRAMER & MAGEE, 1985; SLOMAN, KRAMER & MAGEE, 1986; SEGRE, 1987). Segundo KRAMER & MAGEE (1985), configuração dinâmica de um sistema é a habilidade de modificar e ampliar esse sistema enquanto ele está em operação.

A modificação ou incorporação de alguma função a um sistema já existente pode ser muito trabalhosa quando não impossível se essa possibilidade não tiver sido levada em consideração quando da elaboração do sistema. Mais do que se preocupar com a configuração durante o desenvolvimento do software, deve-se projetar um sistema flexível o suficiente para que possa ser configurado em tempo de execução. O processo de edição, compilação, ligação e carga de todo o sistema, mencionado por KRAMER & MAGEE (1985), pode ser inviável por motivos econômicos ou mesmo de segurança, como é o caso de alguns sistemas de controle de processos e sistemas de controle de navegação (aeronaves, espaçonaves, etc.).

De acordo com KRAMER & MAGEE (1985), um sistema operacional deve ser capaz de anexar e eliminar módulos ao sistema, gerenciar suas interligações e permitir que esses módulos se comuniquem entre si. É ainda desejável que o sistema operacional seja capaz de promover a reconfiguração em tempo hábil sem comprometer a resposta do sistema como um todo sempre que forem detectadas falhas. Essa reconfiguração deve demandar o mínimo possível de esforço adicional ("overhead").

Dentro desse contexto, o endereçamento direto na comunicação entre processos "amarra" o sistema impedindo, assim, sua reconfiguração dinâmica.

IV.4. Implementação de um Mecanismo de Comunicação e Sincronização entre Processos Baseado em Portos:

O PORTOS-TF implementa primitivas de tolerância a falhas (vide capítulo VII). Para a execução adequada dessas primitivas, é fundamental que o sistema possa ser reconfigurado dinamicamente. Com o objetivo de garantir a modularidade entre os processos que integram o sistema, foi feita a opção do emprego de portos como forma de endereçamento usado nas trocas de mensagens entre os processos.

Os portos definidos pelo sistema operacional pode ter dois sentidos: sentido de entrada e sentido de saída. A princípio, um porto pode ser ligado a vários outros, desde que esses tenham sentido contrário ao dele. Dessa forma um porto de entrada pode estar ligados a vários portos de saída e um porto de saída pode estar ligado a vários portos. Vários portos de saída ligados a um de entrada é uma configuração característica do tipo clientes-servidor. A configuração contrária, isto é, um porto de

saída ligado a vários de entrada se presta à difusão de mensagens. Portos podem também ter ligações únicas, ou seja, um porto de saída ligado a um porto de entrada.

Não há, contudo, nenhuma caracterização do tipo de conexão feita pelo porto (simples ou múltipla) na estrutura que o descreve. O tipo de conexão é definida na configuração do sistema. Caso se deseje uma conexão simples, um porto deve ser ligado a um único outro e vice-versa durante o processo de configuração. Caso seja adequada uma conexão múltipla, um porto deve ser ligado a todos aqueles que forem convenientes. A figura (IV.5) ilustra esses três tipos de conexão entre portos.

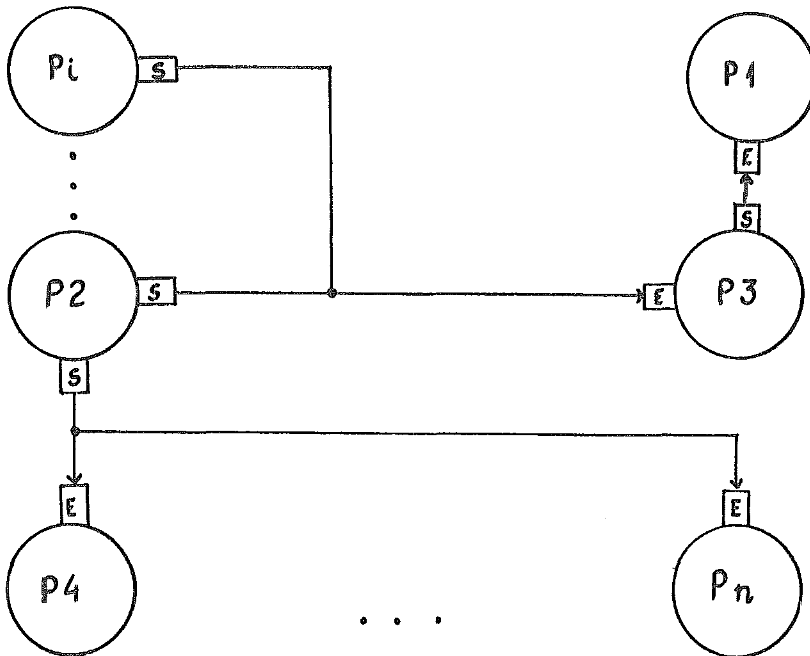


Figura IV.5 : Opções de ligação de portos no PORTOS-TF

IV.4.1. Primitivas para o Manuseio de Portos:

Portos devem ser manipulados pelo processo de configuração com o objetivo de estabelecer as ligações entre os processos. O PORTOS-TF apresenta quatro primitivas para o manuseio de portos: cria_porto,

elimina_porto, liga_porto e desliga_porto.

A primitiva "cria_porto", como o próprio nome sugere, cria (ou cadastra) um porto no sistema operacional. Na criação de um porto são especificados o número do porto, processo a que pertence e seu sentido de comunicação. Para descadastrar (ou eliminar) um porto, deve-se ser empregada a primitiva "elimina_porto". Essa primitiva requer, como parâmetros, o número do porto e o processo a que pertence. Essas duas primitivas devem ser evocadas na fase de criação e eliminação dos processos.

Na fase de configuração inicial do sistema, ou nas situações de reconfiguração, os portos devem ser ligados e desligados. As primitivas "liga_porto" e "desliga_porto" têm essas funções. Cada chamada a uma delas liga ou desliga um par de portos de sentidos diferentes. São parâmetros dessas rotinas os números dos portos e os identificadores dos processos a que pertencem.

A escolha do tipo de sincronismo das trocas de mensagens também não constituem características dos portos. Ao invés disso, uma transação é síncrona ou assíncrona de acordo com os parâmetros das primitivas de comunicação.

IV.4.2. Primitivas de Comunicação e Sincronização:

São três as primitivas do sistema operacional para a comunicação e sincronização entre processos: "envia", "recebe" e "responde". Em todas elas, o processo especifica o número do porto por onde deve passar a mensagem. O transporte dessas mensagens entre portos respeita as ligações especificadas na configuração do sistema.

As primitivas do tipo "recebe" podem ser

agrupadas de modo a que o processo possa ficar esperando mensagens que venham por mais de um porto. Para criar essa estrutura foram introduzidas duas primitivas auxiliares: "selec" e "fim_selec".

IV.4.2.1. Envia:

A primitiva "envia" escreve em um porto de saída, cujo número é passado como parâmetro, uma mensagem que tem seu endereço como um segundo parâmetro da função. O tamanho dessa mensagem também se constitui em um parâmetro da primitiva.

Essa primitiva pode ser síncrona ou assíncrona. No caso síncrono, deve ser indicado na chamada da função o endereço e o tamanho de um "buffer" que será usado para guardar a resposta à mensagem enviada. No caso de uma comunicação assíncrona, não há espera por resposta. Conseqüentemente, esses parâmetros são nulos. O último parâmetro da primitiva indica o tipo de sincronismo a ser adotado. Ele mostra o tempo de espera pela resposta da mensagem enviada.

Um tempo de espera igual a zero indica uma comunicação assíncrona, ou seja, sem resposta. Dessa forma, a primitiva "envia" tem um comportamento não-bloqueante, isto é, a mensagem é enviada e o processo continua seu fluxo.

Quando uma mensagem é escrita em um porto, ela é copiada em todos os portos de entrada a ele ligados. O procedimento de cópia verifica, para cada processo dono de um porto de entrada ligado, se ele está esperando alguma mensagem. Caso esteja, então a mensagem será copiada diretamente para o "buffer" do processo. Caso contrário, será copiada para um "buffer" do sistema operacional que ficará ligado a esse porto de entrada. Quando este

processo chamar a primitiva "recebe", vista mais adiante, a mensagem já estará disponível para ser lida.

Um tempo de espera diferente de zero indica uma comunicação síncrona e, conseqüentemente, com uma resposta associada. O processo ficará no estado "bloqueado" até que chegue a resposta ou até que expire seu tempo de espera. Ao ser desbloqueado, o processo poderá saber qual das duas situações ocorreu pelo código de retorno da primitiva. Um valor "infinito" para o tempo de espera, cujo correspondente numérico depende da máquina hospedeira do PORTOS-TF, indica que o processo chamador da primitiva "envia" só será liberado após a recepção da resposta. Caso esta nunca chegue, o processo ficará bloqueado para sempre.

IV.4.2.2. Recebe:

A segunda primitiva de comunicação, a "recebe", lê uma mensagem de um porto de entrada. Seu número é passado como parâmetro e ela é escrita em um "buffer" que tem seu endereço como um segundo parâmetro da função. O tamanho dessa mensagem também se constitui em um parâmetro da primitiva.

Essa primitiva, a exemplo da anterior, também pode ser síncrona ou assíncrona. Nesse caso, contudo, o sincronismo não tem a ver com a recepção de uma resposta, mas sim com a recepção da própria mensagem. A seleção do tipo de sincronismo também é feito pelo parâmetro "tempo de espera". Um valor nulo desse parâmetro indica que a transação será assíncrona, enquanto que um valor diferente de zero, síncrona.

Assim como na primitiva "envia", não há espera no caso de uma comunicação assíncrona. Se já houver alguma mensagem no porto, ela será copiada no "buffer" passado

como parâmetro e o controle retornado ao processo chamador. Caso não haja nenhuma mensagem, o controle do processador voltará ao processo chamador imediatamente, sendo essa situação indicada pelo código de retorno da primitiva.

Se a comunicação, por outro lado, for do tipo síncrona, o processo chamador ficará no estado "bloqueado" até que chegue alguma mensagem ou até que expire seu tempo de espera. Ao ser desbloqueado, o processo poderá saber qual das duas situações ocorreu pelo código de retorno da primitiva, de forma análoga ao caso da "envia". Também aqui, um valor "infinito" para o tempo de espera indica que o processo chamador da primitiva "recebe" só será liberado após a recepção da mensagem. Caso esta nunca chegue, o processo ficará bloqueado para sempre.

IV.4.2.3. Selec:

Um porto de entrada ligado a vários portos de saída permite que um processo receba (ou ao menos esteja apto a receber) mensagens de vários processos. Ocorre, porém, que, nesse tipo de estrutura, as mensagens oriundas desses processos são do mesmo tipo. Melhor explicando, quando vários processos clientes se comunicam com um processo servidor para obter um mesmo serviço, uma ligação de portos do tipo vários-para-um é ideal. Porém, quando os serviços são diferentes, deve-se esperar mensagens de tipos e tamanhos também distintos.

Mensagens de tipos diferentes podem ser enviadas por processos que requerem serviços distintos. Esses serviços podem ser específicos para cada processo ou podem atender grupos diferentes formados a partir de processos que demandam o mesmo serviço. Mensagens de tipos diferentes podem ter tamanhos diferentes. Assim, a única forma de se utilizar apenas uma chamada à primitiva recebe em um esquema de ligação de portos do tipo vários-para-um é

através da alocação de um "buffer" cujo tamanho seja suficiente para receber a mensagem de maior tamanho dentre as possíveis. Entretanto, apesar de possível, este não é um esquema elegante para ser usado em um sistema de comunicação baseado em portos.

Um processo pode ser dono de vários portos de entrada e de saída. Funcionalmente, cada tipo de mensagem deve ser associado a um porto diferente. Dessa forma, um processo escreve mensagens diferentes em portos de saída diferentes e lê de um mesmo porto de entrada apenas mensagens de um mesmo tipo, mesmo que enviadas por processos diversos. Quando um processo desejar ler mensagens de tipos diferentes, ele deverá fazê-lo através de portos distintos.

No PORTOS-TF, em uma chamada à primitiva "recebe" só é possível se especificar um único porto de entrada. Em conseqüência, para receber mensagens através de mais de um porto, devem ser feitas mais de uma chamada à referida primitiva. Se essas chamadas forem independentes entre si, ou seja, forem feitas em pontos distintos do processo, poderão ser empregadas da forma como disposto no ítem IV.4.2.2. (que descreve a primitiva "recebe"). Contudo, se forem chamadas em um mesmo ponto do processo, devem ser incluídas em uma estrutura de "seleção". Essa estrutura permite que mais de uma chamada à primitiva "recebe" seja realizada sem que alguma delas provoque o bloqueio do processo. (Na forma simples, caso haja cláusula de tempo em alguma primitiva e não exista mensagem à disposição, o processo ficará bloqueado retardando, ou mesmo impedindo, a execução das primitivas posteriores.)

Algumas linguagens de programação concorrente, como ADA (MENDES, 1984) e CONIC (SLOMAN, KRAMER & MAGEE, 1986) incorporam essa estrutura de seleção, como ilustrado no exemplo de SLOMAN, KRAMER & MAGEE (1986):

```
loop
    select
        receive signal from clock
        => with reading do
              begin value := readsensor();
              if value > limit then begin
                  state := true;
                  send reading to alarm;
              end;
           end;
    or
        receive signal from query reply reading;
    end;
end;
```

Figura IV.6 : Exemplo de estrutura de seleção baseada em construções de linguagem

No PORTOS-TF, esse tipo de estrutura foi implementado através da criação de três primitivas: "selec", descrita a seguir, e "fim_selec" e "porto_ultima_mensagem", descritas nos próximos ítens.

A primitiva "selec", quando evocada, liga um "flag" do sistema operacional que impede com que uma primitiva "recebe" bloqueie o processo chamador. Nada mais é alterado na rotina original. Dessa maneira, pode-se ter a seguinte forma:

```
selec( );
    recebe( 1, mens1, tam_mens1, SEM_ESPERA );
    recebe( 2, mens2, tam_mens2, SEM_ESPERA );
fim_selec( SEM_ESPERA );
```

Figura IV.7 : Exemplo de estrutura de seleção no PORTOS-TF

IV.4.2.4. Fim_Selec:

A primitiva "fim_selec" é usada para terminar um bloco de seleção. Como parâmetro dessa rotina, é passado o tempo máximo pelo qual o processo ficará bloqueado esperando por uma mensagem. Quando for recebida uma mensagem em qualquer um dos portos especificados nas chamadas às primitivas "recebe" do bloco de seleção, o processo será desbloqueado. Se o tempo de espera se esgotar, então o processo também será liberado, mas a primitiva retornará o valor que indica que o tempo de espera se esgotou. Caso haja alguma mensagem disponível antes do inicio da execução do bloco de seleção, então o processo não chegará a ficar bloqueado.

IV.4.2.5. Porto_Ultima_Mensagem:

Quando um processo deixa uma região de seleção, é fundamental conhecer-se a razão pela qual ele o fez. Pelo código de retorno da primitiva "fim_selec" pode se saber se o tempo de espera se esgotou ou se uma mensagem (pelo menos uma) foi recebida. Caso tenha ocorrido a segunda hipótese, é muito provável que o caminho pelo qual seguirá o processo dependa do porto por onde chegou a mensagem (que indiretamente reflete o tipo do serviço a ser prestado). O processo pode obter tal informação através da chamada da primitiva "porto_ultima_mensagem".

Dessa forma, o conjunto "selec"-"fim_selec"-"porto_ultima_mensagem" implementa, na forma de primitivas a estrutura de "região de seleção" que, como mostrado no ítem IV.4.2.3, é encontrado, em outros sistemas, na forma de construções de linguagem.

IV.4.2.6. Responde:

A última primitiva de comunicação, de nome "responde", escreve em um porto de entrada, cujo número é passado como parâmetro, uma mensagem de resposta ao último pedido recebido por aquele porto. Seu endereço é o segundo parâmetro da função. O tamanho dessa resposta também se constitui em um parâmetro da primitiva.

Só faz sentido essa primitiva ser assíncrona. Logo a rotina "responde" é sempre do tipo não-bloqueante.

Quando a resposta é escrita no porto, é verificado se o processo que enviou a última mensagem para aquele porto está esperando por uma resposta. Caso esteja, então a mensagem será copiada diretamente no o "buffer" correspondente do processo e esse será liberado. Caso contrário, nenhuma ação será tomada. Há duas interpretações para o fato do processo remetente não estar esperando uma resposta. A primeira hipótese é que a primitiva envia usada para remeter a mensagem era assíncrona (parâmetro "tempo de espera" igual a zero). Nesse caso, a arquitetura dos processos comunicantes provavelmente estará incorreta. A outra hipótese é que o porto de saída correlato esteja ligado a vários de entrada. Nesse caso, se a primitiva "envia" era do tipo síncrono, o que ocorreu é que um dos outros processos ligados ao remetente já enviou a resposta. Esse esquema permite o acesso a servidores múltiplos de forma transparente ao processo cliente.

CAPÍTULO V

GERÊNCIA DE MEMÓRIA

V.1. Introdução:

Um programa faz uso de variáveis para armazenar informações que serão necessárias na sua execução. Ocorre, porém, que nem todas elas necessitam ou têm como ficar na memória do início ao fim da execução do programa. Algumas são geradas em um determinado instante da execução e utilizadas por algum tempo, após o qual se tornam desnecessárias. Este é, normalmente, o caso de elementos de estruturas de dados como listas, árvores, etc (HOROWITZ & SAHNI, 1984).

Essas estruturas não são criadas e utilizadas todas ao mesmo tempo. Dessa forma, pode-se otimizar o uso da memória compartilhando-a entre tais estruturas, pois as posições de memória necessárias para armazená-las só precisam estar a elas atribuídas enquanto o seu conteúdo for utilizado pelo processo.

Portanto, para reduzir o tamanho de sua área de dados, o sistema operacional deve permitir a **alocação e desalocação** dinâmica de memória. As primitivas que implementam essas funções são utilizadas na alocação e desalocação dinâmica de estruturas como, por exemplo, descritores de processo (vide capítulo III), "buffers" de comunicação (vide capítulo IV), nós de árvore (vide capítulo VI), etc.

Esses "blocos" de memória que são alocados dinamicamente estão em uma área da memória expecificamente reservada para este fim. Na hora de se alocar um novo bloco, deve-se escolher um dentre esses blocos livres.

Segundo DEITEL (1984) e MADNICK & DONOVAN (1974), existem três políticas básicas para a seleção de qual bloco de memória alocar. São elas:

. "first-fit": o que primeiro couber;
. "best-fit": o que melhor couber e
. "worst-fit": o que pior couber.

Em todas elas, naturalmente, o bloco escolhido deve ter tamanho igual ou maior ao requisitado.

A política de "first-fit" diz que a partição a ser ocupada será a primeira cujo tamanho for igual ou superior ao desejado. A política de "best-fit" será escolhida a partição cujo tamanho melhor se ajuste ao desejado, isto é, a de menor tamanho dentre as de tamanho maior ou igual ao desejado. A terceira poítica, a de "worst-fit", caracteriza-se por eleger a partição de maior tamanho dentre todas.

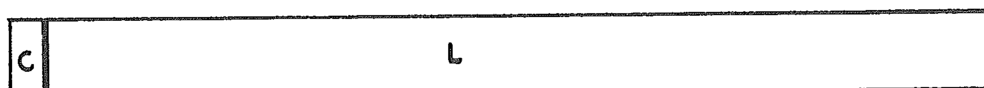
V.2. Opções de Projeto:

Para a implementação das primitivas de alocação e desalocação dinâmica de memória, o PORTOS-TF adota um "buffer pool", que é onde ficam todas as áreas dinâmicas. Esse "buffer", inicializado com toda a sua área disponível (figura V.1a), será dividido, por demanda, em blocos (figuras V.1b e V.1c). Esse "pool" assume um tamanho predefinido que, para algumas aplicações, pode ser insuficiente. Nesse caso, pode-se, ao instalar o sistema operacional, passar um parâmetro especificando um tamanho superior ao original.

Quando é chamada a primitiva de alocação de memória, a escolha do bloco a ser alocado é feita de acordo com a política de "first-fit" (primeiro que couber). Essa política foi adotada visando dois aspectos:

. tempo e

. fragmentação.



(a)

(b)

(c)

LEGENDA: C = CABEÇALHO
L = ÁREA LIVRE
O = ÁREA OCUPADA (ALOCADA)

Figura V.1 : Exemplo de alocação de áreas de memória

Como se sabe, a política de "first-fit" é a que decide mais rapidamente qual bloco será o escolhido (DEITEL, 1984). Além disso, ela favorece a utilização de blocos localizados no início do "buffer pool" (MADNICK & DONOVAN, 1974).

A política de "best-fit", além de ser mais demorada para a escolha do bloco a alocar, aumenta também a fragmentação do "buffer pool" (MADNICK & DONOVAN, 1974).

A política de "worst-fit" foi preterida não só pelo tempo que leva para a seleção do bloco a alocar, mas também por dividir os grandes blocos. Se é verdade, por um lado, que ao se dividir um grande bloco, alocando-se

uma parte, a parte restante terá ainda um tamanho grande o suficiente para ser útil (DEITEL, 1984) (ao contrário da política de "best-fit"), também é verdade, por outro lado, que, após algum tempo, esses blocos poderiam ser incapazes de acomodar algumas mensagens.

Essas hipóteses relativas aos problemas de fragmentação são muito dependentes do estado corrente do "buffer pool" e da seqüência das chamadas de alocação e desalocação. Esse perfil é muito difícil de ser estimado. Portanto, apesar do ítem "fragmentação" ter sido considerado, o fator "tempo" foi determinante na escolha da política de "first-fit" para a alocação de blocos do "buffer pool".

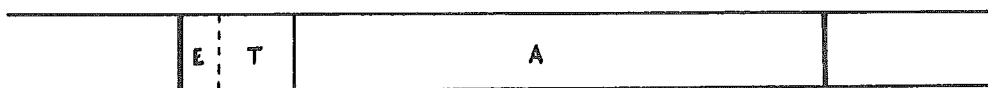
Vale a pena chamar a atenção de que não é possível fazer a compactação dos blocos pois, dentro do "hardware padrão" adotado, não há como manter ponteiros e endereços atualizados. (MADNICK & DONOVAN (1974) abordam a questão da relocação de blocos de memória e o hardware utilizado nesse caso).

V.3. Aspectos da Implementação:

Como ilustra a figura (V.2), cada um dos blocos do "buffer pool" é composto por:

. um cabeçalho que contém o tamanho do bloco e o estado do bloco (disponível ou alocado) e
. a área de trabalho do bloco.

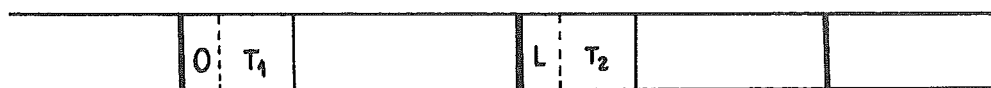
Uma vez escolhido o bloco a ser alocado, este será divido em dois. O primeiro pedaço será marcado como "alocado" e terá o tamanho igual ao requisitado. O segundo pedaço constituirá um novo bloco "disponível" e terá um tamanho igual ao anterior decrescrido do trecho alocado e de seu próprio cabeçalho (figura V.3a).



LEGENDA : E = ESTADO (LIVRE OU OCUPADO)
T : TAMANHO
A : ÁREA DE TRABALHO

Figura V.2 : Esquema de um elemento do "buffer pool"

(a)



(b)

LEGENDA : ESTADOS { L = LIVRE, O = OCUPADO }

TAMANHOS : $T = T_1 + T_2$ + TAMANHO DE CABEÇALHO

Figura V.3 : Esquema de desmembramento de bloco de memória

Ocorre, porém, que se a área do bloco escolhido para alocação for apenas um pouco (alguns bytes) maior do que a requisitada, duas situações podem se caracterizar, no que diz respeito à area restante:

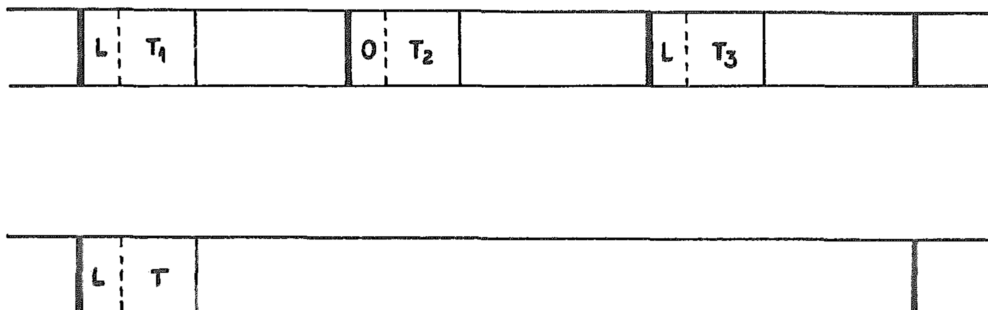
. área insuficiente para um novo bloco e
. área inútil para um novo bloco.

No primeiro caso, a área restante (isto é, a diferença entre a área disponível e a requisitada) pode ser tão pequena que nela não seja possível montar-se um

cabeçalho. No segundo, apesar de ser possível criar-se um novo bloco, este terá um tamanho tão diminuto que muito dificilmente conseguirá satisfazer algum outro pedido de alocação.

Para resolver esta questão, o PORTOS-TF estabelece um tamanho mínimo para um bloco. Quando um bloco é alocado, se o tamanho da área restante for maior ou igual a *esse mínimo*, então o procedimento de desmembramento descrito anteriormente será efetuado. Caso contrário, não haverá desmembramento e o bloco será todo alocado, sobrando então uma folga (figura V.3b). (Na implementação em equipamentos do tipo IBM-PC, o tamanho de bloco mínimo adotado foi de quatro bytes.)

Como mencionado anteriormente, não é possível compactar-se os blocos "disponíveis". Todavia, quando um bloco é liberado, examina-se seus adjacentes. Se um deles ou ambos estiverem disponíveis, haverá uma fusão dos blocos, diminuindo assim a fragmentação do "buffer pool" (figura V.4). Dessa forma pode-se afirmar que não existem dois blocos disponíveis adjacentes no PORTOS-TF.



LEGENDA: ESTADOS { L = LIVRE; O = OCUPADO

TAMANHOS: $T = T_1 + T_2 + T_3 + 2 \times$ TAMANHO DE CABEÇALHO

Figura V.4 : Esquema de fusão de blocos de memória

Para alocar-se uma área de memória, deve-se chamar a função de alocação passando como parâmetro o tamanho da área desejada. Será retornado pela função um ponteiro para o início da área livre (imediatamente após o cabeçalho). Na desalocação, este ponteiro deve ser passado como parâmetro. A rotina, então, retrocederá para o cabeçalho, decrementando o ponteiro no número de bytes que o compõe, promovendo em seguida a liberação do bloco e sua eventual fusão com blocos adjacentes.

CAPÍTULO VI

ESTRUTURAS DE DADOS

VI.1. Introdução:

As estruturas de dados empregadas em um sistema operacional (lista de descritores de processos, lista de temporizações, etc.) são de grande importância para a flexibilidade e desempenho desse sistema.

Essas estruturas podem ser organizadas em tabelas (ou vetores ou, ainda, arranjos), listas simples, circulares, árvores, etc. (HOROWITZ & SAHNI, 1984).

Tabelas são a forma mais natural de se manter e manusear essas estruturas (figura VI.1). Isto porque para se referenciar a qualquer uma delas são empregados índices. Para navegar através de uma tabela, isto é, mover-se pelos registros para a frente ou para trás, basta incrementar ou decrementar um índice.

	A	B	C
1			
2			
3			
4			
5			
6			

Figura VI.1 : Exemplo de tabela

A desvantagem das tabelas está na sua estrutura estática. Por ter um tamanho pré-determinado seu emprego pode por um lado limitar o número de registros armazenados e por outro ocasionar o desperdício de memória.

Uma outra forma de armazenar os dados do sistema operacional é através de listas encadeadas (figura VI.2). Uma lista encadeada, ou simplesmente lista, pode começar com o tamanho zero e, à medida que vão surgindo novos registros a serem guardados, novos nós vão sendo criados e incluídos na lista. Analogamente, quando uma determinada informação não é mais necessária, seu nó correspondente é retirado da lista e a área por ele ocupada é liberada.

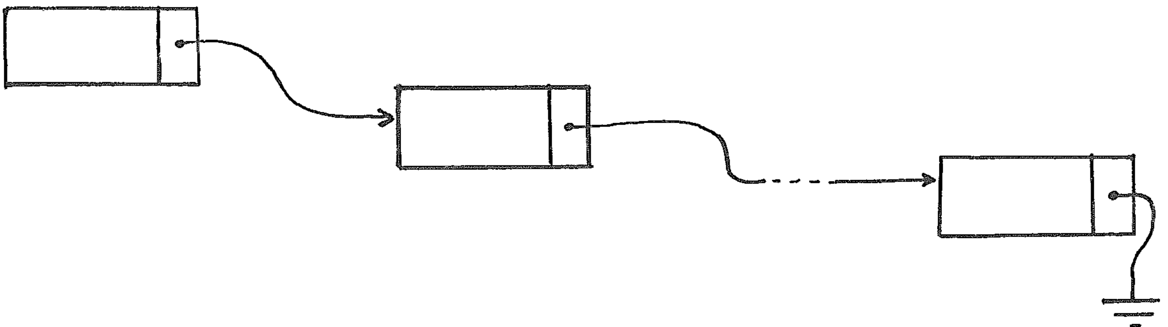


Figura VI.2 : Exemplo de lista encadeada

As listas podem ser:

- abertas e simplesmente encadeadas;
- abertas e duplamente encadeadas;
- circulares e simplesmente encadeadas e
- circulares e duplamente encadeadas.

A escolha entre lista aberta e lista fechada depende basicamente do perfil da aplicação. Quando o acesso aos nós da lista é randômico (isto é, não segue nenhuma seqüência) ou seqüencial finito (isto é, cada nó é acessado após seu antecessor sucessivamente até o último

nó), o uso de listas abertas é mais conveniente, por ser mais simples de gerenciar. Por outro lado, se o acesso aos nós da lista for seqüencial infinito (ou seja, cada nó é acessado após seu antecessor sucessiva e indefinidamente, onde o primeiro é acessado após o último), então será justificada a adoção de uma lista circular, onde o último elemento aponta para o primeiro.

A adoção de listas duplamente encadedadas simplifica os algorítmos de busca com a finalidade de inserção ou retirada do nó, já que de um nó se sabe seu antecessor e seu sucessor. Listas simplesmente encadeadas levam vantagem sobre as duplamente encadeadas por ocuparem menos memória, além de necessitarem atualizar um menor número de ponteiros pois cada nó só usa um, e não dois ponteiros, para se ligar a seus adjacentes. Por outro lado, na busca da posição de inserção ou retirada de um nó, deve ser sempre guardada a posição do nó antecessor, o que torna o algoritmo um pouco mais complexo.

Uma outra estrutura de dados extremamente importante é a "árvore" (figura VI.3). Segundo HOROWITZ & SAHNI (1984), "árvore é um conjunto finito de um ou mais nós de tal natureza que: (i) existe um nó especialmente designado, de nome raiz; (ii) os nós restantes estão desdobrados em "n" conjuntos ("n" >= 0) separados T1, etc., Tn, em que cada um dos referidos conjuntos se constitui em uma árvore. T1, etc., Tn se denominam subárvores da raiz." Os nós que são as raízes das subárvores T1, etc., Tn são chamados "filhos" da raiz da árvore. O nó da raiz da árvore, por sua vez, é o "pai" das raízes das subárvores. Filhos de um mesmo pai são "irmãos". Um nó de uma árvore que não possui filhos é chamado de "folha".

"Altura" de uma árvore pode ser definida como o número mínimo de nós que se deve percorrer para ir da raiz até uma folha qualquer, contando a raiz e a folha.

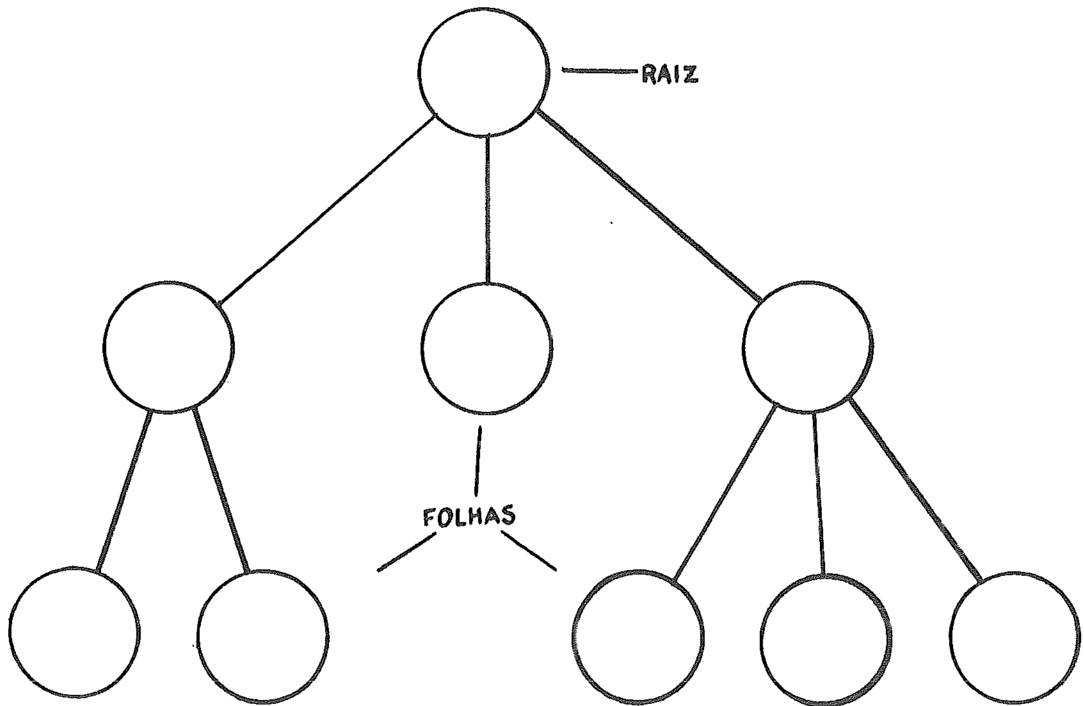


Figura VI.3 : Exemplo de árvore

Uma árvore notável e de grande aplicação prática é a chamada "árvore binária". Diz-se que uma árvore é binária quando sua raiz tem, no máximo, dois filhos, sendo cada um deles raiz de uma árvore também binária (figura VI.4). Uma árvore vazia também é uma árvore binária.

Em 1962, Adelson-Velskii & Landis introduziram um novo conceito de árvore: a "árvore binária balanceada". Define-se árvore binária balanceada como sendo uma árvore binária não vazia onde cada filho de sua raiz é raiz de uma árvore binária também balanceada e onde a diferença em módulo entre as alturas das subárvores da direita e da esquerda da raiz é menor ou igual a um. A árvore vazia também é uma árvore binária balanceada.

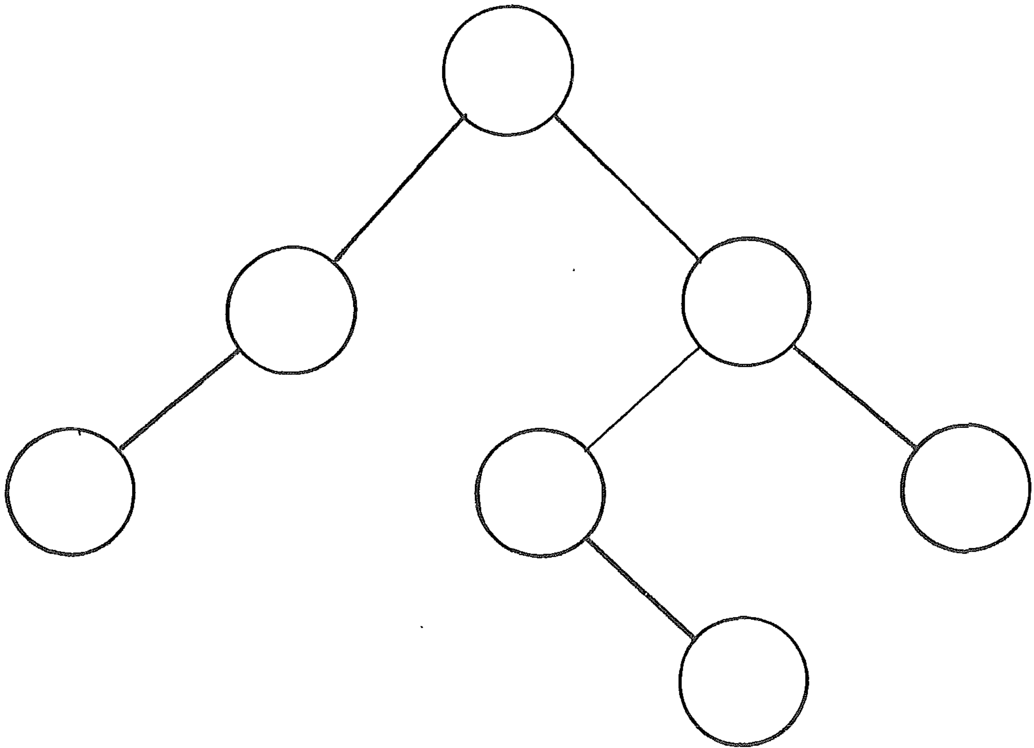


Figura VI.4 : Exemplo de árvore binária

Árvores são estruturas mais complexas de serem mantidas do que listas. Como estas, as árvores também são estruturas de tamanho flexível, ao contrário de tabelas. Sua grande vantagem sobre listas está no tempo de busca, inserção e retirada de um determinado nó. Pode-se provar que para achar um elemento em uma lista, um algoritmo ótimo é O(n), onde n e´o número de elementos da lista. Já para uma árvore binária balanceada, um algoritmo ótimo é O(log n).

VI.2. Opções de Projeto:

Há duas estruturas de dados (ou tipos de estruturas) que devem ser comentadas por serem fundamentais para o entendimento dos algoritmos e funções que os manuseiam. São elas: as listas de processos e a estrutura do descritor de processo (DP).

VI.2.1. "Listas" do Sistema Operacional:

No PORTOS-TF todas as tabelas e listas foram organizadas de forma hierárquica. Para tanto foi adotada a estrutura em árvore binária balanceada em conjunto com listas circulares. A utilização desse tipo de estrutura tem dois objetivos principais:

. rapidez de acesso a um determinado nó da estrutura e
. degradação lenta com o aumento do número de processos.

É sabido que o gerenciamento de árvores envolve algoritmos mais trabalhosos do que os de filas simples. Dessa forma, é de se esperar que, para um pequeno número de processos, a adoção de estruturas em árvore seja pior do que em filas simples. Por outro lado, como algoritmos de busca, inserção e retirada de árvores binárias balanceadas são O(log n), contra algoritmos O(n) para listas simples, a organização das estruturas do sistema em árvores faz com que a degradação do sistema, com o aumento do número de estruturas, seja mais suave.

A forma geral dessa estrutura citada anteriormente está representada na figura (VI.5). Nela, cada nó da árvore é dividido em cinco campos:

. chave:
  tem como função identificar o nó para sua classificação;
. nível:
  identifica o nível do nó na árvore. É sempre uma unidade maior do que o maior nível dentre os dos filhos. É usado no balanceamento da árvore;
. ponteiro para lista circular:
  aponta para uma lista circular que contém os elementos com a mesma chave.
. ponteiro para o filho da esquerda:
  aponta para a subárvore da esquerda e

. ponteiro para o filho da direita:
aponta para a subárvore da direita.

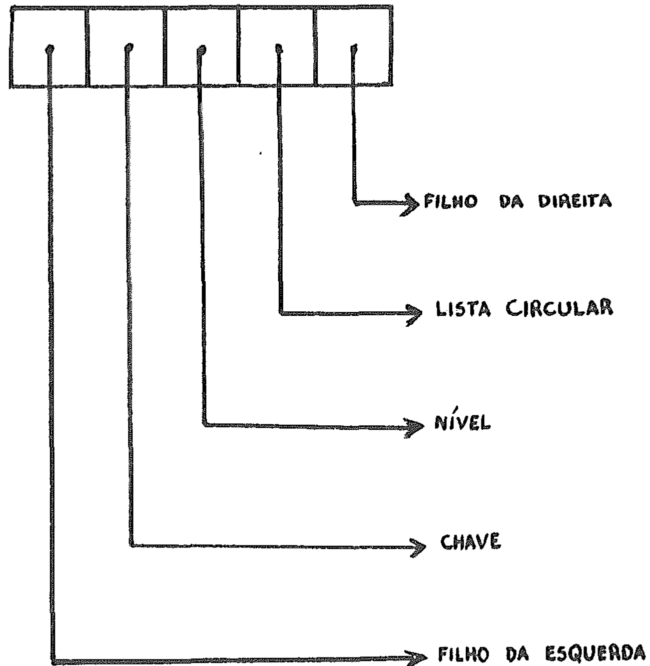


Figura VI.5 : Estrutura de um nó das árvores empregadas no PORTOS-TF

Na árvore, a chave do filho da esquerda é sempre menor do que a do pai que, por sua vez, é menor do que a chave do filho da direita. Elementos de mesma chave são inseridos no final da lista circular associada àquele nó da árvore.

Cada elemento da lista circular é composto de dois campos:

. ponteiro para DP:
aponta para um Descritor de Processos e

. próximo:
ponteiro para o próximo nó da lista.

O fato dessa lista circular compor-se de nós ponteiros para Descritores de Processos e não dos próprios Descritores, que poderiam ser encadeados uns aos outros,

explica-se pelo fato de se desejar fazer essa estrutura genérica, de forma a tornar todas as rotinas de manipulação associadas a esse tipo de estrutura comuns a todas as árvores empregadas no sistema operacional.

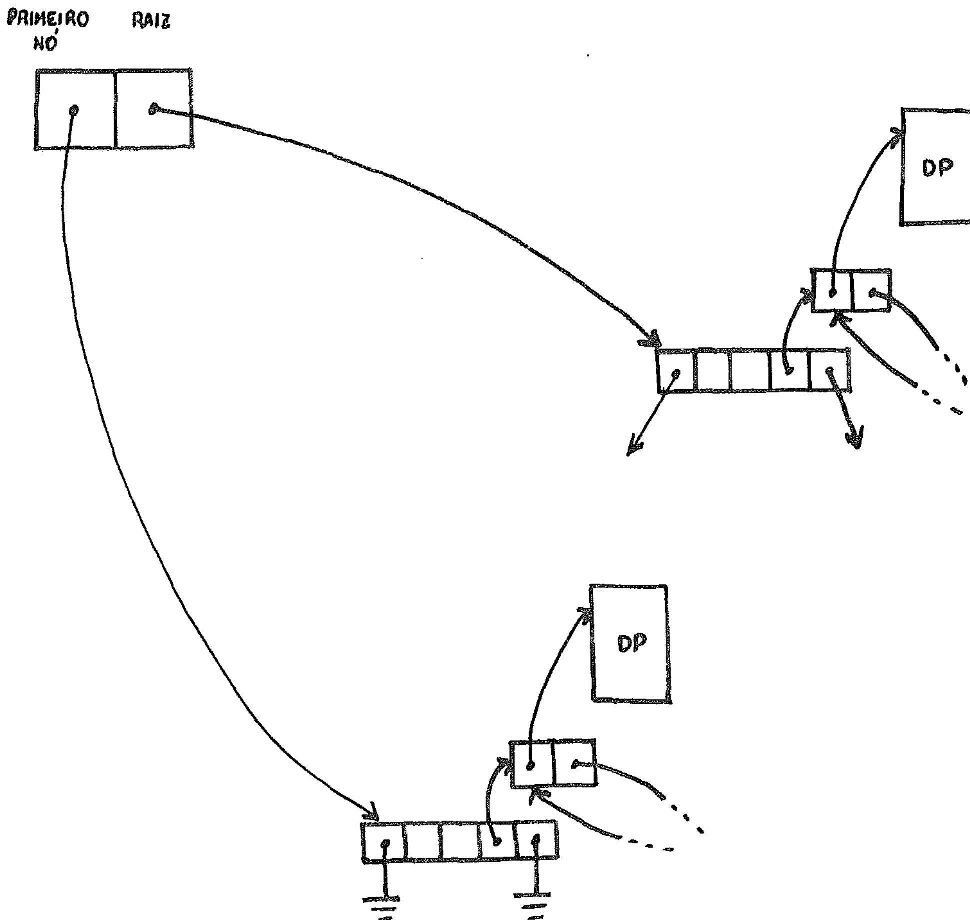


Figura VI.6 : Estrutura geral das "listas" do PORTOS-TF

Essa estrutura descrita foi montada com o objetivo de atender a três "listas" do sistema operacional (ver capítulo III):

. a lista dos descritores de processos;
. a lista dos processos no estado PRONTO e
. a lista de processos bloqueados por tempo.

A disposição de elementos de mesma chave em uma lista circular é conveniente no caso da "Lista dos

Processos Prontos", na ocasião do escalonamento.

Como é necessário acessar-se com grande freqüência a primeira posição (a de menor chave) de algumas dessas "listas", a cada uma delas está associado um descritor. Este descritor contém dois campos:

. ponteiro para a raiz da árvore e
. ponteiro para o primeiro nó da árvore.

A figura (VI.6) ilustra a estrutura completa aqui exposta.

VI.2.2. O Descritor de Processos:

O descritor de processos (DP) "é uma estrutura de dados que contém certas informações importantes sobre o processo" (DEITEL, 1984). No PORTOS-TF, essas informações são as seguintes (vide capítulo III):

.identificador do processo;
.prioridade;
.estado;
.tamanho da fatia de tempo;
.tempo de espera;
.endereço da rotina de serviço;
.endereço da pilha e
.lista dos portos.

O identificador do processo, a prioridade, o tamanho da fatia de tempo e o tempo de espera são valores numéricos que variam de 1 até um valor máximo. Esse valor máximo depende da máquina. Na implementação para equipamentos do tipo IBM-PC, por exemplo, o valor máximo desses campos é FFFFh, com exceção da prioridade que é 255, decimal. Os processos mais prioritários são os de prioridade igual a 1. Os de menor prioridade são os que têm o valor 255 como, por exemplo, o processo "ocioso". O estado dos processos pode ter dois valores: "pronto" e

"bloqueado". O processo "rodando" mantém a indicação de "pronto" em seu DP pois não necessita alterá-la, uma vez que sua identificação é feita pela manutenção do endereço do DP corrente em uma variável do sistema operacional. Um tempo de espera igual a zero indica que o processo não está bloqueado, enquanto que um tempo igual ao valor máximo indica que ele só será liberado por evento e não por tempo. Um outro valor qualquer implica na contagem de um tempo máximo de bloqueio. Esse valor é decrementado de uma unidade a cada pulso do relógio. O campo "endereço da rotina de serviço" traz o endereço absoluto da rotina que deverá ser executada caso o processo seja desbloqueado por estouro do tempo de espera. Essa rotina será executada antes do processo retomar sua execução. O endereço do topo da pilha do processo no momento do escalonamento é também guardado no DP.

O último elemento do DP é um ponteiro para uma lista que contém a relação dos portos empregados pelo processo (vide capítulo IV). Esta é uma lista linear e simplesmente encadeada, composta por elementos que têm a seguinte estrutura:

.número;
.sentido;
.tipo;
.lista dos portos aos quais está ligado;
.lista das mensagens recebidas;
.endereço do "buffer" da mensagem seguinte;
.tamanho desse "buffer" e
.próximo nó.

O número do porto varia de 1 até um valor máximo. Esse valor máximo também é dependente da máquina. Na implementação para equipamentos do tipo IBM-PC, por exemplo, ele é igual a FFFFh. O sentido de um porto pode ser de entrada ou de saída, enquanto que seu tipo pode ser "confiável" ou "não-confiável" (vide capítulo VII).

A lista dos portos aos quais está ligado é uma lista linear e simplesmente encadeada. Ela contém, em cada nó, o número de um processo e seu respectivo porto ao qual o porto em questão está ligado. Essa lista contém um nó de cabeçalho que, além de simplificar os algoritmos de inserção e retirada, armazena a identificação do porto externo que enviou a última mensagem ao porto em questão. Essa informação é utilizada pela primitiva "responde".

A lista de mensagens também é uma lista linear e simplesmente encadeada. Ela contém, em cada um de seus elementos, o endereço e o tamanho do "buffer" do sistema operacional onde está armazenada uma mensagem já enviada para o porto mas ainda não consumida por ele. Além dessas informações, ele também armazena o identificador desse porto de saída.

Os campos endereço e tamanho do "buffer" da mensagem seguinte trazem o endereço absoluto e o tamanho do "buffer" que o processo dono do porto alocou para receber a próxima mensagem. Um endereço nulo indica que o processo não está esperando nenhuma mensagem por aquele porto de entrada. Um endereço não nulo implica, obrigatoriamente, em uma lista de mensagens vazia, pois significaria um erro o processo estar aguardando uma mensagem quando já existisse pelo menos uma a ser consumida.

O último campo aponta para o próximo nó da lista.

VI.3. Aspectos da Implementação:

Pela característica evidente de recursividade das árvores (vide sua própria definição), as rotinas relativas às árvores e listas foram, em quase sua

totalidade, implementadas através de chamadas recursivas. Destacam-se entre elas as rotinas de inserção, obtenção e retirada de elementos das "listas".

Por questões de simplificação dos algoritmos de inserção e retirada de nós das listas circulares, o campo do nó da árvore que aponta para a lista não indica o primeiro (ou próximo) nó dessa lista, mas sim para o último. Isso se justifica pelo fato de ser necessário atualizar-se o ponteiro para o próximo nó do nó anterior àquele que se quer inserir ou retirar.

Dessa forma, o primeiro nó da lista circular é aquele que é o próximo do que está sendo apontado pelo nó da árvore. No caso particular de haver apenas um nó na lista, este algoritmo continua válido pois o próximo nó daquele apontado pelo nó da árvore é ele próprio (figura VI.7).

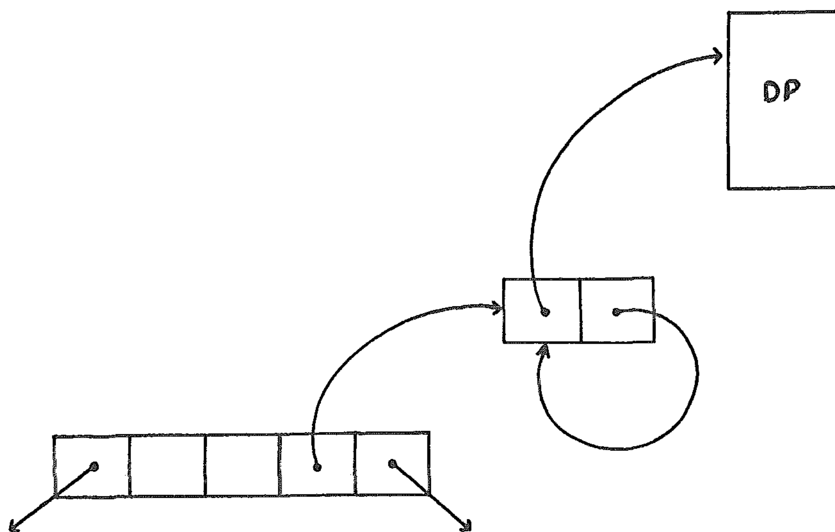


Figura VI.7 : Esquema de um nó de "lista" do PORTOS-TF com apenas um elemento

Não existe a possibilidade da lista circular associada a um nó da árvore ser vazia, pois nesse caso,

esse nó da árvore seria retirado da estrutura.

A inserção de elementos nas árvores é feita de forma recursiva. Para inserir um novo elemento em uma árvore, compara-se a chave do elemento com a da raiz da árvore. Se a chave do elemento for menor do que a da raiz árvore, então ele será inserido na subárvore da esquerda. Caso seja maior, ele o será na subárvore da direita. Caso seja igual, será inserido no último lugar da lista circular associada ao nó raiz da árvore. Não há crítica, nessa rotina, de duplicidade de elementos. Foi feita a opção de deixar esse controle por conta das rotinas do núcleo que irão utilizar as árvores.

Caso se tenha decidido inserir o novo elemento em uma das subárvores, o procedimento de inserção será novamente chamado, de forma recursiva, até ser encontrado o ponto de inserção. Esse ponto pode ser um nó raiz de mesma chave do elemento a ser inserido ou um nó raiz vazio. No primeiro caso, ocorre apenas a inserção de um novo nó na lista circular associada a esse raiz. No segundo, é criado e inserido nesse ponto um novo nó de árvore. A lista circular associada a esse nó terá como único elemento aquele que se está inserindo.

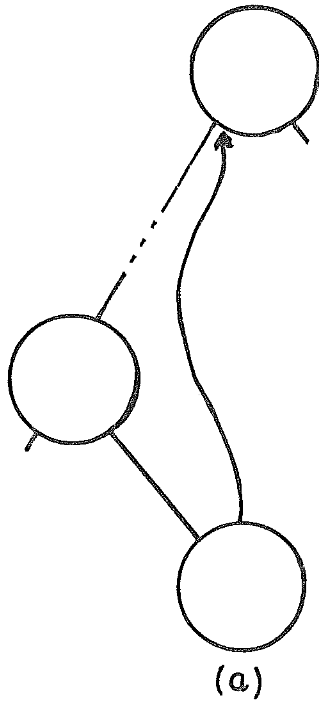
Após o retorno de uma chamada recursiva, pode-se afirmar que o elemento foi inserido na estrutura. Contudo, essa inserção pode ter causado o desbalanceamento da árvore. Portanto, o nível da raiz é atualizado e é chamado o procedimento de balanceamento da árvore. Esse procedimento será comentado mais adiante.

Cabe salientar que um novo elemento só pode ser inserido (1) em uma lista circular de um nó da árvore já existente ou (2) em uma folha da árvore. Um novo elemento jamais será inserido entre dois nós (pai e filho) de uma árvore.

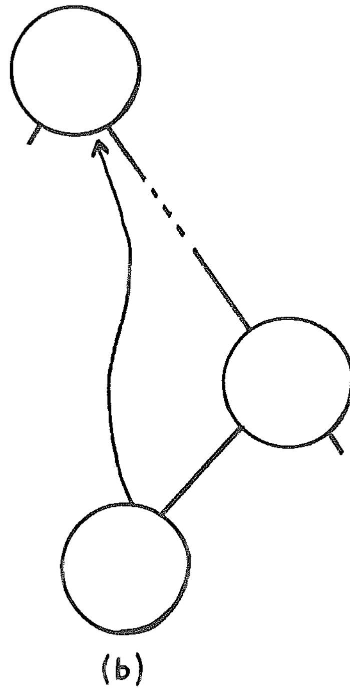
O procedimento de retirada de um nó da árvore é similar ao de inserção no que diz respeito ao processo recursivo de procura do nó. No entanto, existe uma diferença marcante entre os dois algoritmos. Na retirada, um nó pode ser retirado do "meio" da árvore, isto é, o nó retirado pode ter filhos, enquanto que na inserção essa situação é impossível de ocorrer. Na hipótese de se retirar um nó do meio da árvore, o problema se resume à retirada da raiz de uma árvore, já que são usadas chamadas recursivas. Em seu lugar, é colocado o elemento imediatamente anterior (figura VI.8a) ou posterior (figura VI.8b) a ele na árvore. Essa escolha depende da altura das subárvores.

O elemento retirado deve pertencer à subárvore de maior altura. Dessa forma, garante-se que a árvore não ficará desbalanceada. Pode-se fazer essa afirmativa pelo fato de que quando se retira um nó de uma árvore ela pode ter sua altura reduzida em, no máximo, uma unidade. Se o elemento retirado for da subárvore de maior altura, pode-se afirmar que a diferença entre as alturas das duas subárvores da raiz continuará menor ou igual a um. Essa mesma afirmativa pode ser feita no caso das subárvores terem a mesma altura. Nesse caso a escolha de que subárvore será retirado o elemento é arbitrária.

Pelo exposto, pode-se garantir que não é necessário rebalancear-se a árvore cuja raiz foi retirada. Todavia, tanto a árvore da qual foi retirado o elemento substituto quanto a árvore original podem ter ficado desbalanceadas e, portanto, necessitarem ser rebalanceadas.



(a)



(b)

Figura VI.8 : Possibilidades de substituição de raiz de árvore binária

O núcleo do sistema operacional, na realidade, não chama diretamente essas rotinas de inserção e retirada, mas sim rotinas intermediárias que atualizam os descritores das listas após chamarem essas rotinas de inserção e retirada.

A questão mais complexa no que se refere a árvores binárias balanceadas é a manutenção do seu balanceamento.

HOROWITZ & SAHNI (1984) sugerem um algoritmo de inserção em árvore binária balanceada que mantém o balanceamento. Ocorre que também na retirada de um elemento da árvore, esta pode ficar desbalanceada. Logo, tanto na inserção quanto na retirada, deve-se identificar se houve desbalanceamento da árvore e, em caso afirmativo, efetuar o novo balanceamento.

No PORTOS-TF foi adotado um algoritmo que segue a mesma idéia do apresentado por Horowitz & Sahni.

VI.3.1. Algoritmo de Balanceamento de Árvore Binária:

O primeiro passo do algoritmo adotado consiste na identificação da situação de desbalanceamento. Por definição, uma árvore binária está balanceada se e somente se seus dois filhos são raizes de árvores binárias também balanceadas, cujas alturas não diferem, em módulo, em mais de uma unidade.

Pelo fato do algoritmo de inserção e do de retirada de nós da árvore serem recursivos, garante-se que, ao retornarem de uma chamada recursiva, a árvore cuja raiz foi passada como parâmetro estará balanceada. Essa árvore passada como parâmetro pode ser a subárvore da direita ou a da esquerda, dependendo do valor da chave sobre a qual se

está efetuando a operação. A causa de um eventual desbalanceamento fica então restrita à diferença de altura entre as duas subárvores.

De acordo com o exposto no ítem VI.2, o campo "nível" da raiz de uma árvore indica a altura dessa árvore. O algoritmo de verificação de balanceamento se vale dos campos "nível" dos filhos da raiz da árvore para determinar se a árvore está ou não balanceada.

No caso da árvore estar desbalanceada, existem duas hipóteses: ela estar desbalanceada à direita (isto é, a altura da árvore da direita ser duas unidades maior do que à da esquerda) ou à esquerda (isto é, a altura da árvore da esquerda ser duas unidades maior do que à da direita). Como o balanceamento é verificado a cada inserção ou retirada de um nó, a variação máxima na altura de uma das subárvores é de uma unidade. Logo, no algoritmo adotado, a diferença de altura entre duas subárvores não será nunca superior a duas unidades. Como o desbalanceamento de uma árvore em um sentido pode ser tratado de forma simétrica ao em outro, será abordado apenas o caso em que a árvore fica desbalanceada à direita.

Suponha-se, então, uma árvore "R" desbalanceada à direita. Esta árvore possui uma subárvore à esquerda "E" e uma à direita "D". A subárvore "D", por sua vez, tem uma subárvore à esquerda "DE" e uma à direita "DD" (figura VI.9). Para haver desbalanceamento à direita, "R", "D" e pelo menos uma entre as subárvores "DE" e "DD" são necessariamente não nulas. As demais podem o ser.

Seja, então, "h+1" a altura da árvore "R". Logo, como essa árvore está desbalanceada à direita, a subárvore da direita "D" terá uma altura "h" e a subárvore da esquerda "E" uma altura "h-2". Para reduzir a diferença entre as alturas das subárvores "D" e "E", sugere-se

aumentar a altura de "E" em uma unidade, pois como a diferença entre as alturas de "D" e "E" é de duas unidades esse aumento será o suficiente.

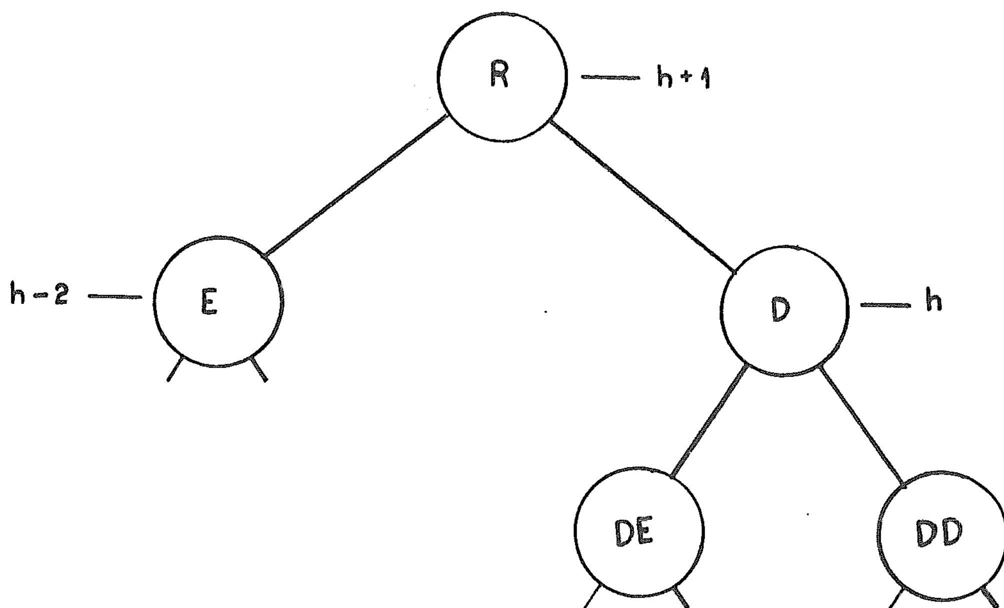
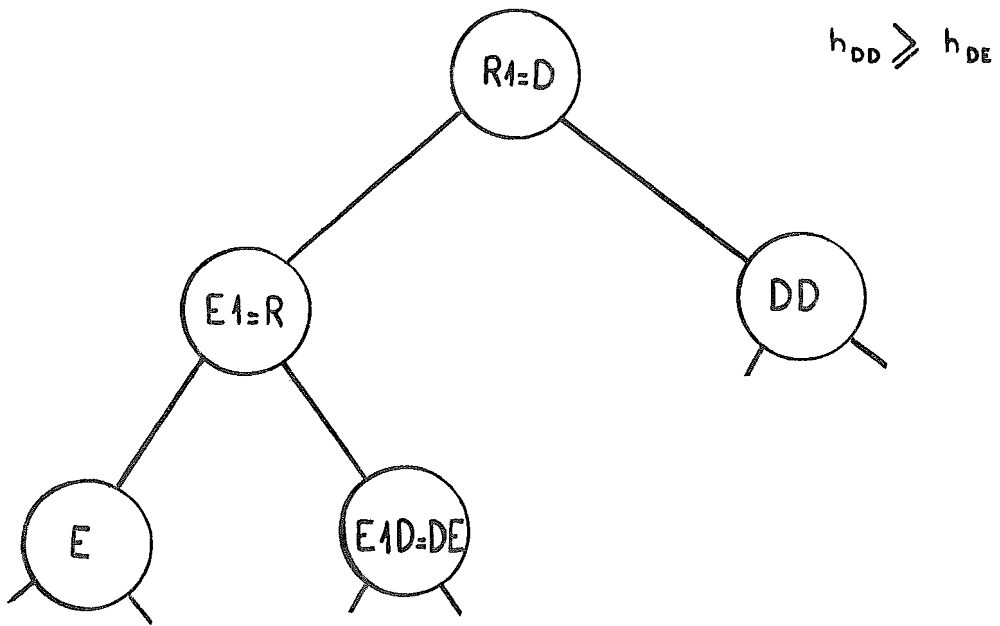


Figura VI.9 : Esquema de uma árvore binária desbalanceada à direita

Nenhum nó de "D" pode ser transferido para "E" porque todos eles têm chave maior que a raiz "R" e à sua esquerda só podem haver elementos de chaves menores. Logo, a única opção é criar uma nova árvore "R1" onde a subárvore da esquerda "E1" será formada pela raiz original "R" tendo "E" como subárvore da esquerda. Como subárvore da direita "E1D" será adotada uma subárvore de "D" cuja escolha, da mesma forma que "R1", depende ainda de outros fatores a serem abordados.

Se a altura de "DD" for maior ou igual à de "DE", então "D" será escolhida como a nova raiz "R1" tendo "E1" como subárvore da esquerda e "DD" como subárvore da direita. A subárvore "E1D", por sua vez, será a subárvore "DE" (figura VI.10a).



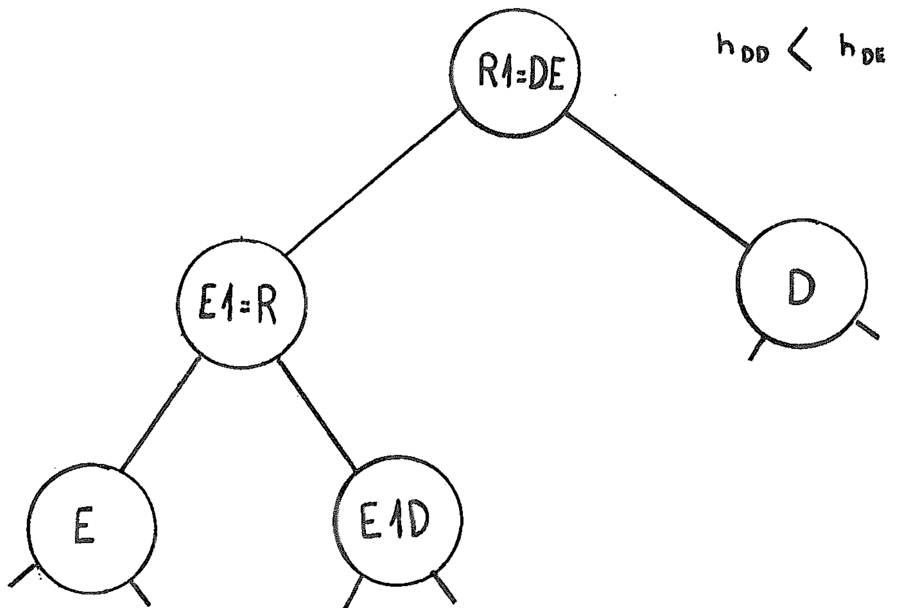


Figura VI.10 : Opções de balanceamento da árvore da figura VI.9

Se a altura de "DD" for maior do que a de "DE", então a altura de "E1" passará a ser "h-1" já que, da mesma forma que "E", a altura de "DE" também é "h-2", pois é menor do que a de "DD" e "D" é uma árvore balanceada. Como a altura de "DD" continua a mesma, então seu valor permanece sendo "h-1". Nesse caso, a árvore "R1" é balanceada de altura "h".

Se a altura de "DD" for igual à de "DE", então a altura de "E1" passará a ser "h" já que "E" continuará com altura "h-2" e "DE" com altura "h-1", pois não tiveram suas estruturas alteradas. Como a altura de "DD" também não se modificou, então continua sendo "h-1". Nesse caso, a árvore "R1" é balanceada de altura "h+1".

Porém, se a altura de "DE" for maior do que a de "DD", então a nova raiz "R1" escolhida será "DE" tendo "E1" como subárvore da esquerda e "D" como subárvore da direita. A subárvore "E1D", nesse caso, será a subárvore da esquerda de "DE". A nova subárvore da esquerda de "D" passará então a ser a antiga subárvore da direita de "DE" (figura VI.10b).

A altura de "E1", então passará a ser "h-1" já que a altura de "E" é "h-2" e a de "E1D" é "h-2" ou "h-3" por ser inferior a "DE", mas não mais do que duas unidades, uma vez que "DE" é uma árvore balanceada. Como a altura de "DD", não mudou, continua sendo "h-2", pois era menor duas unidades do que "D". A subárvore da direita de "DE" tinha altura "h-2" ou "h-3". Logo, a altura de "D" passa a ser "h-1". Nesse caso, a árvore "R1" é balanceada de altura "h".

CAPÍTULO VII

TOLERÂNCIA A FALHAS

VII.1. Introdução:

Um sistema de computação é composto por vários módulos de hardware e software. Por mais bem projetados e testados que sejam, existe sempre a possibilidade de que venham a apresentar um mau funcionamento. Segundo ANDERSON & LEE (1982), em concordância com MELLIAR-SMITH & RANDELL (1977), um sistema consiste de um conjunto de componentes que interagem de acordo com um projeto. Componentes podem ser encarados como sendo sistemas menores, ou subsistemas.

O hardware, como qualquer elemento físico, pode manifestar um comportamento errôneo devido ao seu desgaste (queima de componentes, mau contato ou rompimento de ligações, etc.). Uma outra causa de falhas no hardware e' devida a seu projeto (considera-se a implementação como sendo também projeto, mesmo que de baixo nível pois, da mesma forma que o projeto funcional, requer tomadas decisões (RANDELL, 1975)). O software, por sua vez, pode apenas apresentar um comportamento inadequado devido a erros de projeto, uma vez que é um componente lógico do sistema.

Termos como "falha" e "erro" estão sendo usados até aqui sem nenhum rigor terminológico pois, por enquanto, os sentidos com os quais são empregados no cotidiano são suficientes para introduzir o assunto. Esses termos serão, contudo, definidos formalmente posteriormente neste capítulo.

Tradicionalmente, falhas em componentes

físicos, devido ao desgaste, têm sido estudadas e técnicas com o propósito de evitá-las ou tolerá-las propostas. Mais recentemente, contudo, o estudo de falhas de projeto, em especial de software, vêm ganhando atenção crescente (AVIŽIENIS & outros, 1985). É objetivo desse trabalho concentrar esforços no estudo de falhas de software. Não obstante, considerações relativas a falhas de hardware não serão desprezadas, memo porque, em várias situações, falhas de hardware e de software não se encontram disjuntas e, portanto, existem várias técnicas que valem tanto para um quanto para outro tipo.

O prejuízo ocasionado por uma falha em um sistema de computação varia enormemente, de acordo com o tipo de sua aplicação.

Uma falha pode ser imperceptível. Caso o seja no sentido de não afetar o resultado final do processamento, então não será necessário dar-lhe grande importância pois, a menos de uma eventual degradação em termos de tempo, ela não trará grandes problemas ao sistema. Ocorre, porém, que algumas falhas podem levar a resultados errados sem que isso seja perceptível. Nesses casos, as conseqüências podem ser desastrosas.

Uma falha pode, ainda, provocar a parada do sistema de computação. O transtorno causado por esse tipo de falha pode ser, simplesmente, uma perda de tempo provocada pela espera até o conserto do sistema ou pela reexecução de um processamento perdido quando da ocorrência da falha. Existem, contudo, situações onde o sistema não pode ser consertado, como em naves espaciais, aeronaves em vôo, navios em alto mar, etc. Em sistemas de tempo-real, por exemplo, a interrupção do processamento do sistema pode levar a danos irreparáveis.

Observa-se com isso que, dependendo da

aplicação do sistema de computação, é de fundamental importância sua capacidade em lidar com situações de falha. Este capítulo destina-se ao estudo de técnicas de tolerância a falhas. Nele serão apresentadas algumas definições de termos e conceitos empregados nessa área. Em seguida, serão estudadas algumas técnicas e mecanismos propostos para tolerância a falhas. Por fim, será descrito o mecanismo adotado no PORTOS-TF.

VII.2. Terminologia:

A questão da terminologia na área de sistemas tolerantes a falhas tem merecido a atenção de alguns autores (MELLIAR-SMITH & RANDELL, 1977; ANDERSON & LEE, 1982; LAPRIE, 1985). Alguns termos empregados nessa área são empregados no cotidiano com sentido muito próximo do proposto por esses autores. Mesmo assim, convém apresentar uma definição formal dos termos mais comumente encontrados na literatura e citados nesse trabalho, para que se possa discutir tecnicamente idéias e propostas em cima de termos amplamente aceitos pelos estudiosos do assunto.

VII.2.1. Termos Gerais:

No estudo de qualquer ramo da ciência são empregados termos que não são de âmbito específico e os textos de tolerância a falhas não fogem à regra. Algumas expressões se aplicam a várias outras áreas do conhecimento, mas como são fundamentais para a definição e o esclarecimento de outras específicas do tema "tolerância a falhas", serão apresentadas a seguir.

O ponto de partida para o estudo de mecanismos de tolerância a falhas é a caracterização de do universo observável. Esse universo caracteriza-se por um

sistema. Segundo ANDERSON & LEE (1982), "qualquer mecanismo identificável que mantenha um padrão de comportamento em uma interface com seu ambiente pode ser considerado como um sistema". Ainda segundo esses autores, "um sistema consiste de um conjunto de componentes que interagem sob o controle de um projeto. Um componente é simplesmente um outro sistema". Essa definição de componente sugere uma hierarquia de módulos, onde sistemas são compostos de sub-sistemas (ou componentes, do ponto de vista do nível hierárquico do sistema). Uma interface é o local onde ocorrem as interações entre dois sistemas. Desta forma, observa-se que também o ambiente é um sistema.

Um sistema dito "atômico" é aquele cujos detalhes internos não são do interesse do estudo em questão. Por exemplo, no projeto de um computador tolerante a falhas, mecanismos devem ser supridos de forma a evitar que um circuito integrado ao queimar faça com que o computador deixe de operar. Não interessa aos projetistas do computador (mesmo porque eles geralmente não podem) evitar ou mascarar a queima do circuito integrado. Dessa forma, este circuito integrado, que é um componente", constitui-se em um "sistema atômico" para os projetistas do computador. Esse conceito de sistemas atômicos é muito importante na escolha do nível em que se pretende tolerar falhas. Os componentes desse nível devem, pois, ser considerados atômicos, no sentido ora definido.

As manifestações externas de um sistema são conseqüência de sua atividade interna. Se o sistema não é atômico, então essa atividade pode ser investigada e manipulada. Dessa forma, é possível inserir mecanismos capazes de evitar que falhas em componentes impliquem em um comportamento inadequado do sistema.

VII.2.2. Termos Específicos:

Alguns termos vêm sendo empregados nesse texto sem nenhum rigor científico, como por exemplo "falha" e "erro". Apesar de seus significados não serem totalmente estranhos aos leigos, é necessário formalizar-se uma definição destes e de mais alguns outros para que se possa efetuar um estudo que não deixe margem a más interpretações.

Há três termos muito ligados à questão de mau funcionamento de sistemas. São eles: **falta**, **falha** e **erro**. Diversos textos propões definições para eles, como RANDELL, LEE & TRELEAVEN (1978), ANDERSON & KNIGHT (1981), ANDERSON & LEE (1982) e LAPRIE (1985). As definições ora apresentadas são baseadas nas propostas por MELLIAR-SMITH & RANDELL (1977):

.uma **falha** ocorre em um sistema quando seu comportamento externo não é o previsto em sua especificação.

.um sistema apresenta um erro quando, a menos que seja tomada uma atitude corretiva, poderá ocorrer uma falha cujas causas são anteriores ao aparecimento do erro (mesmo que este não tenha sido detectado).

.uma **falta** é um defeito que leva a um erro no sistema.

Quando *um componente* apresenta *uma* falta, esta poderá fazer com que ele falhe. Quando o estado externo de um ou mais componentes de um sistema tem que ser modificado para que o ele esteja em um estádo válido, isto é, de acordo com as especificações, diz-se que há um erro no sistema. Os estados desses componentes são erros do sistema. O estado de um desses componentes pode ser

resultado de uma falha do componente causada por uma falta interna a ele. Mas nem sempre esse estado, que constitui um erro para o sistema representa uma falha do componente. Nesse caso, a falha é do projeto. Conclui-se, pois, que um sistema entra em um estado de erro sempre que houver uma falha em um ou mais de seus componentes ou no projeto. Esse erro pode levar à falha do sistema. Assim, como encontrado em ANDERSON & LEE (1982) "uma falta é a causa de um erro e um erro é a causa de uma falha".

Qualquer sistema em operação tem a probabilidade (teórica ou verificada na prática através de observações) de não se comportar conforme especificado. Existe uma série de termos, que têm sido definidos por vários autores (RANDELL, LEE & TRELEAVEN, 1978; LAPRIE, 1985) que são empregados na avaliação dessa probabilidade. A seguir serão apresentadas algumas definições:

.**confiabilidade** (do inglês "reliability") é a medida do tempo contínuo em que o sistema se mantém em operação atendendo à sua especificação, desde o início de sua utilização.

.**disponibilidade** (do inglês "availability") é a porcentagem do tempo total em que o sistema opera de acordo com sua especificação.

.MTBF (do inglês "Mean Time Between Failures"), ou Tempo Médio Entre Falhas, é o valor médio dos intervalos de tempo nos quais o sistema permanece em operação seguindo sua especificação.

.MTTR (do inglês "Mean Time To Repair"), ou Tempo Médio de Reparo, é o valor médio dos intervalos de tempo nos quais o sistema não opera de acordo com sua especificação.

A disponibilidade de um sistema indica o quanto se pode contar com ele: por exemplo, sistemas com disponibilidade perto de 100% (cem por cento) estão quase sempre em operação adequada, enquanto que, à medida que esse valor diminui, fica cada vez mais difícil poder utilizar o sistema para os fins que foi projetado. A disponibilidade pode ainda ser calculada pela seguinte expressão:

$$\text{disponibilidade} = \frac{MTBF}{MTBF + MTTR}$$

Quanto maior o tempo médio entre falhas de um sistema maior sua disponibilidade. Ao contrário, quanto maior for o tempo médio de reparo, isto é, quanto mais tempo levar para um sistema ser "consertado", menos disponível ele será.

Sistemas altamente disponíveis (isto é, com disponibilidade próxima de 100%) são bastante aceitáveis, mas existem alguns casos onde uma interrupção em sua operação pode implicar em grandes perdas. Um sistema de distribuição de energia elétrica de uma cidade deve apresentar uma disponibilidade muito elevada, mas caso haja uma falta de luz por um pequeno período de tempo (MTTR baixo) este será tolerado. Já no caso de uma sonda espacial, como citado em POWELL & DESWARTE (1985), uma interrupção no funcionamento de um dos sistemas de controle da sonda pode significar o fim da missão, implicando na perda de vários milhões de dolares. Nesses casos não basta que o sistema seja altamente disponível. Ele deve ser muito confiável. Para aumentar a confiabilidade de um sistema suas falhas devem ser tratadas de forma a evitar a ocorrência de erros. A seguir serão estudados os tipos de tratamentos de faltas, erros e falhas.

LOQUES & KRAMER (1986) apresentam a seguinte classificação de sistemas quanto à influência que uma falha em um de seus módulos tem sobre a confiabilidade global:

.Baixa Dependência a Falhas: O projeto desse tipo de sistemas garante que, mesmo que um de seus módulos apresente uma falha, o sistema como um todo preservará sua confiabilidade, mesmo que passe a operar com um desempenho degradado.

.Alta Dependência a Falhas: Nessa classe de sistemas, a confiabilidade como um todo não se mantém quando ocorre uma falha em um módulo. Dessa forma, se um dos módulos do sistema falha, o sistema inteiro também falha.

VII.3. Classificação das Falhas:

Um sistema pode falhar de várias maneiras. Dependendo do modo como não obedece à sua especificação, o sistema pode causar danos diversos. A forma de tratamento de falhas de um componente ou de um sistema é muito dependente do tipo da falha. CRISTIAN & outros (1985), propuseram a seguinte classificação de faltas:

.falhas por omissão;
.falhas de tempo e
.falhas bizantinas.

Quando um componente nunca responde a um pedido ou toma alguma ação, diz-se tratar de uma falha por omissão. Este é o caso, por exemplo, em um sistema distribuído, em que um processo cliente que faz um pedido a um processo servidor que está em uma estação que não está ligada. Caso não haja nenhuma temporização, o cliente irá esperar eternamente pela resposta. Esse tipo de falha

também é conhecido como falha do tipo "fail-stop" (POWELL, 1985).

Ocorrem situações em que, apesar do componente fornecer o serviço da forma como especificado, isto é feito fora do intervalo de tempo desejado. Uma eventual sobrecarga do meio de comunicação ou uma falha intermitente pode causar um atraso intolerável em um sistema. De forma inversa, um relógio cuja freqüência esteja acelerada em relação aos demais do sistema pode precipitar, por exemplo, a expiração de um tempo de espera fazendo que o processo que aguardava o serviço inicie, impropriamente, um procedimento de tratamento de falha. A esse tipo de falhas, dá-se o nome de falhas de tempo.

O terceiro e de mais difícil tratamento tipo se refere a falhas bizantinas. Nesse grupo estão as falhas que fornecem serviços dentro do intervalo de tempo determinado, mas com uma forma indevida. Um algoritmo que, a partir de uma entrada válida apresente como resultado uma saída inválida é dito exibir uma falha bizantina. Falhas bizantinas têm as mais diversas origens. Podem ser devidos desde faltas em componentes de hardware até interferência eletromagnética, passando por erros de projeto. Falhas bizantinas são a forma mais geral de falhas.

O termo "bizantina" tem a ver com a analogia relativa ao problema de coordenação de generais bizantinos no comando de várias divisões acampadas ao redor de uma cidade inimiga (WALTER, KIECKHAFER & FINN, 1985). Os generais deveriam tomar decisões de atacar ou recuar de acordo com informações fornecidas pelos demais. Como alguns deles eram traidores, podiam fornecer informações diferentes, certas ou erradas, para generais diferentes. O problema consiste, então, em conseguir tomar as decisões corretas a partir da suposição de que algumas das mensagens recebidas poderiam estar erradas de forma completamente

arbitrária.

POWELL (1985) agrupa as falhas por omissão e falhas de tempo em uma mesma classe a qual denomina falhas "no domínio do tempo", enquanto as falhas bizantinas constituem falhas "no domínio do valor".

VII.4. Tipos de Tratamentos de Faltas, Erros e de Falhas:

Como visto anteriormente, a existência de faltas em componentes ou no projeto podem causar erros no sistema que podem levar a sua falha. Dessa forma, faltas erros e falhas devem ser tratadas de forma a aumentar a confiabilidade do sistema.

Há duas formas tradicionais de se construir sistemas altamente confiáveis (ANDERSON & LEE, 1982): prevenção de faltas ou tolerância a faltas. A primeira estratégia consiste garantir que o sistema não contém nem conterá falta alguma. Já em técnicas de tolerância a faltas ou, como é mais comum encontrar, tolerância a falhas, assume que podem haver faltas no sistema, ou seja, falhas nos componentes ou no projeto.

Em procedimentos de prevenção de faltas, dois pontos devem ser observados: evitar, em um primeiro passo, a introdução de faltas e, em uma etapa posterior, remover as faltas que porventura não tenham sido evitadas na primeira etapa. A utilização de componentes de maior qualidade, elaboração das técnicas de interconexão de elementos e cuidados com o encapsulamento do hardware com o propósito de evitar interferência eletromagnética (RANDELL, LEE & TRELEAVEN, 1978), bem como a aplicação de metodologias de desenvolvimento de projeto (ANDERSON & LEE, 1982) fazem com que sejam diminuídas as faltas do sistema. Na área de engenharia de software, técnicas de

especificação formal têm sido também empregadas. Pode-se citar como exemplo a especificação formal de protocolos. Posteriormente, através de procedimentos de teste e validação, o sistema é depurado ainda mais.

Ocorre, porém, que, por mais cuidado que se tenha no projeto, implementação e teste de um sistema, isso não garante a eliminação total da existência de faltas. Em RANDELL, LEE & TRELEAVEN (1978), em uma citação de Avižienis, é mencionado que "falhas ocasionais do sistema são aceitáveis como um mal necessário, e a manutenção manual é empregada para sua correção". Logo a seguir, neste mesmo texto, os autores sustentam que "há diversas situações nas quais a prevenção de faltas claramente não é suficiente". Segundo ANDERSON & LEE (1982), técnicas de tolerância a falhas "são necessárias porque sistemas complexos certamente contém faltas residuais apesar da aplicação extensiva de prevenção a faltas". Nos casos onde não possa haver reparos ou onde o tempo gasto nessa atividade seja proibitivo, deve-se poder conviver com as faltas de forma a manter alta a confiabilidade do sistema.

Mecanismos de tolerância a falhas não precisam necessariamente diagnosticar a causa da falha, nem mesmo decidir se ela foi devida ao hardware ou ao software (RANDELL, 1975).

VII.5. Princípios de Sistemas Tolerantes a Falhas:

A maioria dos projetos existentes de sistemas tolerantes a falhas partem de três assertivas (RANDELL, LEE & TRELEAVEN, 1978; LOQUES & KRAMER, 1986):

. que o projeto está correto, isto é, que não há nenhum algoritmo errado, seja na especificação ou na implementação. Um módulo só poderá entrar em

falha por outras razões, como falhas de hardware, por exemplo.

. que o erro é confinado no módulo, isto é, se houver falha em um módulo essa falha não será propagada, via mensagens, para os demais.

. que o subsistema de comunicação é confiável, ou seja, que não há como uma mensagem ser recebida degenerada em um módulo. Caso ocorra a degeneração da mensagem durante sua transmissão, o subsistema de comunicação será capaz de restaurá-la ou descartá-la totalmente.

Neste momento, cabe uma melhor definição do que se etá chamando de "módulo". Módulo é um componente de software independente (processo) que se comunica com os demais através de interfaces bem definidas. Essa independência entre os módulos permite a aplicação de técnicas de tolerância a falhas que garantam a confiabilidade do sistema. Não há dados compartilhados pelos módulos. A comunicação entre eles se dá exclusivamente por troca de mensagens. Todas as referências são locais. Um sistema é composto pela interconexão de módulos, ligando-se portos de entrada a portos de saída (vide capítulo IV). Essa configuração lógica torna possível a reconfiguração dinâmica do sistema sem que os módulos se apercebam disso. Configuração, ou reconfiguração, dinâmica é a capacidade de se mudar a configuração do sistema (criar e eliminar processos, ligar e desligar portos) com ele em execução.

A primeira das hipóteses descritas pode ser alcançada através do uso de técnicas e ferramentas de engenharia de software. Provas formais também podem ser feitas, embora isso seja muito custoso. A segunda hipótese depende dos mecanismos de detecção de erros de cada

estação. O baixo custo do hardware permite a duplicação dos processadores que, então, trabalham em um esquema de verificação mútua. Com relação à terceira hipótes, protocolos de comunicação bem definidos e rotas alternativas implementam um subsistema de comunicação confiável.

Apesar de desejáveis, as duas primeiras hipóteses nem sempre podem ser atendidas. Existem técnicas de tolerância a falhas que não assumem essas duas hipóteses. É objetivo deste trabalho relacionar algumas das técnicas que lidam com erros de projeto impedindo ou, pelo menos, limitando sua propagação para outros módulos do sistema.

Já a terceira pode ser mais facilmente alcançada, uma vez que (1) se os processos comunicantes se encontram na mesma estação, então não poderá haver falhas na comunicação (caso ocorram, serão devidas a falhas no hardware da estação que certamente implicarão em falhas internas no processamento dos módulos, constituindo-se, assim, em problemas referentes ao confinamento do erro) e (2) se estão em estações diferentes, então, a comunicação entre eles poderá ser feita através de protocolos que imbutem técnicas de detecção e correção de erros de comunicação. Essa é uma área muito interessante e vasta mas foge ao escopo deste trabalho.

Uma outra abordagem quanto à estruturação de sistemas tolerantes a falhas é a apresentada por MELLIAR-SMITH & RANDELL (1977). Esses autores sugerem quatro etapas que, cumpridas em conjunto, garantem a tolerância a falhas:

.detecção do erro;
.delimitação do dano causado pelo erro;
.recuperação do erro e

.continuação do serviço.

O primeiro passo em uma estrutura de tolerância a falhas é a detecção do erro. Isso ocorre através de algum teste de condição numérica ou do esgotamento de uma cláusula de tempo. Em seguida deve ser determinada a extensão dos danos causados pelo erro ao sistema. Nesse ponto, o erro deve ser recuperado. Essa recuperação se dá, normalmente, através da restauração de um estado correto anterior ao erro ou pela ida a um estado previamente definido para essa situação. Uma vez estando novamente o sistema em um estado correto, resta agora evitar que a falta cujo efeito levou o sistema ao estado errôneo ocorra novamente, logo em seguida. Isso é, em geral, alcançado através da reparação do módulo faltoso e da reconfiguração do sistema.

VII.6. Técnicas de Tolerância a Falhas:

Segundo CAMPBELL, ANDERSON & RANDELL (1983), um sistema tolerante a falhas é confiável na medida em que fornece serviços que atendem às suas expecificações, mesmo que ocorram falhas internas ou que contenham erros intrínsecos. De acordo com LOQUES, KRAMER & ANIDO (1988), todas as técnicas que provêm tolerância a falhas baseiam-se em alguma forma de redundância. Em termos de redundância de software, não basta haver replicação dos programas, mas também redundância no projeto (RANDELL, 1975). Essa redundância pode ser classificada, de forma geral, em duas categorias:

.redundância temporal e
.redundância de recursos.

A primeira delas baseia-se na repetição da atividade que falhou. Já a segunda se vale de um número

extra de recursos desempenhando a mesma função de forma a que, se um falhar, os demais ainda serão capazes de prover o serviço especificado.

A detecção da ocorrência de uma falha baseia-se na execução de algum teste de conformidade ou através da expiração de alguma cláusula de tempo ("time-out").

VII.6.1. Técnicas de Redundância Temporal:

Sistemas assíncronos são compostos de elementos cujos fluxos de operação são independentes, mas que podem eventualmente interagir entre si. Falhas em um componente de um sistema assíncrono podem contaminar alguns dos demais. Esse tema tem merecido a atenção de alguns autores (BELLON & SAUCIER, 1982; KOKAWA & SHINGAI, 1982). A contaminação ocorre pelo fato dos componentes interagirem entre si. Essa interação, contudo, não é total, isto é, não ocorre entre todos os componentes nem, tampouco, durante todo o tempo.

VII.6.1.1. Ações Atômicas:

Um grupo de componentes pode interagir entre si mas não fazê-lo com os demais. Dessa idéia surgiu o conceito de "**ação atômica**". Segundo ANDERSON & LEE (1981), "a atividade de um grupo de componentes constitui uma ação atômica se não há interações entre este grupo e o resto do sistema durante o tempo em que se desenvolve a atividade". O sistema apresenta um estado bem definido na entrada de uma ação atômica e outro na sua saída. Apesar de haver estados intermediários, estes não precisam ser considerados pelo sistema como um todo. As técnicas de tolerância a falhas baseadas na detecção e recuperação de erros atuam no interior de ações atômicas restaurando o estado em que se encontrava o sub-sistema, antes deste iniciar a ação, ou

estabelecendo-lhe um novo estado consistente. Ações atômicas delimitam a propagação de erros causada pela comunicação entre processos e, portanto, provêem o confinamento do erro.

A construção de sistemas assíncronos cujas atividades estejam estruturadas em ações atômicas é de fundamental importância na elaboração de técnicas de tolerância a falhas nesses sistemas. CAMPBELL & RANDELL (1983) propõem dois princípios para a estruturação de sistemas assíncronos tolerantes a falhas:

1. As operações executadas por um sistema assíncrono tolerante a falhas devem ser implementadas através de ações atômicas.

2. Todo e qualquer procedimento de tolerância a falhas deve estar ligados a uma determinada ação atômica e deve envolver todos os seus componentes.

Sistemas são, via de regra, estruturados de forma hierárquica. Um sistema é formado de componentes, que são sistemas menores também formados de componentes, e assim por diante, até chegar-se ao nível de circuitos eletrônicos em um "chip". Um sub-sistema, ou componente, deve prover serviços confiáveis ao sistema que compõe. Dessa forma, procedimentos de tolerância a falhas evocados na recuperação de erros internos a uma ação atômica de um componente devem ser transparentes para o sistema. Um sistema tolerante a falhas pode ser analisado, então, em cada um dos seus níveis de abstração (RANDELL, LEE & TRELEAVEN, 1978).

Em um sistema, durante a execução de uma ação atômica, pode ser evocado o serviço de um componente que, por sua vez, é também estruturado em ações atômicas. A essas últimas chama-se de "ações atômicas internas".

VII.6.1.2. Exceções:

É esperado de todo sistema que ele tenha um comportamento que siga suas expecificações. Quando é detectado um desvio no fluxo normal do processamento, uma das formas de corrigí-lo é levantar (ou sinalizar) uma exceção. Essa exceção indica um comportamento excepcional do sistema. Há vários tipos de exceções em um sistema. Cada tipo está ligada a uma causa diferente. Pode-se citar como exemplos de situações excepcionais a queda de força em uma estação, a violação do espaço de memória destinado a um processo ou, ainda, qualquer outro comportamento que fuja às condições normais de operação do sistema. Ligada a cada tipo de exceção deve existir uma rotina de tratamento da exceção. Essa rotina é acionada sempre que for levantada a exceção a ela associada. Ela tem como propósito efetuar as operações pertinentes, que variam de acordo com o significado da exceção.

As técnicas de tolerância a falhas baseadas em recuperação de estados utiliza amplamente mecanismos de exceção. CRISTIAN (1982) preocupa-se em distingüir a definição de exceções como mecanismos de manipulação de situações excepcionais e de erro daquela empregada por alguns autores como forma de efetuar procedimentos extras durante a execução de um procedimento de acordo com sua especificação.

O quadro de atividade de um módulo tolerante a falhas pode ser ilustrado pela figura (VII.1), copiado de CAMPBELL & RANDELL (1983):

O esquema da figura (VII.1) representa bem o que ocorre em um componente tolerante a falhas de um sistema com vários níveis de abstração. Cada nível desempenha uma função, de acordo com sua especificação. Esse nível fornece serviços ao nível superior mediante

solicitação deste. Da mesma forma, obtém serviços do nível abaixo mediante um pedido a ele.

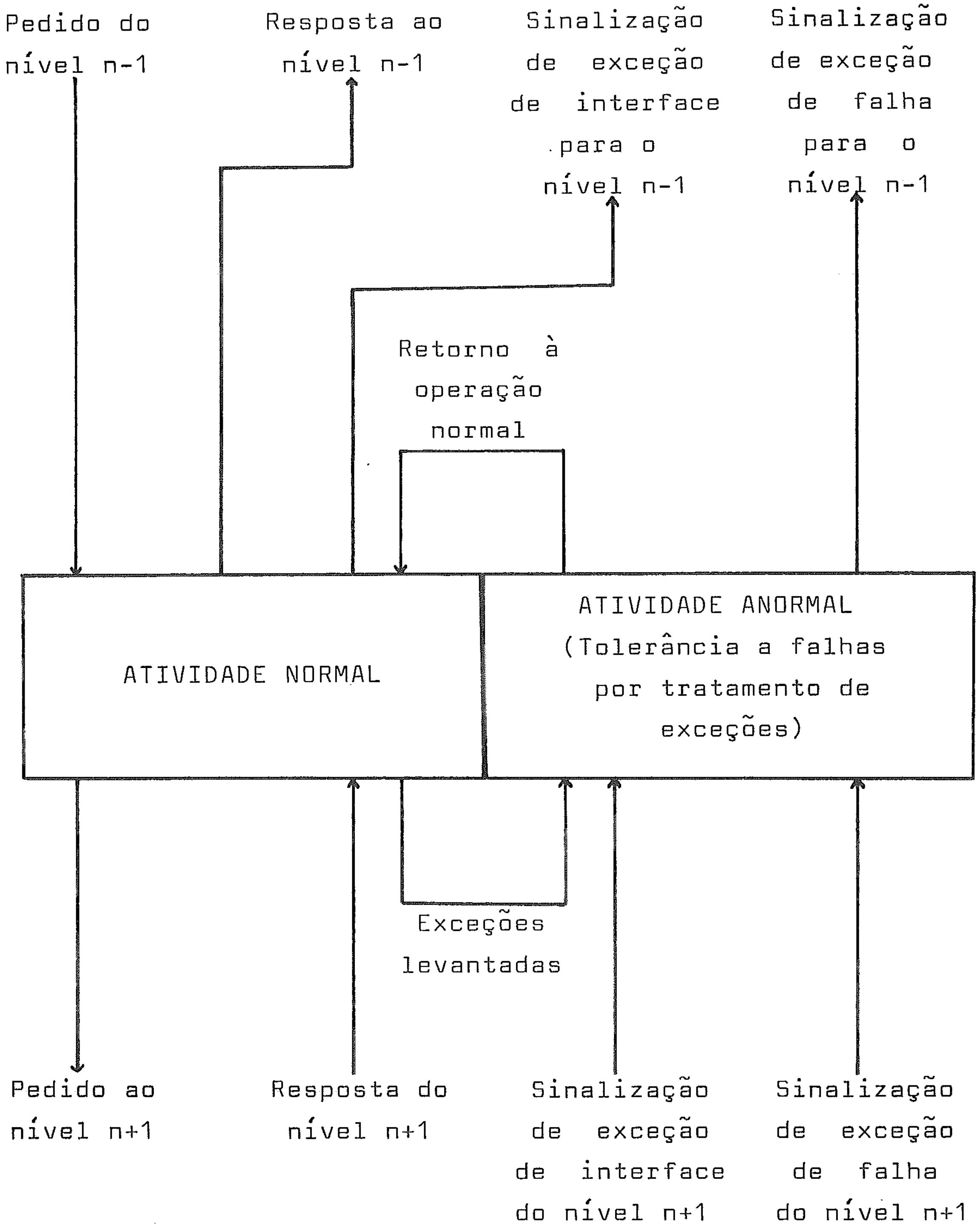


Figura VII.1 : Componente ideal tolerante a falhas

Quando ocorre alguma falha durante o processamento normal, o componente levanta uma exceção.

Como forma de atendimento a essa exceção, o componente passa a executar um procedimento dito excepcional. Esse procedimento tem por objetivo tentar recuperar a falha detectada que causou a exceção. Caso isso não seja possível por algum motivo, resta apenas ao componente avisar ao nível imediatamente superior ao seu. Este, por sua vez, deverá tomar as providências cabíveis para tolerar a falha sinalizada pelo nível inferior. Caso não consiga, sinaliza para o nível susperior e assim por diante. Como não há infinitos níveis superiores, observa-se, então, que essa cadeia de sinalização tem que terminar em algum nível (o nível mais elevado).

CAMPBELL & RANDELL (1983) propõem a utilização de uma "árvore de exceções". Quando o procedimento de tratamento de exceção de um determinado nível não conseguir tratar a falha, ele então sinaliza o seu "pai" e assim por diante, até que chegue à raíz da árvore. Nessa raíz é colocado um procedimento batizado por esses autores como procedimento de "exceção universal".

Árvores de exceção têm também o propósito de lidar com exceções simultâneas. Quando ocorre o levantamento de duas execeções "irmãs", é de pronto acionado o procedimento relativo ao nó "pai".

Como foi possível observar, se uma ação atômica não puder tratar suas próprias falhas, ela deve sinalizar uma exceção. Uma ação atômica que não consiga terminar corretamente nem sinalizar uma exceção não é tolerante a falhas.

VII.6.1.3. Recuperação de Erros:

Uma vez detectado um erro, providências devem ser tomadas no sentido de contê-lo e corrigí-lo. Há duas técnicas, conhecidas na literatura, que são empregadas

com este propósito: "backward error recovery" e "forward error recovery". Segundo CAMPBELL & RANDELL (1983), a técnica de "backward error recovery" restaura um estado do sistema anterior à manifestação da falta. Já a técnica de "forward error recovery" se preocupa em isolar ou corrigir erros específicos e é executada no estado do sistema no qual se manifestou o erro.

VII.6.1.3.1. "Backward Error Recovery":

Uma forma simples de se delimitar o alcance um erro é considerar suspeitos todos os passos executados dentro da ação atômica na qual foi detectado. Como conseqüência, o processamento efetuado até então, dentro da ação atômica deve ser desconsiderado.

A técnica de "backward error recovery" (RANDELL, LEE & TRELEAVEN, 1978; CAMPBELL & RANDELL, 1983) consiste na retroação de um ou mais processos, envolvidos na ação atômica, até um estado anterior ao erro.

Existem várias técnicas de recuperação de erro por retroação ("backward error recovery"). Nessa seção, serão apresentadas algumas delas, a saber: técnicas de redundância passiva ("hot stand-by" e "cold stand-by") (LOQUES & KRAMER, 1986) e blocos de recuperação ("recovery blocks") (RANDELL, 1975).

VII.6.1.3.1.1. Técnicas de Redundância Passiva:

Técnicas de redundância passiva caracterizam-se por apresentar um dos módulos redundantes em atividade enquanto os demais ficam aguardando. Quando ocorre uma falha nesse módulo, então um dentre os demais é ativado de forma a substituir o faltoso. Essa substituição se dá através da repetição de todo ou, pelo menos, parte do processamento realizado pelo módulo faltoso. Daí porque

esta técnica se encontra na classe de redundância temporal. Técnicas de redundância ativa, que estão na classe de redundância de recursos, serão discutidas oportunamente.

Em sistemas distribuídos, as réplicas são especificadas para execução em estações diferentes da original, uma vez que uma falha no hardware onde está o módulo original pode levar toda a estação ao colapso. Essa forma de redundância resolve questões de falhas do tipo "fail-stop" (ver ítem VII.3), ou seja, falhas que provocam a interrupção da execução do módulo quando ocorridas (LOQUES, 1988).

Há duas técnicas de redundância passiva:

. "cold stand-by" e
. "hot stand-by".

"Cold stand-by" é um tipo de redundância que se aplica a sistemas de "baixa dependência a falhas" (LOQUES & KRAMER, 1986). Nesse tipo de redundância, uma ou mais réplicas do módulo são especificadas com o objetivo de substituí-lo caso ele venha a falhar.

LOQUES & KRAMER (1986) propõem a adoção de duas entidades que controlam o funcionamento desses módulos redundantes, que são o "coletor de status" e o "gerente de configuração". O coletor de status tem a responsabilidade de monitorar constantemente o estado do módulo em atividade e, em caso de falha, informar o gerente de configuração. Este, por sua vez, deverá proceder a reconfiguração do sistema através da ativação de uma das réplicas, escolhida de acordo com uma política previamente determinada.

O procedimento de ativação da réplica escolhida consiste na criação desta, na estação especificada, e imediata reconfiguração do sistema de forma

a substituir o módulo faltoso pela réplica. Neste procedimento deve ser considerado o tempo gasto, que não é desprezível e pode comprometer as características de tempo-real do sistema. Uma forma de acelerar esse processo é criar a réplica mesmo antes dela ser posta em execução.

A maior característica desse tipo de redundância está na **não preservação** do estado corrente do módulo em falha que faz com que o módulo tenha que ser iniciado novamente. Essa restrição torna esse mecanismo inviável para sistemas com "alta dependência a falhas".

Em sistemas de "alta dependência a falhas", o processamento já efetuado pelo módulo original não pode ser perdido. Isso quer dizer, em outras palavras, que, mesmo quando um módulo falha, algum estado anterior à falha deve ter sido conservado. Para atingir este objetivo, é adotada uma redundância do tipo "**hot stand-by**". Nessa técnica, a réplica, além de ser criada, fica também pronta para executar, desde o instante da iniciação do sistema. A réplica fica como se estivesse no estado "bloqueado". A diferença deste para um bloqueio comum é que o estado do módulo original é copiado para a réplica em pontos chaves, chamados **pontos de checagem ("checkpoints")**. Controlando cada réplica existe um gerente. Esses gerentes monitoram os processos para detectarem eventuais falhas e, então, tomar as providências cabíveis avisando aos demais gerentes.

Em caso de falha do módulo original, a réplica será acionada. Ela, então, iniciará sua execução a partir do último "checkpoint". O tempo transcorrido nessa operação é menor do que o de ativação de uma réplica do tipo "cold stand-by", pois o tempo gasto é equivalente apenas à detecção da falha e à ativação da réplica.

No caso de ocorrer uma falha na réplica

(como, por exemplo, por mau funcionamento de sua estação) um procedimento de restauração do tipo "cold stand-by" poderá ser adotado.

Quando o sistema prevê a ocorrência de falha em mais de um módulo ao mesmo tempo, poderá trabalhar com várias réplicas simultaneamente. Isso, porém, implicará em um gerenciamento mais complexo e em um custo maior, principalmente no caso de "hot stand-by".

É interessante comentar uma utilização comum, porém errada, dos termos "hot stand-by" e "cold stand-by". É normal referir-se a "hot stand-by" como sendo uma arquitetura onde existe uma duplicata do sistema ativo que é ativada sempre que for detectada uma falha neste sistema. Essa ativação, contudo, equivale a um reinício do sistema, apenas em outra máquina. Em um esquema do tipo "cold stand-by", o comportamento é semelhante ao já descrito, diferindo apenas pelo fato da reposição e reiniciação ser feita manualmente com o auxílio de um operador. Observa-se, então, pelo que foi estudado nessa seção, que o que se está referindo como sendo uma arquitetura do tipo "hot stand-by" é na realidade uma arquitetura do tipo "cold stand-by", enquanto que a referida como tal não passa de uma configuração onde está disponível uma peça de reposição de almoxarifado.

VII.6.1.3.1.1.1. Suporte para Mecanismos de "Hot Stand-by":

A recuperação falhas em módulos a partir de técnicas de "hot stand-by" pressupõe algumas características especiais dos mecanismos de comunicação e sincronização entre processos. Essas características têm a ver com a capacidade das estruturas de comunicação filtrarem mensagens duplicadas e reenviarem mensagens perdidas. Na comunicação entre dois processos, pode falhar tanto o produtor quanto o consumidor da mensagem.

Quando um processo que enviou uma mensagem falha, ele é substituído por um reserva. Ao iniciar sua operação, este processo repetirá o envio da mensagem efetuado pelo processo original. Dessa forma, o processo consumidor da mensagem deve ser capaz de distinguir entre uma nova mensagem e uma retransmissão. No caso de transações do tipo "pedido-resposta", o consumidor deve ainda ser capaz de reproduzir a resposta. Isso pode ser obtido se o consumidor adotar como prática a retenção da última resposta enviada.

O sistema CONIC (LOQUES & KRAMER, 1986) adota uma seqüência de números para este fim. Cada entidade de comunicação tem, associada a ela, uma variável que armazena o número corrente da seqüência. Inicialmente, tanto o processo produtor quanto o consumidor assumem um mesmo valor.

Na transmissão, o produtor incrementa essa variável e agrega seu valor ao pacote da mensagem.

Quando a mensagem é recebida, este valor é comparado à sua referência. Caso seja uma unidade superior, então trata-se de uma nova mensagem e a mensagem será consumida normalmente. Caso seja igual ou menor, consiste em uma retransmissão que deve ser descartada. Se, contudo, seu valor for igual e for um pedido de uma transação do tipo "pedido-resposta", a última resposta enviada, e que está armazenada no consumidor, deve ser retransmitida. Valores maiores do que o da referência do receptor indicam transmissões perdidas. Isso só pode acontecer em transações sem confirmação, o que implica que o produtor não se importa se a mensagem foi ou não recebida. Dessa forma, ao concluir a recepção, o processo consumidor atualiza sua referência tornando-a igual à contida na mensagem. Se houver resposta, esta será

encaminhada com o valor atualizado da referência do consumidor.

Quando o produtor recebe a resposta (no caso de uma transação do tipo "pedido-resposta") ele considera aquelas cuja referência seja igual à sua. Respostas com referência inferior à do produtor são descartadas como sendo cópias. Não há como a referência da resposta ser maior do que a do pedido.

Como já mencionado anteriormente, o emprego da técnica de "hot stand-by" exige que as duplicatas do módulo estejam sempre em sintonia com o último ponto de checagem por onde passou este módulo. Para garantir essa sintonia, é necessário que seu estado seja copiado nas réplicas. Essa cópia é realizada através de uma primitiva de salvamento. Essa primitiva, quando chamada, envia o estado do processo que a chamou para todas as réplicas a ele associadas.

A chamada à primitiva de salvamento de estado marca um ponto de checagem no processo. Quando um módulo duplicata é ativado, ele começa sua execução a partir do último ponto de checagem por onde passou o módulo que falhou. Fica, assim, pendente a questão da localização dos pontos de checagem em um processo, ou seja, em que pontos deve ser evocada a primitiva de salvamento de estado.

A recuperação de uma falha é bem sucedida se o módulo, ao ser executado após a recuperação, é capaz de repetir todos os seus passos como foram antes de falhar. Enquanto o módulo é completamente estanque, isto é, não interage com os demais ou com o meio exterior, pode-se verificar que, a menos do consumo de tempo, ele repetirá sempre os mesmos passos, apresentando os mesmos resultados, mesmo que não possua nenhum ponto de checagem. A

ocorrência de entradas assíncronas, contudo, promovem variações no fluxo de execução de um processo. Logo, pode-se concluir que a perda dessas entradas pode inviabilizar a recuperação do módulo. Observa-se, dessa forma, que as chamadas à primitiva de salvamento de estado estão relacionadas com os pontos de entrada assíncronas.

Suponha-se dois processos A e B que se comunicam através de primitivas do tipo "pedido-resposta" ("request-reply"), como mostra a figura (VII.2). Sejam os pontos de checagem PC(A,0) e PC(B,0). Seja o ponto P(A,1) o ponto onde o processo A envia um pedido ao processo B e fica aguardando sua resposta. Seja o ponto P(B,1) o ponto onde o processo B fica aguardando o pedido do processo A. Seja, finalmente, o ponto P(B,2) o ponto onde o processo B envia a resposta ao processo A.

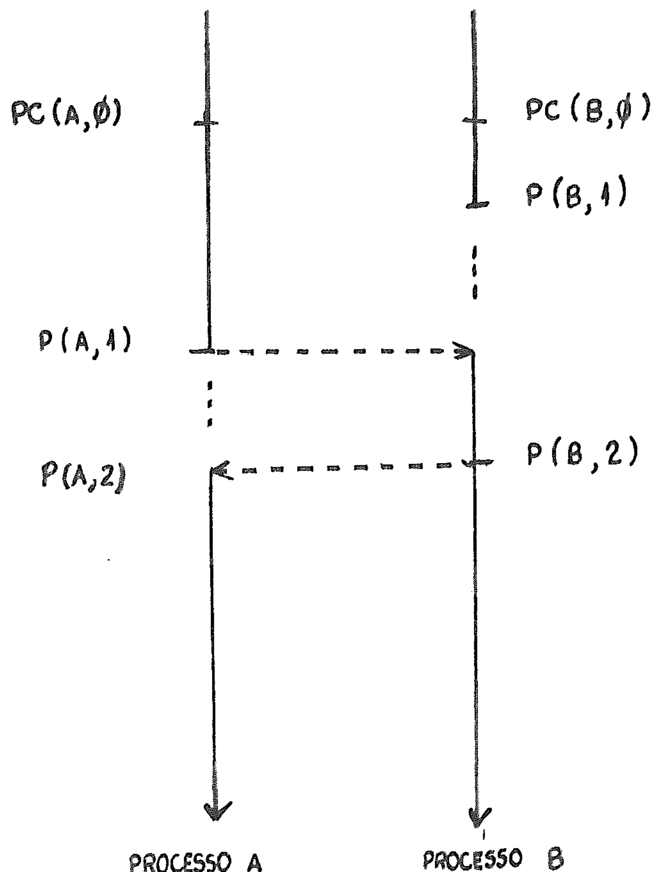


Figura VII.2 : Exemplo de utilização de pontos de checagem

Se houver uma falha no processo A antes de ambos começarem a interagir, isto é, antes dos pontos P(A,1) e P(B,1), o prejuízo ocasionado por esta falha, após a recuperação, será a reexecução do trecho entre PC(A,0) e P(A,1). Prejuízo análogo terá o processo B em situação semelhante. Conclui-se, portanto, que NÃO é imperativo o salvamento do estado dos processos nesse trecho de código.

O processo A ficará bloqueado enquanto o processo B estiver entre os pontos P(B,1) e P(B,2). Para garantir-se que uma falha ocorrida em B entre esses dois pontos será imperceptível a A, deve-se obrigar que o pedido enviado a B provocará, de qualquer forma, uma resposta a ser recebida por A. Assumindo-se a hipótese da comunicação entre processos ser totalmente confiável (ver ítem VII.5), garante-se que o pedido enviado por A será recebido por B. Da mesma forma, a resposta elaborada por B será recebida por A. Se o processo B falhar entre os pontos P(B,1) e P(B,2), então o processo B será substituído por uma réplica B'. Essa réplica iniciará sua execução a partir de PC(B,0), que foi o ultimo ponto de checagem do processo B, e ficará bloqueado em P(B,1) aguardando o pedido de A. O processo A, por sua vez, não receberá sua resposta dentro do tempo máximo de espera estipulado. Como conseqüência, o pedido será reenviado. O processo B' será desbloqueado, consumirá a mensagem e produzirá e enviará resposta, desbloqueando o processo A. Observa-se, então, que até este ponto não houve necessidade de se criar um novo ponto de checagem.

O problema surge quando o processo B falha após enviar a resposta ao processo A. Como pode ser observado, o processo B' só foi desbloqueado, dando assim continuidade ao processamento, porque A reenviou o pedido. Uma vez de posse da resposta, o processo A sairá do ponto P(A,2) e não mais reenviará o pedido original. Assim

sendo, a menos que haja um novo ponto de checagem após P(B,1), o processo B' ficará bloqueado eternamente neste ponto. Caso o processo A seja cíclico, um novo pedido desbloqueará B', mas o pedido original ficará perdido para ele. Dessa forma, LOQUES & KRAMER (1986) propõem a seguinte regra:

-Um ponto de checagem deve ser estabelecido após uma mensagem ser recebida e antes do resultado ser enviado. (Regra 1)

Se o processo A falhar após o ponto P(A,1), será substituído por um A' que iniciará no ponto PC(A,0). O pedido será então reenviado. Como o pedido será repetido, então o processo B reenviará a mesma resposta dada ao pedido original (enviado por A). A recuperação se realizará a contento. Ocorre, porém, que se A falhar após uma nova transação "pedido-resposta", posterior à realizada em P(A,1), a primeira não poderá mais ser recuperada pois a segunda resposta se sobrepôs à primeira, no processo B. Assim sendo, LOQUES & KRAMER (1986) propõem uma segunda regra que, juntamente com a primeira, são suficientes para garantir a tolerância a falhas.

-Um ponto de checagem deve ser estabelecido entre dois envios de mensagens através de uma mesma estrutura (como um porto, por exemplo). (Regra 2)

É importante salientar que este raciocícino só é válido para falhas do tipo "fail-stop" (ou falhas de omissão).

A transformação de um sistema não tolerante a falhas em um tolerante pode ser feita de forma transparente a partir das duas regras enunciadas acima. Em um sistema que tenha como primitivas de comunicação e sincronização entre processos as funções "envia", "recebe"

e "responde", como o PORTOS-TF (vide capítulo IV), ou análogas, isto pode ser conseguido através das seguintes modificações nessas primitivas:

1) É executado um salvamento de estado em toda a conclusão da primitiva "recebe". (atendendo a Regra 1)

2) É executado um salvamento de estado em toda a conclusão da primitiva "envia". (atendendo a Regra 2)

Como estas operações são executadas pelo sistema operacional, o processo que as evoca não toma conhecimento desse fato. Isso facilita a elaboração destes, pois elimina do projetista a preocupação com tolerância a falhas. Dessa forma, um mesmo módulo pode ser usado em uma configuração redundante para tolerar falhas ou isoladamente, sem esse objetivo.

Como o salvamento do estado nessas primitivas pode ser opcional (podem coexistir chamadas com salvamento do estado e chamadas sem salvamento), a decisão de se um módulo fará parte ou não de uma estrutura redundante ficará em um nível de configuração do sistema. Fica, pois, a cargo de um processo ou de um grupo de processos configuradores do sistema a determinação do tipo de comunicação que haverá entre as entidades de comunicação dos processos de operação: se confiável (ou seja, com salvamento) ou não (isto é, sem salvamento).

A existência explícita de uma primitiva de salvamento de contexto não é necessária, como foi visto, para a implementação de um mecanismo de "hot stand-by". Contudo, como ela deve existir pelo menos de forma implícita, para ser usada pelas primitivas "envia" e "recebe", pode ser interessante torná-la explícita. O projetista pode querer fazer uso desta primitiva para, por exemplo, evitar a repetição de um longo trecho de código no

caso de ativação de uma réplica. O sistema CONIC (LOQUES & KRAMER, 1986) é um exemplo dos que seguem essa idéia.

VII.6.1.3.1.2. Blocos de Recuperação ("Recovery Blocks"):

Um programa é processado através da execução, passo a passo, de suas instruções. A verificação da ocorrência de erros a cada instrução ou mesmo a cada linha é impraticável, seja por motivos de eficiência, pelo menos pela falta de uma visão mais global do processamento. Um programa bem feito é estruturado em blocos (laços, subrotinas, etc.).

RANDELL (1975) propõe a técnica conhecida como blocos de recuperação ou, no inglês, "recovery blocks", na qual um conjunto de blocos de um programa, agregados de algumas informações adicionais, funcionam como uma estrutura de detecção e recuperação de erros. Nesse esquema, não há nenhuma dependência quanto à forma de estruturação dos blocos, escopo das variáveis ou forma de passagem de parâmetros. A única exigência é que a entrada e saída de cada bloco de recuperação seja feita de forma explícita.

A referência explícita a mecanismos de tolerância a falhas no projeto dos módulos é, em muitos casos, desconfortável, pois introduz uma preocupação adicional na elaboração do módulo. A incorporação de mecanismos de tolerância a falhas de forma transparente ao projeto do módulo é uma característica que vem sendo buscada em trabalhos mais recentes (LOQUES, KRAMER & ANIDO, 1988). Como pôde ser observado no ítem anterior, o mecanismo de salvamento de estados usado na implementação da técnica de "hot stand-by" pode ser embutido de forma transparente ao projetista dos módulos.

Um bloco de recuperação consiste de um bloco

tradicional acrescido de:

.um teste de aceitação, onde é testada a existência de um erro e

.nenhuma, uma ou mais réplicas de reserva.

A sintaxe de um bloco de recuperação é a seguinte (RANDELL, 1975):

```
<bloco de recuperação> ::= ensure <teste de aceitação>
                           by <alternativa 1>
                           else by <alternativa 2>
                                  :
                                  :
                           else by <alternativa n>
                           else error

<teste de aceitação> ::= <expressão lógica>

<alternativa 1> ::= <lista de comandos>

<alternativa i> ::= <vazia> ou
                    <outra lista de comandos>
```

Figura VII.3 : Sintaxe de um bloco de recuperação

A <alternativa 1> constitui-se do código do programa original, isto é, aquele que seria usado em uma configuração não tolerante a falhas. Quando o processamento do programa atinge o bloco de recuperação ele inicialmente executa a <alternativa 1>. Ao terminar sua execução, será executado o <teste de aceitação>. Caso o resultado do teste seja afirmativo, então é presumido que não houve falha na <alternativa 1> e, então, o programa deixa o bloco de recuperação. Contudo, se o teste constatar que houve falha durante a execução da <alternativa 1>, então será recuperado o estado do processo

imediatamente anterior à entrada no bloco de recuperação. Em seguida, será iniciada a execução da <alternativa 2>, se houver. Caso não exista, então será sinalizado um erro para ser tratado em um nível superior (se houver). Daí até a última alternativa, o comportamento é análogo ao já descrito.

Um exemplo de um bloco de recuperação simples é mostrado em RANDELL (1975):

```
A:   ensure AT
     by        AP: begin
                         <texto de programa>
                   end
     else by   AQ: begin
                         <texto de programa>
                   end
     else error
```

Figura VII.4 : Exemplo de um bloco de recuperação simples

Neste exemplo, a alternativa AP é executada quando o processo entra no bloco de recuperação A. Ao seu término, o teste de aceitação AT será efetuado. Caso ele não indique falha em AP, então o processo sai de A normalmente. Caso contrário, o processo volta ao estado anterior à execução de AP e, então, inicia a execução da alternativa AQ. Ao final desta, o teste de aceitação AT é novamente processado. Caso não tenha ocorrido falha o processo deixa o bloco de recuperação normalmente. Caso o teste de aceitação AT tenha indicado uma falha em AQ, então, o processo deixará o bloco de recuperação A, mas será indicado um erro.

Um bloco de recuperação pode conter outros blocos de recuperação, ou seja, as alternativas de um bloco de recuperação podem conter outros blocos de recuperação,

como no exemplo VII.5, transcrito de RANDELL (1975).

```
A: ensure AT
      by       AP: begin declare Y
                        <programa>
                        B: ensure BT
                              by       BP: begin declare U
                                                <programa>
                                           end
                              else by BQ: begin declare V
                                                <programa>
                                           end
                              else by BR: begin declare W
                                                <programa>
                                           end
                              else error
                        <programa>
                   end
      else by AQ: begin declare Z
                        <programa>
                        C: ensure CT
                              by       CP: begin
                                                <programa>
                                           end
                              else by CQ: begin
                                                <programa>
                                           end
                              else error
                        D: ensure DT
                              by       DP: begin
                                                <programa>
                                           end
                              else error
                   end
      else error
```

Figura VII.5 : Bloco de recuperação mais complexo

Como se pode notar, o mecanismo de blocos de recuperação não tenta diagnosticar o erro. Ela simplesmente detecta-o e chaveia a execução para um módulo alternativo. Esse mecanismo assume que as falhas ocorridas são "devidas a inadequações residuais do projeto e que, portanto, tais falhas ocorrem somente em circunstâncias excepcionais" (RANDELL, 1975). Como o número de circunstâncias que podem surgir em um componente de software, mesmo que seja bastante simples, é imenso, então, mesmo que a <alternativa 1> falhe em uma determinada situação, ela continuará sendo a primeira a ser executada nas vezes posteriores. Os erros detectados nos testes de aceitação são inventariados para posterior análise e eventuais modificações nos módulos.

A aplicabilidade de "blocos de recuperação" está diretamente ligada à criação de alternativas e testes de aceitação eficientes.

VII.6.1.3.1.2.1. As Alternativas:

Como visto, a primeira alternativa é sempre a primeira a ser executada. Dessa forma, deve ser sempre a mais eficiente e cujo resultado seja o mais próximo do desejado. As demais alternativas, devem tentar cumprir a especificação, mas através de outro algoritmo, presumivelmente menos eficiente, mas preferencialmente mais simples. Essas outras alternativas, em muitos casos, podem até levar a resultados menos desejáveis que a primeira, mas ainda assim aceitáveis para o prosseguimento do programa. O exemplo da figura (VII.6) (retirado de RANDELL, 1975), ilustra bem essa questão.

O bloco de recuperação do exemplo da figura (VII.6) tem como objetivo incluir um novo elemento em uma seqüência. Mesmo que isso não seja possível, o programa

que contém o bloco aceita prosseguir sua execução desde que considere o estado final consistente. Pelo exemplificado, um estado final consistente é composto (1) da seqüência inicial aumentada do elemento, (2) da seqüência inicial acompanhada de uma mensagem de aviso de perda do ítem, (3) de uma nova seqüência composta apenas pelo novo ítem acompanhada de uma mensagem de aviso de perda da seqüência ou (4) de uma nova seqüência vazia acompanhada de uma mensagem de aviso de perda da seqüência e do ítem.

```
ensure seqüência consistente (S)
by      estenda S com (i)
else by concatene a S (construa seqüência (i) )
else by aviso ("ítem perdido")
else by S := construa seqüência (i);
        aviso ("seqüência perdida")
else by S := esvazie seqüência (S);
        aviso ("seqüência e ítem perdidos")
else error
```

Figura VII.6 : Um bloco de recuperação com alternativas que chegam a resultados diferentes mas, ainda assim, aceitáveis, apesar de menos desejáveis

VII.6.1.3.1.2.2. O Teste de Aceitação:

O teste de aceitação realizado após a execução de cada alternativa tem como função verificar se o processamento realizado por cada uma delas satisfez as especificações do programa que evocou o bloco. Dessa forma, ele não pode ser efetuado sobre variáveis locais ao bloco de recuperação, mas sim sobre variáveis globais ao programa, mesmo porque as primeiras só fazem sentido quando o programa está dentro do bloco de recuperação. O teste de aceitação deve estabelecer se os resultados de uma alternativa estão dentro dos limites especificados no

programa. Como as alternativas devem ser funcionalmente equivalentes, fica evidente que o teste de aceitação deve ser único para qualquer uma delas.

O teste de aceitação não necessita garantir a correção absoluta do bloco de recuperação. O projetista deve decidir o quão rigoroso será o teste. Idealmente, o teste verificará se o bloco de recuperação alcançou todos os requisitos da especificação do programa que o chamou. Contudo, por questões de custo e complexidade, pode-se aceitar um teste alguma coisa menos rigoroso do que isso. O exemplo da figura (VII.7) (RANDELL, 1975) ilustra essa situação. Nele, a ordenação de um vetor "S" é feita através de um bloco de recuperação. Ao invés do teste de aceitação verificar se, de fato, os elementos do vetor estão em ordem (verificando se o posterior é sempre maior ou igual ao anterior), ele simplesmente verifica se a soma dos elementos do vetor ordenado é igual à soma dos elementos do vetor original.

```
ensure ordenada( S ) ^ ( soma( S ) = soma( prior S ) )
by      ordenação_mais_rápida( S )
else by ordenação_rápida( S )
else by ordenação_por_bolha( S )
else error
```

Figura VII.7 : Um programa de ordenação tolerante a falhas

O teste de aceitação pode também conter erros. Espera-se, contudo, que eles sejam muito menos freqüentes que os encontrados nos blocos de recuperação, uma vez que são bem mais simples que eles. Os erros nos testes de aceitação podem ser recuperados nos blocos de recuperação que os englobam (vide figura VII.5), se existirem. Da mesma forma que erros nas alternativas, os erros nos testes de aceitação são inventariados para análise posterior.

Qualquer variável não-local ao bloco de recuperação que tiver sido alterada por ele deve ter seu valor original salvo para que, caso o teste de aceitação constate a ocorrência de uma falha, ele valor possa ser restaurado.

VII.6.1.3.1.2.3. Restauração do Estado Inicial:

A restauração do estado original do processo antes de iniciar a execução de uma alternativa só pode ser conseguida se os valores originais das variáveis estiverem salvos em algum lugar. Esse salvamento é feito automaticamente, de forma implícita sendo, assim, transparente ao projetista do processo. Logo, o mecanismo de blocos de recuperação deve implementar, de alguma forma, o salvamento e a restauração de estados dos processos.

A idéia mais imediata para o salvamento das variáveis que compõem um estado inicial de um bloco de recuperação é salvar todas as variáveis do processo. Apesar de suficiente, este procedimento não é necessário e, portanto, não deve ser adotado por ser muito ineficiente. Executar de forma inversa, uma a uma, todas as instruções executadas na alternativa que falhou também seria impraticável.

Na restauração do estado de um processo as variáveis locais ao bloco de recuperação não precisam ser consideradas. O escopo destas variáveis é interno a cada uma das alternativas e, portanto, não fazem parte do estado original do processo. O mecanismo de salvação do estado deve, então, concentrar-se nas variáveis globais. Uma variável cujo valor não foi alterado pela alternativa que está sendo desfeita não necessita receber seu valor imediato, pois já o contém. Dessa forma, verifica-se ser imperativo apenas o salvamento das variáveis que forem

alteradas pela alternativa. Mesmo assim, basta guardar o valor anterior à primeira alteração, pois o novo valor já é interno à alternativa e, portanto, deverá ser desprezado caso ela falhe.

RANDELL (1975) propõe um mecanismo de salvamento de variáveis não locais imediatamente antes de sua primeira alteração ao qual batizou de "'cache' recursivo" (do inglês, "recursive cache"). Esse mecanismo, composto parcialmente de hardware, é dividido em regiões, uma para cada nível de recuperação (para o caso de blocos de recuperação aninhados). Se um teste de aceitação indica que a alternativa falhou, então os valores salvos no "cache" são restaurados às variáveis e a região é descartada por inteiro. Porém, se o bloco de recuperação terminar satisfatoriamente, então parte de suas entradas serão descartadas, mas aquelas relativas a variáveis não locais ao bloco de recuperação do nível superior serão consolidadas com aquelas da região subjacente.

O "overhead" de tempo causado pelo mecanismo de "cache" de recursivo é linearmente proporcional ao número de variáveis não locais que são modificadas. Apesar de existir, ele é certamente muito menor do que um devido a operações explícitas de salvamento e recuperação.

VII.6.1.3.1.2.4. Recuperação de Erros em Processos Comunicantes:

No mecanismo de blocos de recuperação apresentado, o conceito de evolução no processamento do sistema tem levado em conta apenas alterações efetuadas em variáveis. Desta forma, para restaurar um estado, basta recuperar o valor das variáveis quando nesse estado. Mecanismos de "cache" recursivo, por exemplo, podem ser aplicados com esse fim.

Ocorre, porém, que um processo evolui de outras maneiras como, por exemplo, imprimindo caracteres em uma impressora, adquirindo dados em tempo-real de um sensor ou, ainda, enviando ou recebendo mensagens de outros processos. Desfazer tais ações é uma tarefa muito difícil ou mesmo impossível, em alguns casos. Apesar disto, eles devem ser desfeitos a fim de não comprometerem a recuperabilidade de estados inerente ao mecanismo de "cache" recursivo.

Pode-se observar que todas essas formas de avanço no processamento resumem-se a interações entre processos. Em alguns casos, não se tratam de programas, mas sim processos mecânicos, humanos, naturais, etc. Em outros, contudo, são processos computacionais que interagem entre si, cada qual estruturado em blocos de recuperação próprios.

Observe-se a situação de dois ou mais processos que trocam informações entre si. Estejam esses processos estruturados em blocos de recuperação, que podem estar aninhados. Suponha-se que, durante a execução de uma alternativa, o processos receba e destrua uma mensagem enviada por outro processo. Se, ao final de sua execução, o teste de aceitação invalidar essa alternativa, provavelmente outra será acionada e, então, a mensagem terá que ser reenviada pelo processo emissor. Ocorre, porém, que este processo já estará mais adiante em seu processamento. Logo, para repetir o envio da mensagem, ele também terá que ser retroagido. Outra situação onde a recuperação do estado anterior de um processo ocasiona a retroação de outros é quando o teste de aceitação invalida uma alternativa na qual houve a emissão de mensagens para outros processos. Esses outros processos também devem ser retroagidos de forma a receberem novamente a mensagem (provavelmente correta, dessa vez) para continuarem seu processamento a partir de dados corretos. De uma forma

geral, quando um processo volta a um ponto de recuperação, ele forçará que todos aqueles com os quais tenha interagido dentro do bloco de recuperação em questão também retornem a seus pontos de recuperação imediatamente anteriores às transações.

Em alguns casos, dependendo da forma com que estejam estabelecidos os pontos de recuperação e as transações entre os processos, a retroação sucessiva de vários processos pode ocorrer em cascata, fazendo com que todos os processos envolvidos retornem ao inicio do bloco de recuperação mais externo. Esse efeito é conhecido na literatura como "efeito dominó" (RANDELL, 1975; LOQUES & KRAMER, 1986), em analogia ao efeito observado quando se derruba o primeiro de uma carreira de dominós dispostos um atrás do outro.

Na figura (VII.8) é ilustrado um exemplo (transcrito de RANDELL (1975)) de configuração onde pode ocorrer o efeito dominó. Nesse exemplo, os processos 1, 2 e 3 estão no quarto bloco de recuperação, dentro de uma estrutura hierárquica, como a ilustrada na figura (VII.5). Caso o processo 3 falhe, pode-se observar que todos eles retornarão ao ponto 1.

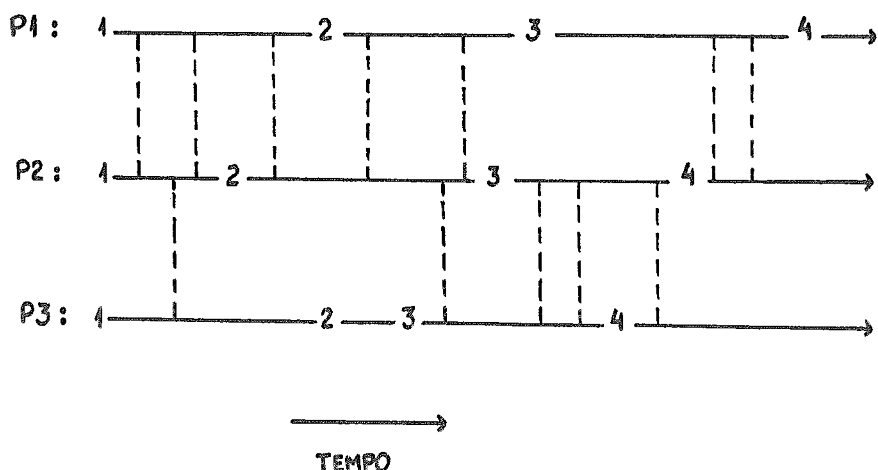


Figura VII.8 : Estrutura ilustrativa do "efeito dominó"

Segundo RANDELL (1975), a combinação de duas circunstâncias permite a ocorrência do efeito dominó:

1) A estrutura dos blocos recuperação dos vários processos comunicantes não é coordenada, nem leva em consideração as interdependências causadas pelas interações.

2) Os processos são simétricos em relação à propagação de erros, isto é, qualquer um dos membros do par que está se comunicando pode provocar a retroação do outro.

A remoção de qualquer uma dessas circunstâncias elimina a possibilidade da ocorrência do efeito dominó.

RANDELL (1975) propõe um esquema de coordenação no arranjo dos blocos de recuperação de processos comunicantes a fim de evitar o efeito dominó. A esse esquema foi dado o nome de "conversação".

Uma conversação é uma estrutura análoga a um bloco de recuperação, diferindo apenas pelo fato de ser relativa a dois ou mais processos, ao contrário do bloco de recuperação que pertence a apenas um processo. Uma conversação estabelece o ponto para o qual todos os processos nela envolvidos deverão ser retroagidos caso ocorra um erro em um ou mais deles. Há duas normas para a utilização correta de uma conversação:

1) Os processos devem entrar na conversação antes de trocarem qualquer informação.

2) Os processos devem deixar a conversação todos ao mesmo tempo.

A figura (VII.9), também extraída de RANDELL (1975), ilustra duas conversações entre três processos.

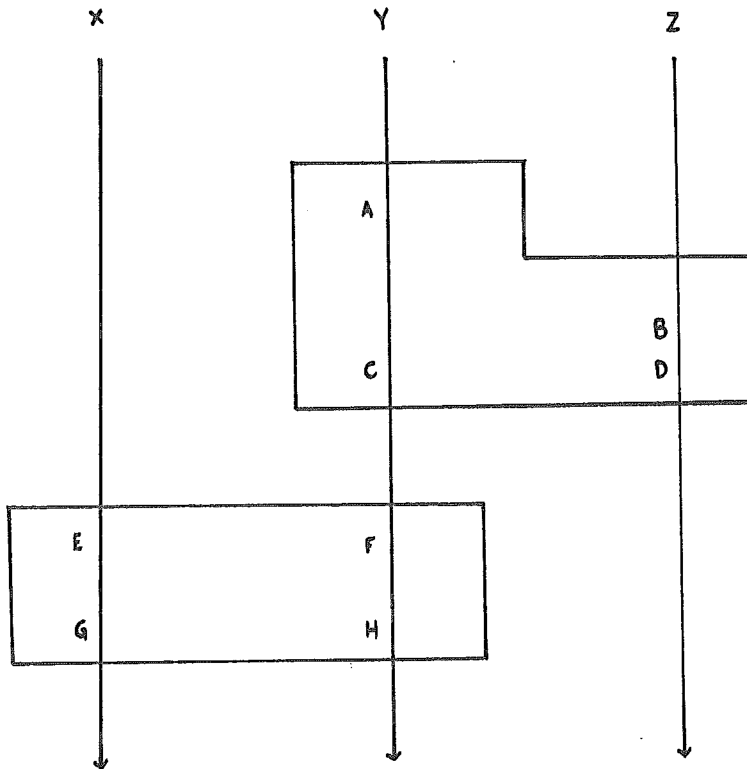


Figura VII.9 : Transações entre processos através de conversações

Como pode ser observado na figura (VII.9), os processos não necessitam entrar simultaneamente na conversação. Conversações podem ser aninhadas, assim como blocos de recuperação e, assim como eles, não podem se interceptar. O esquema de conversação é válido para qualquer tipo de mecanismo de comunicação empregado, seja baseado em variáveis compartilhadas, seja baseado em troca de mensagens.

VII.6.1.3.2. "Forward Error Recovery":

Como visto anteriormente, na técnica de "backward error recovery" os processos faltosos são levados a um estado anterior ao erro para, em seguida, prosseguir seus processamento através de caminhos alternativos. Em contraste com essa proposição, existe a técnica de "forward error recovery" (RANDELL, LEE & TRELEAVEN, 1978; CAMPBELL & RANDELL, 1983) que tenta fazer uso do estado no qual foi detectado o erro.

A técnica de "forward error recovery" consiste na execução de um certo conjunto de procedimentos que têm o propósito de tratar o erro e que são acionados a partir da sua detecção. Assim sendo, um sistema, cuja tolerância a falhas é baseada em "forward error recovery", é projetado e implementado juntamente com procedimentos específicos para o tratamento de cada tipo de erro. Quando um desses erros ocorre o procedimento a ele relativo é executado.

Segundo RANDELL, LEE & TRELEAVEN (1978), há duas formas de se implementar mecanismos de tolerância a falhas baseados em "forward error recovery":

.compensação e
.gerenciamento de exceções.

A primeira forma consiste na geração de informações suplementares capazes de corrigir o efeito de alguma informação errônea produzida e já consumida por outros módulos do sistema. Essa informação suplementar pode ser produzida pelo próprio componente faltoso ou por outro, em seu benefício. Esse tipo de mecanismo é de grande valia em situações onde não se pode utilizar mecanismos de "backward error recovery" por não ser possível salvar-se estados anteriores do sistema.

Outra forma de se implementar mecanismos de tolerância a falhas do tipo "forward error recovery" é através de mecanismos de exceção (ítem VII.6.1.2 e CRISTIAN, 1982). Neste tipo de implementação toda a vez que for detectado um erro, a exceção a ele relativa será levantada (ou sinalizada). Mediante essa sinalização, a rotina de tratamento da exceção será executada e o erro reparado.

"Forward error recovery" aplica-se bem ao tratamento de erros previsíveis e bem conhecidos, como os causados por falhas de componentes ou nas interações entre eles. Em comparação à técnica de "backward error recovery", ela é mais simples (pois não desfaz o que já foi feito e refaz de outra forma) e, portanto, mais rápida.

Ocorre, todavia, que, ao contrário da técnica de "backward error recovery", onde as etapas de delimitação do dano e reparo estão bem distintas da continuação do serviço, em "forward error recovery" essas questões estão misturadas (RANDELL, LEE & TRELEAVEN, 1978). Além disso, enquanto na primeira a delimitação do dano causado pelo erro é feito independentemente da causa do erro, na segunda existe uma grande ligação com a identificação da falta ou, pelo menos, com o conhecimento de todas as suas conseqüências. Assim sendo, conclui-se que mecanismos de "forward error recovery" não se adequam ao tratamento de falhas imprevisíveis, como falhas de projeto, por exemplo. Para esse tipo de falhas é preferível a adoção de mecanismos de "backward error recovery".

A tentativa de se prever erros de projeto, através, por exemplo, de diagnose automatizada, consiste em total desperdício de esforços pois, como bem citado em MELLIAR-SMITH & RANDELL (1977), quando o tipo e a

localização de uma falha de projeto puder ser prevista esta deverá ser removida ao invés de tolerada.

Como observado em CAMPBELL & RANDELL (1983), "forward" e "backward error recovery" complementam-se mutuamente. Enquanto a primeira permite o tratamento eficiente de falhas esperadas, a segunda provê uma estratégia geral de tratamento de erros que o projetista não pôde ou não quis antever.

VII.6.1.3.3. "Backward Error Recovery", "Forward Error Recovery" e Exceções:

Como mencionado no ítem anterior (VII.6.1.3.2), mecanismos de "forward error recovery" podem ser implementados através da utilização de mecanismos de exceção. O exemplo da figura (VII.10), extraído de CAMPBELL & RANDELL (1983), ilustra de que forma isto pode ser alcançado:

```
falha_do_motor : rotina_de_tratamento_de_exceção
    begin
    evitar_estol
    abaixar_flaps
    escolher_campo_de_pouso_de_emergência
    chavear_tanques_de_combustível
    chavear_magnetos
    ...
    end
```

Figura VII.10 : Procedimento de emergência para aviões leves

Nesse exemplo, a estratégia de "forward error recovery" tenta conter o dano ao motor do avião, tentando aterrizá-lo de forma segura. Esse procedimento é chamado através do levantamento da exceção correspondente.

```
(* garante operação correta *)
bloco_primário : componente_do_sistema
     inicia_cache
     habilita( outras_exceções, alternativa_1 )
     executa_algoritmo_primário
     se não ( teste de aceitação )
          então sinaliza( exceção_alternativa )
     desabilita( outras_exceções, alternativa_1 )
     descarta_cache
     retorna
(* fim do bloco primário *)

bloco_alternativo_1 : rotina_de_tratamento_de_exceção
     restaura_cache
     habilita( outras_exceções, alternativa_2 )
     executa_algoritmo_alternativo
     se não ( teste de aceitação )
          então sinaliza( exceção_alternativa )
     desabilita( outras_exceções, alternativa_2 )
     descarta_cache
     retorna
(* fim do bloco alternativo 1 *)

bloco_alternativo_2 : rotina_de_tratamento_de_exceção
     restaura_cache
     sinaliza( exceção de falha )
(* fim do bloco alternativo 2 *)
```

Figura VII.11 : Implementação de bloco de recuperação através de gerenciamento de exceções

Mecanismos de "backward error recovery" do tipo blocos de recuperação podem ser implementados de forma transparente através de, por exemplo, estruturas do tipo "cache" recursivo, apresentado em VII.6.1.3.1.2.3. A implementação de blocos de recuperação, contudo, pode ainda

ser feita de forma explícita, através da utilização de mecanismos de exceção (CAMPBELL & RANDELL, 1983), apesar de serem naturalmente aplicáveis a tecnicas de "forward error recovery". Nesta implementação, exceções inexperadas são transformadas em exceções esperadas mas do tipo "outras exceções", como forma de ativação de outras alternativas do bloco de recuperação. Além disso, o tratamento do "cache" recursivo e do teste de aceitação também é feito de forma explícita. O exemplo da figura (VII.11) (CAMPBELL & RANDELL, 1983) ilustra esse esquema.

VII.6.2. Técnicas de Redundância de Recursos:

Técnicas de redundância de recursos baseiam-se na multiplicidade de elementos que efetuam, simultaneamente, tarefas funcionalmente equivalentes. Ao contrário de técnicas de redundância temporal (vide item VII.6.1), onde a tolerância a falhas se dá pela detecção da falha, recuperação de um estado anterior e execução de um novo módulo alternativo, mecanismos de redundância de recursos mascaram a falha através de alguma forma de comparação dos resultados obtidos após a execução paralela dos módulos redundantes. O exemplo canônico desse tipo de estrutura é a "Redundância Modular Tripla" (ou, no inglês, "Triple Modular Redundancy") (RANDELL, LEE & TRELEAVEN, 1978). A figura (VII.12) ilustra a estrutura de um mecanismo de redundância modular tripla.

No esquema da figura (VII.12), três módulos funcionalmente equivalentes recebem a mesma entrada e, cada qual isoladamente, apresenta uma saída. Como pode ter ocorrido uma falha em um dos três módulos, então, a saída de cada um deles será comparada às dos demais em um elemento chamado "votador". Esse elemento tem como função apresentar uma saída que seja válida independentemente de falhas que possam ter ocorrido nos módulos. A escolha dessa saída baseia-se em uma votação majoritária, onde, se

pelo menos dois dos três módulos votantes concordarem, o sistema poderá suprir uma saída confiável. Caso haja uma discordância geral, isto é, cada módulo votante apresentar uma saída diferente das demais, então o votador indicará uma falha não tratável. (No ítem VII.6.2.2. serão discutidas situações onde resultados diversos não representam falha intratável.)

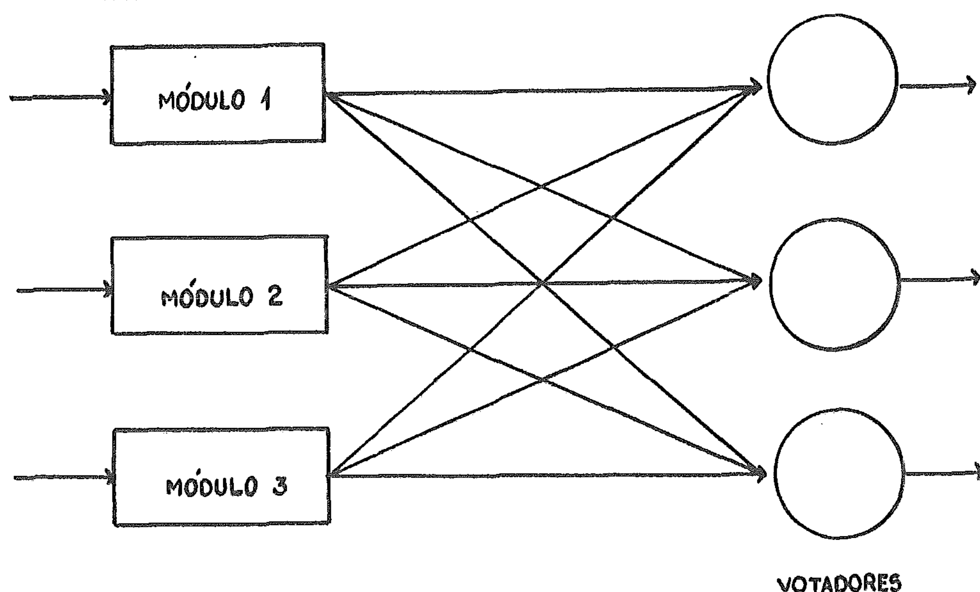


Figura VII.12 : Esquema de redundância modular tripla

A votação é uma forma mais eficiente de detecção de erro em um módulo do que testes de aceitação. Isso se deve ao fato de que o resultado de um módulo é comparado com os dos demais que executam algoritmos análogos ao seu. No caso de um teste de aceitação, ocorre apenas o estabelecimento de um limite de validade do resultado da alternativa. Mesmo que o teste de aceitação seja eleborado de forma a efetuar um algoritmo equivalente aos das alternativas para checá-los de forma mais rígida, ainda assim haverá apenas dois elementos sendo comparados (enquanto que em N-versões há tantos quanto se queira). Mesmo neste caso, existe a necessidade de garantir-se a correção do teste de aceitação sob pena dele recusar

alternativas corretas.

É importante salientar que a redundância modular tripla, onde vale a relação 2 de 3, não é a única combinação possível em mecanismos baseados na redundância de recursos. Qualquer número de módulos pode ser empregado, seguindo-se uma votação majoritária (por exemplo, 3 de 5, 4 de 7, etc.).

Mecanismos de redundância de recursos também são conhecidos como mecanismos de redundância ativa, pois seus módulos estão em atividade ao mesmo tempo, ao contrário de mecanismos tidos como de redundância passiva, como "hot stand-by", por exemplo.

VII.6.2.1. Os Módulos:

Mecanismos de tolerância a falhas que utilizam redundância de recursos tratam não só falhas devidas ao desgaste de componentes de hardware, mas também falhas de projeto. Essas causas podem provocar os três tipos de falha apresentados no ítem VII.3: falhas por omissão, falhas de tempo e falhas bizantinas. Cabe aos votadores a tarefa de detectar uma falha em um dos módulos, mas a forma com que eles são projetados permite uma taxa maior ou menor de ocorrência de falhas.

As N-versões de um mesmo módulo são reunidas em uma estrutura chamada "grupo" (do inglês, "troupe" (COOPER, 1985) ou "cluster" (LOQUES, KRAMER & ANIDO, 1988)). Os membros de um grupo podem estar em uma mesma estação ou em estações distintas. No caso de um sistema distribuído, esses membros podem ser replicados para tolerarem falhas do tipo "fail-stop" das estações. Essas réplicas podem ser idênticas ou diferentes, mas funcionalmente equivalentes. No caso de uma arquitetura individual, ou seja, de um único computador, não há o caso

da estação apresentar uma falha do tipo "fail-stop". Dessa forma, não faz sentido a utilização de réplicas idênticas. Portanto, em arquiteturas individuais, só faz sentido trabalhar com versões diferentes do mesmo módulo. Essa técnica, conhecida como N-versões, tem merecido a atenção de vários autores como: AVIŽIENIS & outros (1985), KNIGHT, LEVESON & St. JEAN (1985), COOPER (1986), LOQUES & KRAMER (1986), POWELL (1987) e LOQUES, KRAMER & ANIDO (1988).

Na técnica de "N-versões" os módulos redundantes são implementados independentemente, normalmente por programadores diferentes, a partir de uma mesma especificação funcional. Seu objetivo é fazer com que eventuais erros de projeto em um módulo sejam mascarados por outros módulos onde não exista tal erro. Sendo os módulos independentes, a confiabilidade do sistema é igual à enésima potência da confiabilidade dos módulos isoladamente. Todavia, parece ser impossível projetar módulos sem que não haja nenhuma correlação entre eles.

KNIGHT, LEVESON & St. JEAN (1985) desenvolveram uma pesquisa que constatou que, de fato, os N módulos de um sistema do tipo N-versões não são absolutamente independentes. Isso se deve ao fato de que existe a tendência de se cometer o mesmo erro na solução de um mesmo problema, mesmo quando se está trabalhando independentemente. Além disso, existem trechos de um programa que são naturalmente mais difíceis. O tipo de ambiência educacional, cultural e social também pode levar a raciocínios similares. O resultado da pesquisa, porém, não desmente o aumento da confiabilidade do sistema trazido pelas N-versões, pelo contrário, confirma este fato. Ele mostra, apenas, que esse aumento não é tão grande quanto o seu valor teórico, uma vez que este assume que os módulos não apresentam nenhuma correlação.

VII.6.2.2. Os Votadores:

Alguns autores têm publicado trabalhos que discutem a questão da votação em sistemas tolerantes a falhas (GUNNINGBERG, 1983; AVIŽIENIS & outros, 1985; COOPER, 1985; WALTER, KIECKHAFER & FINN, 1985; POWELL, 1987; LOQUES, KRAMER & ANIDO, 1988).

O tipo de votação mais imediato que existe consiste em comparar os vários resultados bit-a-bit (ou byte-a-byte). Caso a maioria dos resultados seja igual, então esse resultado "mais votado" é considerado como válido. Caso haja módulos discordantes, pode ser feito um registro para análise posterior, como nos blocos de recuperação (vide ítem VII.6.1.3.1.2) ou esse fato pode ser simplesmente ignorado. Se, todavia, não houver um resultado majoritário, então nenhum resultado deve ser apresentado, uma sinalização indicando a discordância deve ser feita e o sistema deve ser suspenso de forma segura. A questão da votação não é tão simples assim. Existem três fatores complicadores: o conteúdo da composição dos resultados, o tempo que eles levam para estar disponíveis e a ordem com que chegam ao votador.

O primeiro desses fatores tem a ver com o fato de que os valores bit-a-bit de dois resultados podem não coincidir sem que isso implique que um dos módulos falhou. Essa diferença entre dois resultados pode ser devida à diferença da precisão de números reais em estações distintas, por exemplo. Essa diferença independe do fato se se estar usando N-versões ou réplicas idênticas. Outra causa de diferenças é a implementação de cada uma das réplicas através de algoritmos alternativos. Cada algoritmo pode chegar a valores próximos porém diferentes. Essa situação pode ocorrer mesmo em uma única máquina onde foi adotado um mecanismo de N-versões. Por fim, dois resultados podem ser inteiramente diversos e ainda assim

estarem corretos. É o caso, por exemplo, de um módulo de geração de números randômicos. Isso força com que a votação tenha que ser efetuada sobre vários tipos de dados: booleanos, inteiros, em ponto flutuante e até randômicos. A título de ilustração, pode-se citar o sub-sistema de votação do MAFT ("Multicomputer Architecture for Fault-Tolerance") (WALTER, KIECKHAFER & FINN, 1985), aplicado em sistemas de tempo-real que requerem tanto alto desempenho quanto alta confiabilidade. No MAFT, o sub-sistema votador trabalha com dados de vários tipos incluindo-se valores booleanos, vários formatos de inteiros e o padrão IEEE de 32 bits para variáveis em ponto flutuante.

O segundo fator que deve ser levado em conta na votação é o tempo que leva para que os resultados fornecidos por todos os módulos votantes esteja disponíveis para a votação. Os resultados que devem ser comparados em uma votação não chegam ao votador ao mesmo tempo. Esse intervalo aumenta ainda mais quando se trata de um sistema distribuído. Para tanto, um intervalo de tempo deve ser adotado para que, durante esse tempo, o votador aguarde os valores restantes. De acordo com AVIŽIENIS & outros (1985) "o único meio de se detectar que uma versão não produziu um resultado quando este era esperado ou quando o resultado fica 'preso' em algum lugar no sistema de comunicação é usar uma função de tempo de espera". Aqueles que não chegarem dentro do prazo estipulado serão considerados como omissos, ou seja, será presumido que o módulo responsável por sua geração está inativo. Porém, como exposto em POWELL (1987), "o problema da técnica de tempo de espera é o do dimensionamento dos tempos de espera já que são, em essência, muito dependentes da carga do sistema e podem ter que ser modificados a medida em que o sistema evolui".

AVIŽIENIS & outros (1985) sugerem duas formas de ativação da contagem de tempo. Na primeira delas, a contagem do tempo é iniciada quando o processo

chega no início de cada bloco de votação. Todas as versões devem fornecer seus resultados dentro deste intervalo de tempo. Em uma segunda política, o tempo de espera só é iniciado quando chega a maioria dos resultados a serem votados. Essa segunda opção se baseia em uma comparação entre os tempos relativos de execução ao invés de utilizar o tempo absoluto, como na primeira técnica. Outra vantagem é que versões que apresentem mau funcionamento através da entrega prematura de seu resultado não trará problemas à votação, uma vez que não iniciarão a contagem.

O último fator que deve ser analisado na votação é a seqüência com que chegam resultados para serem votados. Quando um resultado é recebido em um votador ele deve, de alguma forma, ser verificado para assegurar-se não se tratar de uma retransmissão de um resultado anterior ou um resultado antigo que está sendo entregue com atraso. Além disso, em sistemas distribuídos, um sub-sistema de comunicação tolerante a falhas deve, segundo CRISTIAN & outros (1985):

1) Entregar todas as mensagens produzidas por versões corretas a todos os destinatários corretos dentro de um intervalo de tempo pré-determinado.

2) Assegurar que toda a mensagem cuja transmissão (difusão) seja iniciada em uma versão deve ser entregue a todas as versões destinatárias corretamente ou não deve ser entregue a nenhuma delas.

3) Garantir que todas as mensagens entregues de todas as versões devem sê-lo na mesma ordem a todas as versões destinatárias.

Esse tipo de protocolo recebe o nome de "protocolo de difusão atômica" (do inglês, "atomic broadcast protocol"). CRISTIAN & outros (1985) propõem

como forma de ordenar as mensagens a inclusão no pacote de um "selo de tempo" que garanta que mensagens corretas não sejam descartadas por mau dimensionamento do tempo de espera e que a ordem de entrega seja respeitada em todas as versões destinatárias. Todos os processadores trabalham com "relógios aproximadamente sincronizados". Como há um atraso "delta" na transmissão das mensagens através do sub-sistema de comunicação, todos os processadores concordam em qualquer instante "T" no histórico de todos os eventos ocorridos até o tempo de relógio "T - delta". Estes protocolos são conhecidos como "protocolos síncronos", já que todos os processadores atingem mesmos tempos locais em um intervalo de tempo-real cujo tamanho é limitado por um desvio máximo igual a "épsilon".

VII.6.2.3. Primitivas do Sistema Operacional:

Um sistema tolerante a falhas depende da existência de algumas primitivas no núcleo do sistema operacional sobre as quais ele possa ser implementado. A literatura ilustra uma série de sistemas que dispõem dessas facilidades: o DEDIX (AVIŽIENIS & outros, 1985), da Universidade da Califórnia em Los Angeles, o CIRCUS (COOPER, 1985), da Universidade da Califórnia em Berkeley e o CONIC (LOQUES, KRAMER & ANIDO, 1988), do Imperial College, Londres, entre outros.

Um sistema operacional deve posibilitar a criação de uma estrutura de N-versões. A transparência e seletividade dessa estrutura do ponto de vista do projeto das versões é uma característica tida como importante e vem sendo buscada em trabalhos recentes (LOQUES, KRAMER & ANIDO, 1988). O projetista dos membros do grupo não deve se preocupar com o fato de existirem ou não outros membros. Além disso, deve ser facultativo, do ponto de vista global do sistema, a tolerância a falhas de um módulo. Dessa forma, um sistema tolerante a falhas só deve ser montado em

uma etapa de configuração e não na fase de projetos dos módulos.

O elemento básico dessa arquitetura é o "grupo" (ou "cluster" ou, ainda, "troupe"). A fim de criar um grupo, o sistema operacional deve suprir primitivas de inclusão e retirada de processos (réplicas) no grupo, além de incorporar às primitivas de comunicação e sincronização os mecanismos de votação, de forma transparente.

VII.6.3. Comparação entre Redundância Temporal e Redundância de Recursos:

Pelas características já estudadas dos mecanismos de redundância temporal e de redundância de recursos, é possível se fazer algumas comparações entre os dois tipos de redundância. Essas comparações são importantes na medida em que são decisivas na escolha da técnica a empregar para tornar um sistema tolerante a falhas.

A adoção de mecanismos de tolerância a falhas impõe, sem dúvida alguma, uma queda de desempenho em relação a um sistema equivalente sem tais mecanismos. Essa queda se manifesta através de um processamento adicional. De acordo com o tipo de redundância empregado (temporal ou de recursos) os efeitos desse processamento são diferentes. Em mecanismos de redundância temporal existe apenas um componente redundante em atividade em um dado instante. Quando este componente falha, então outro é acionado. Já em mecanismos de redundância de recursos, todos os módulos redundantes estão em atividade ao mesmo tempo. Dessa forma, observa-se que mecanismos de redundância temporal acarretam um aumento de carga no processamento global do sistema menor do que os mecanismos de redundância de recursos. Em contrapartida, mecanismos de redundância de recursos são mais rápidos do que os de redundância temporal

porque não necessitam de aguardar a restauração de um estado prévio e posterior execução de um módulo alternativo. O tempo gasto para a continuação do serviço em mecanismos baseados em redundância temporal pode inviabilizar sua utilização em sistema de tempo-real.

Além da questão da recuperação de erros, cabe também observa-se a questão da sua detecção. As falhas em um sistema podem ser de três tipos: de omissão, de tempo e bizantinas (ver ítem VII.3). Falhas por omissão ou de tempo podem ser detectadas em ambas as classes de tolerância a falhas através de alguma forma de temporização ou protocolo. Já a detecção de falhas bizantinas, ou seja, falhas quaisquer, é feita de forma distinta pelas duas classes. Testes de aceitação, efetuados em blocos de recuperação conseguem, no máximo, estabelecer um intervalo de resultados válidos em uma alternativa. Além disso, o algoritmo do próprio teste de aceitação pode não ser absolutamente independente daqueles das alternativas que valida, tornando-o, assim, impróprios em algumas situações de erro. Mecanismos baseados na redundância de recursos, por outro lado, conseguem alcançar um maior espectro de detecção de falhas bizantinas, embora se saiba que não infinito (ver ítem VII.6.2). Apesar de também não haver, na prática, uma independência total entre os módulos na redundância de recursos, esse problema é menor do que na temporal já que a comparação envolve todos os módulos (ao contrário de em um teste de aceitação onde só estão envolvidos a alternativa que acabou de ser executada e o próprio teste de aceitação.

VII.7. Implementação de um Mecanismo de N-Versões:

O PORTOS-TF é um sistema operacional que implementa um mecanismo de N-versões para tolerância a falhas. A escolha de N-versões ao invés de outros mecanismos como "hot stand-by", "cold stand-by" ou blocos de recuperação deveu-se a dois fatos. O primeiro deles tem a ver com as características de tempo-real do sistema operacional. Como já exposto anteriormente, mecanismos de redundância de recursos são mais apropriados para esse tipo de sistemas do que os de redundância temporal. Em segundo lugar, N-versões foi escolhido porque permite uma conferência mais acurada dos resultados de um módulo.

O PORTOS-TF permite o desenvolvimento dos módulos de software de forma independente de sua utilização ou não em uma configuração tolerante a falhas. Cada módulo, na verdade, funciona como um chip de software. Uma vez projetado, sua operação interna é ignorada pelo sistema. Importam apenas seus "pinos", ou seja, suas interfaces de comunicação. Em tempo de configuração, os processos são criados, seus portos ligados e sua execução iniciada.

Como o PORTOS-TF usa portos como estrutura de comunicação, quando dois processos e suas transações devem ser tolerantes a falhas, então eles devem se comunicar através de um "porto confiável". Esse conceito, similar ao do CONIC (LOQUES, KRAMER & ANIDO, 1988), permite a votação das N-versões antes da entrega da mensagem ao processo destinatário. Dessa forma, o PORTOS-TF implementa uma primitiva de criação de um porto confiável.

A primitiva de criação de um porto confiável necessita, além do número e tipo do porto (entrada ou saída), como na criação de um porto comum (vide capítulo IV), de um parâmetro que indique o algoritmo empregado na

votação. Este parâmetro, na realidade, informa o endereço de uma rotina que executa o algoritmo de votação. O sistema operacional têm dois algoritmos originais: votação byte-a-byte e votação randômica. O primeiro tipo só valida as opções se elas coincidirem byte a byte. O segundo tipo não faz nenhuma comparação, pois assume que o conteúdo da mensagem é resultado de uma geração randômica. Sua única função é detectar falhas no domínio do tempo.

Todavia, se nenhum dos algoritmos inerentes ao PORTOS-TF atender às necessidades de tolerância a falhas do sistema, o projetista poderá indicar outro qualquer de sua autoria, informando o endereço da rotina que o contém. O sistema operacional chama a rotina de votação passando para ele dois parâmetros: o número de opções que devem ser votadas e o endereço do vetor que contém os endereços dos "buffers" que, por sua vez, contém as versões e seus respectivos tamanhos. Dessa forma, a função de votação definida pelo projetista deve respeitar essa interface. As mensagens a serem votadas estarão a partir do elemento 1 do vetor. A posição zero é reservada para uma eventual mensagem obtida, por exemplo, pelo cálculo da média dos valores das mensagens. O PORTOS-TF assume que a aceitação de mensagens "gêmeas" com tamanhos diferentes é de responsabilidade única do algoritmo de votação. Por exemplo, no algoritmo de comparação byte-a-byte, as mensagens devem ter o mesmo tamanho, mas no algoritmo de comparação randômica não há nenhum teste quanto a isso.

A rotina do algoritmo de votação alternativo deverá retornar o valor um, se houver concordância entre as mensagens. Nesse caso, o sistema operacional entregará a primeira delas para ser consumida pelo processo dono do porto de entrada. Caso o algoritmo seja do tipo que escolhe um valor médio como resultado da votação, deverá retornar o valor zero. Além disso, deverá indicar na posição zero do vetor de mensagens o endereço do "buffer"

onde colocou esse valor médio. Caso não haja um resultado majoritário, então o valor retornado pela função do algoritmo de votação deverá ser dois.

Para cada porto confiável de saída associado a um porto confiável de entrada existe um número de seqüência. Esse número é iniciado com o valor zero quando os portos são ligados. Quando uma mensagem vai ser transmitida, o número da seqüência do porto de saída é incrementado e incorporado à mensagem. Quando esta é recebida no porto de entrada, é comparada com o respectivo número de seqüência. Se o número da mensagem for uma unidade superior em relação ao do porto de entrada, então a mensagem será aceita e poderá ser votada. Caso seja igual ou menor, será descartada por representar uma retransmissão. O número da mensagem não pode ser maior do que uma unidade em relação ao número da seqüência do porto de entrada. Ao término da votação, o número da seqüência do porto de entrada será igualado ao da mensagem. No caso de uma resposta, este número será usado para rotular o pacote de resposta. No porto de saída que originou o pedido, um procedimento análogo ao descrito para o porto de entrada será efetuado. A diferença estará no fato de o número da mensagem deverá ser igual ao da seqüência e não uma unidade superior. Alem disso, ao término da votação, não haverá alteração do número de seqüência do porto de saída.

Quando dois portos são ligados, o sistema operacional verifica se ambos são "confiáveis" ou se ambos são comuns. Caso sejam de tipos diferentes, então a ligação não será efetuada.

Quando um processo usa um porto confiável para se comunicar ele estará efetuando uma transação tolerante a falhas automatica e transparentemente.

O PORTOS-TF define uma exceção para o caso de não conseguir ser feita uma escolha majoritária. Nesse caso, essa exceção é levantada e o sistema é interrompido para evitar que isso possa provocar alguma falha no ambiente externo a ele.

CAPÍTULO VIII

CONCLUSÕES

Este trabalho descreve a implementação de um sistema operacional portátil de tempo-real com primitivas de tolerância a falhas: o PORTOS-TF.

Nesta versão, o PORTOS-TF se apresenta implementado em computadores do tipo IBM-PC (vide apêndice A). O ambiente multitarefa criado permite a execução concorrente de vários processos em uma mesma máquina, sendo um em "foreground" e os demais em "background". O sistema operacional nativo é preservado e suas funções podem continuar a ser acessadas pelos processos. Uma vez instalado, o PORTOS-TF fica transparente para um usuário que executa um programa em "foreground" desenvolvido originalmente para o sistema operacional nativo. A criação de processos é feita a partir de chamadas à primitiva adequada executadas por um processo já existente. A cada novo processo é atribuída uma prioridade e um tamanho de fatia de tempo no momento de sua criação. Esses parâmetros, contudo, podem ser alterados a qualquer momento que se julgue conveniente (vide capítulo III).

O mecanismo de comunicação e sincronização adotado é o de troca de mensagens. Portos são usados como estruturas de endereçamento. Primitivas do tipo "envia", "recebe" e "responde" são empregadas nas transações entre os processos, sendo que as duas primeiras podem ainda ser executadas de forma síncrona ou assíncrona. O recebimento de mensagens através dos portos pode se dar de forma múltipla e seletiva (vide capítulo IV). A utilização de portos permite a programação independente dos processos e sua posterior configuração (ou reconfiguração) de acordo com a aplicação.

A implementação de um mecanismo de N-versões (vide capítulo VII) permite que o PORTOS-TF seja usado em aplicações de alta confiabilidade. O projeto de cada uma das versões é feita de forma independente das demais. Cada módulo do sistema ignora se será usado isoladamente ou em uma estrutura replicante. Isso permite a introdução seletiva e transparente de estruturas tolerantes a falhas na aplicação.

Além da discussão das opções de projeto adotadas no núcleo (capítulo III), nos mecanismos de comunicação e sincronização entre processos (capítulo IV) e de tolerância a falhas (capítulo VII), são descritas as estruturas de dados usadas pelo PORTOS-TF (capítulo VI) e de que forma ocupam a área de dados reservada na memória para o sistema operacional (capítulo V). São também efetuadas considerações quanto à portabilidade e reutilização de módulos de software e do sistema operacional (capítulo II). É feito ainda um estudo de mecanismos de tolerância a falhas (em especial falhas de software) (capítulo VII).

Alguns sistemas operacionais nativos não possuem código reentrante. Além deles, é muito comum que as rotinas do BIOS ("Basic I/O System") da máquina também não permitam chamadas recursivas. Nesse caso, tanto as rotinas do sistema operacional nativo quanto as do BIOS constituem-se em recursos compartilhados que devem ser usados de forma exclusiva. O compartilhamento desses recursos não foi considerado pelo PORTOS-TF por ser particular para cada sistema operacional nativo e máquina diferente, mas contitui-se em uma tarefa necessária quando não apenas o processo "foreground" da estação deseja utilizár essas funções. A criação de mecanismos de compartilhamento das rotinas citadas poderia ser sugerida como um trabalho a ser realizado a partir desta tese.

O PORTOS-TF segue um modelo distribuído. Sua implementação em um ambiente desse tipo não foi possível devido à falta de uma arquitetura sobre a qual pudesse ser construído e, também, por sua complexidade. Nessa linha, pode-se efetuar uma extensão do mecanismo de N-versões para um ambiente distribuído. O principal ponto a ser abordado, neste caso, é o projeto e implementação de mecanismos de difusão atômica e ordenação das mensagens.

O desenvolvimento de uma linguagem de programação e configuração dinâmica para sistemas distribuídos baseados na estrutura do PORTOS-TF é, também, uma outra fonte da qual podem ser extraídos trabalhos futuros. Essa linguagem poderia ser totalmente nova ou simplesmente uma extensão de uma linguagem seqüencial já existente.

Uma outra opção de continuação do trabalho ora iniciado é a elaboração de mecanismos de análise de desempenho tanto do sistema operacional em si quanto dos mecanismos de tolerância a falhas objetivando seu aprimoramento.

O transporte do PORTOS-TF para outros tipos de equipamentos pode ser igualmente interessante seja para aplicações isoladas ou em ambientes distribuídos heterogêneos.

REFERÊNCIAS BIBLIOGRÁFICAS

ANDERSON, T. & KNIGHT, J. C., (1981), "Practical Software Fault Tolerance for Real-Time Systems". **Technical Report**, Nr. 169, Computing Laboratory, University of Newcastle upon Tyne, 40 p., England, June.

ANDERSON, T. & LEE, P. A., (1981), "**Fault Tolerance, Principles and Practice**". Prentice-Hall International, Englewood Cliffs, New Jersey, EUA.

ANDERSON, T. & LEE, P. A., (1982), "Fault Tolerance Terminology Proposals". **Technical Report**, Nr. 174, Computing Laboratory, University of Newcastle upon Tyne, 7 p., England, April.

AVIŽIENIS, A. & outros, (1985), "The UCLA DEDIX System: A Distributed Testbed for Multiple-Version Software". **The Fifteenth Annual International Symposium on Fault-Tolerant Computing - FTCS 15**, pp. 126-134, Ann Arbor, Michigan, EUA.

BELLON, C. & SAUCIER, G., (1982), "Protection against External Errors in a Dedicated System". **IEEE Transactions on Computers**, Vol. c-31, No. 4, pp. 311-317, April.

BRINCH HANSEN, P., (1973), "**Operating Systems Principles**". Prentice-Hall, 366p., Englewood Cliffs, New Jersey, EUA.

CAMPBELL, R. H. & RANDELL, B., (1983), "Error Recovery in Asynchronous Systems". **Technical Report**, Nr. 186, 36p., Computing Laboratory, University of Newcastle upon Tyne, England, July.

CAMPBELL, R. H.; ANDERSON, T. & RANDELL, B., (1983), "Practical Fault Tolerant Software for Asynchronous Systems". Technical Report, Nr. 187, 7p., Computing Laboratory, University of Newcastle upon Tyne, England, August.

CHERITON, D. R. & outros, (1979), "THOTH, A Portable Real-Time Operating System". Communications of the ACM, Vol. 22, No. 2, pp. 105-115, February.

COOPER, E. C., (1986), "Replicated Procedure Call". ACM Operating Systems Review, Vol. 20, No. 1, pp. 44-56, January.

CRISTIAN, F., (1982), "Exception Handling and Software Fault Tolerance". IEEE Transactions on Computers, Vol. C-31, No. 6, pp. 531-540, June.

CRISTIAN, F. & outros, (1985), "Atomic Broadcast: from Simple Message Diffusion to Byzantine Agreement". The Fifteenth Annual International Symposium on Fault-Tolerant Computing - FTCS 15, pp. 200-206, Ann Arbor, Michigan, EUA.

CRISTIAN, F., (1987), "Issues in the Design of Highly Available Computing Systems". IBM Research Report RJ5856 (58907) Computer Science, 8p., San Jose - CA, EUA, July.

DEITEL, H. M., (1984), "An Introduction to Operating Systems". Addison-Wesley, 673p., Reading - Mass., EUA.

FANTECHI, A. & outros, (1983), "The Cnet INTERNODE Communication Mechanism". Publicazzione interna del Progetto Finalizzato Informatica C.N.R. Progetto Cnet, 100, 26p., Pisa, Italia, diciembre.

FERREIRA, L. A. A., (1985), "Proposta de uma Arquitetura de um Sistema Operacional de Tempo-Real". Tese de Mestrado, Coordenação dos Programas de Pós-Graduação em Engenharia - COPPE, Universidade Federal do Rio de Janeiro - UFRJ, 159p., Rio de Janeiro, junho.

GUNNINGBERG, P., (1983), "Voting and Redundancy Management Implemented by Protocols in Distributed Systems". The Thirteenth Annual International Symposium on Fault-Tolerant Computing - FTCS 13, pp. 182-185, Milano, Italia.

HOARE, C. A. R., (1972), "Towards a Theory of Paralell Programming". In: Hoare, C. A. R. & Perrot, R. H. (eds.) - Operating Systems Techniques. Academic Press, pp. 61-71, New York, EUA.

HOLT, R. C. & outros, (1978), "Structured Concurrent Programming with Operating Systems Applications". Addison-Wesley, 262p., Reading - Mass., EUA.

HOROWITZ, E. & SAHNI, S., (1984), "Fundamentos de Estruturas de Dados". Campus, 494p., Rio de Janeiro.

IBM, (1987), "DOS 3.30 Technical Reference", International Business Machines Corporation.

JOSEPH, M., (1981), "Schemes for Communication". Technical Report CMU-CS-81-122, Department of Computer Science, Carnegie-Mellon University, 41p., EUA, June.

KERNIGHAN, B. W. & RITCHIE, D. M., (1978), "The C Programming Language". Prentice Hall, New Jersey, EUA.

KIRNER, C., (1986), "Desenvolvimento de Suporte Básico para Sistemas Operacionais Distribuídos". Tese de Doutorado, Coordenação dos Programas de Pós-Graduação em Engenharia - COPPE, Universidade Federal do Rio de Janeiro - UFRJ, 232p., Rio de Janeiro, agosto.

KIRNER, C. & MENDES, S. B. T., (1988), "Sistemas Operacionais Distribuídos: Aspectos Gerais e Análise de sua Estrutura". Campus, 184p., Rio de Janeiro.

KIRRMANN, H. D. & KAUFMANN, F., (1984), "Poolpo - A Pool of Processors for Process Control Applications". IEEE Transactions on Computers, vol. c-33, no. 10, pp. 869-878, October.

KNIGHT, J. C.; LEVESON, N. G. & St. JEAN, L. D., (1985), "A Large Scale Experiment in N-Version Programming". The Fifteenth Annual International Symposium on Fault-Tolerant Computing - FTCS 15, pp. 135-139, Ann Arbor, Michigan, EUA.

KOKAWA, M. & SHINGAI, S., (1982), "Failure Propagating Simulation and Nonfailure Paths Search in Network Systems". Automatica, Vol. 18, No. 3, pp. 335-341.

KRAMER, J. & outros, (1983), "CONIC: An Integrated Approach to Distributed Control Systems". IEE Proceedings, Vol. 130, No. 1, pp. 1-10, January.

KRAMER, J. & MAGEE, J., (1985), "Dynamic Configuration for Distributed Systems". IEEE Transactions on Software Engineering, vol. SE-11, no. 4, pp. 424-436, April.

LAPRIE, J. C, (1985), "Dependable Computing and Fault Tolerance: Concepts and Terminology". The Fifteenth Annual International Symposium on Fault-Tolerant Computing - FTCS 15, pp. 2-11, Ann Arbor, Michigan,

EUA.

LEITE, J. C. B. & ARIBE, A. R., (1988), "Recuperação de Erros em Sistemas de Processos Concorrentes: uma Proposta". PUC, Rio de Janeiro.

LISKOV, B., (1979), "Primitives for Distributed Computing". Proceedings of the Seventh Symposium on Operating Systems Principles, pp. 33-42, Pacific Grove California, EUA, December.

LOMET, D. B., (1977), "Process Structuring, Syncronization, and Recovery Using Atomic Actions". ACM SIGPLAN Notices (Proceedings of an ACM Conference on Language Design for Reliable Software), Vol. 12, No. 3, pp. 128-137, March.

LOQUES Fo., O. G. & KRAMER, J., (1986), "Flexible Fault Tolerance for Distributed Computer Systems". IEE Proceedings, vol. 133, Pt. E, No. 6, pp. 319-332, November.

LOQUES Fo., O. G., (1988), "Replicação Automática e Transparente de Módulos em Sistemas Distribuídos". PUC, Rio de Janeiro.

LOQUES Fo., O. G.; KRAMER, J. & ANIDO, R., (1988), "Diverse and Selective Fault-Tolerance in a Distributed Environment". 8th IFAC Workshop on Distributed Computer Control Systems, Vitznav - Switzerland, September.

MADNICK, S. E. & DONOVAN, J. J., (1974), "Operating Systems". McGraw-Hill, New York, N.Y., EUA.

MAO, T. W. & YEH, R. T., (1980), "Communication Port: A Language Concept for Concurrent Processing". IEEE

Transactions on Software Engineering, vol. SE-6, no. 2, pp. 194-204, March.

MELLIAR-SMITH, P. M. & RANDELL, B., (1977), "Software Reliability: The Role of Programmed Exception Handling". ACM SIGPLAN Notices (Proceedings of an ACM Conference on Language Design for Reliable Software), Vol. 12, No. 3, pp. 95-100, March.

MENDES, S., (1984), "Programação Concorrente: Mecanismos de Comunicação e Sincronização de Processos". Quarta Escola de Computação, Instituto de Matemática e Estatística - USP, 182p., São Paulo - SP, julho.

POWELL, D. R. & DESWARTE, Y., (1985), "Candidate Solutions Proposed for OTV and DSP Missions". Research Report, no. 85.277, Centre National de la Recherche Scientifique - Laboratoire d'Automatique et d'Analyse des Systemes - LAAS, 16p., Toulouse, France, October.

POWELL, D. R., (1987), "Fault-Tolerance in Delta-4". Research Report, no. 87.064, Centre National de la Recherche Scientifique - Laboratoire d'Automatique et d'Analyse des Systemes - LAAS, 19p., Toulouse, France, February.

QUANTUM, (1988), "QNX - Reference Guide", version 2.1, Quantum Software Systems Ltd., Ontario, Canada, March.

RANDELL, B., (1975), "System Structure for Software Fault Tolerance". Technical Report, Nr. 75, Computing Laboratory, University of Newcastle upon Tyne, 15p., England, May.

RANDELL, B.; LEE, P. A. & TRELEAVEN, P. C., (1978), "Reliability Issues in Computing System Design". ACM Computing Surveys, Vol. 10, No. 2, pp. 123-165, June.

REID, L. G., (1980), "Control and Communication in Programmed Systems". Ph.D. Thesis, Department of Computer Science, Carnegie-Mellon University, 148p., EUA, September. (Technical Report CMU-CS-80-142).

RUGGIERO, W. V. & BRESSAN, G., (1982), "Um Modelo de Programação para Sistemas Distribuídos". **Revista Brasileira de Computação**, vol. 2, no. 3, pp. 131-149, setembro.

SANTOS, J. R. G., (1987), "Sistemas Distribuídos Aplicados à Automação Industrial". **Simpósio sobre Sistemas Distribuídos**, pp. 129-142, Gramado - RS, agosto.

SCHWARZ, G., (1981), "**Computadores em Tempo-Real: Uma Introdução aos Conceitos Básicos**". COPPE/UFRJ, 493p., Rio de Janeiro.

SEGRE, L. M., (1987), "**Um Estudo de Ambientes de Programação Distribuída: Proposta de Extensões para MODULA-2**". Tese de Doutorado, Coordenação dos Programas de Pós-Graduação em Engenharia - COPPE, Universidade Federal do Rio de Janeiro - UFRJ, 338p., Rio de Janeiro, dezembro.

SHRIVASTAVA, S. K., (1979), "Structuring Distributed Systems for Recoverability and Crash Resistance". **Technical Report**, Nr. 149, Computing Laboratory, University of Newcastle upon Tyne, 26p., England, December.

SHRIVASTAVA, S. K. & PANZIERI, F., (1981), "The Design of a Reliable Remote Procedure Call Mechanism ". **Technical Report**, Nr. 171, Computing Laboratory, University of Newcastle upon Tyne, 18p., England, October.

SILBERSCHATZ, A., (1981), "Port Directed Communication". The Computer Journal, vol. 24, no. 1, pp. 78-82, February.

SLOMAN, M.; KRAMER, J. & MAGEE, J., (1986), "The CONIC Toolkit for Building Distributed Systems". 4o. Simpósio Brasileiro de Redes de Computadores, pp. 148-157, Recife - PE, março.

SOUZA, M. L. D., (1987), "Um Sistema Operacional de Rede". Anais do XX Congresso Nacional de Informática, São Paulo - SP, pp. 860-864, setembro.

TSICHRITZIS, D. C. & BERNSTEIN, P. A., (1974), "Operating Systems". Academic Press, 298p., New York, EUA.

WALTER, C. J.; KIECKHAFER, R. M. & FINN, A. M., (1985), "MAFT: A Multicomputer Architecture for Fault-Tolerance in Real-Time Control Systems". IEEE, pp. 133-140.

APÊNDICE A

IMPLEMENTAÇÃO EM EQUIPAMENTOS DO TIPO IBM-PC

A.1. Introdução:

Computadores do tipo IBM-PC têm tido uma aceitação cada vez maior no mercado. Sendo utilizado originalmente em escritórios, esse tipo de máquina já apresenta um emprego bastante difundido também na automação industrial e na área de controle de processos. Vários tipos de interfaces industriais para esse tipo de equipamento como, por exemplo, interfaces de entradas e saídas digitais e analógicas, são encontradas no mercado. Computadores do tipo IBM-PC construídos de acordo com as normas para aplicações em ambientes industriais já são também fabricados, inclusive pela indústria nacional.

Pela sua grande popularidade, existe uma imensa quantidade de programas que são executados sob o sistema operacional DOS (IBM, 1987), ou compatível. Nesse apêndice, referências ao sistema operacional DOS englobam seus compatíveis. Doravante, qualquer uso da expressão "sistema operacional" será relativa ao PORTOS-TF.

Encontram-se disponíveis no mercado diversos compiladores da linguagem "C" para essa máquina sob o sistema operacional DOS. Além dos compiladores, depuradores bastante poderosos também podem ser empregados nesse ambiente. Dessa forma, as ferramentas de desenvolvimento de software em computadores do tipo IBM-PC sob o sistema operacional DOS permitem a eleboração de programas, inclusive na categoria de software básico, com um relativo conforto.

Por esses motivos, computadores do tipo IBM-

PC sob o sistema operacional DOS foram escolhidos como ambiente de desenvolvimento dessa primeira implementação do PORTOS-TF.

A.2. Operação em Conjunto com o DOS:

Chama-se de "sistema operacional nativo" ao sistema operacional original de um computador. Esse sistema operacional pode ser monotarefa e gerenciar apenas os recursos ligados ao computador onde se encontra. Ele, geralmente, provê um sistema de gerenciamento de arquivos. A operação em conjunto de um sistema operacional distribuído e o sistema operacional nativo do computador não é uma idéia nova. De acordo com SOUZA (1987), ao se manter o sistema operacional nativo, o ambiente original é preservado e, por conseqüência, programas desenvolvidos para o ambiente original continuam a ser executáveis no novo ambiente.

Como exposto no capítulo I, o PORTOS-TF não provê nenhum sistema de gerenciamento de arquivos. Ele assume que todo o software, básico e aplicativo, está carregado na memória primária da estação. A partir desse fato, o PORTOS-TF é instalado no computador e entra em operação sem desativar o sistema operacional nativo, isto é, o DOS.

A operação em conjunto do PORTOS-TF e do DOS ocorre da seguinte forma. O PORTOS-TF foi desenvolvido, usando-se as ferramentas disponíveis para o sistema operacional nativo (compilador "C", depurador, etc.). Dessa forma, foi gerado um programa que tem, em linhas gerais, o seguinte comportamento para efeitos de instalação. Em primeiro lugar, faz as iniciações necessárias (variáveis, tabelas, vetores de interrupção, etc.). Em seguida, chama a função número 31h (em

hexadecimal) do vetor de funções do DOS localizado na interrupção de software número 21h (IBM, 1987). Essa função termina a execução do programa, mas deixa seu código residente na memória. A partir daí, todo acesso ao núcleo do sistema operacional será feito pela geração de uma interrupção de software. A interrupção escolhida foi a de número 60h. Essa interrupção é usada pelos processos do sistema quando querem solicitar algum serviço ao PORTOS-TF. Uma outra forma de ser executado o código do sistema operacional é através da interrupção do relógio de tempo-real.

Quando o PORTOS-TF é instalado, ficam cadastrados junto ao núcleo dois processos: o processo "ocioso" e o processo "foreground". O processo "foreground" corresponde, logo após a instalação, ao código do DOS de interação com o usuário da estação. O fato do PORTOS-TF estar residente e operando passa desapercebido ao usuário. Para ele, tudo se passa como se o sistema operacional da máquina fosse apenas o DOS. O único efeito que pode ser sentido é alguma eventual queda de desempenho quando vários processos estiverem "prontos", competindo pelo processador. Dessa forma, quando um programa (um editor de textos, um compilador, um aplicativo qualquer, etc.) é executado, ele passa a ser encarado pelo PORTOS-TF como se fosse uma subrotina do processo "foreground". Quando o programa termina, é como se o "foreground" tivesse retornado da subrotina. Isso quer dizer que, para o sistema operacional (PORTOS-TF), os procedimentos de carga, execução e término de um programa são encarados como desvios no fluxo de execução do processo "foreground".

A vantagem adicional que se tira ao manter o DOS ativo é, como mencionado anteriormente, permitir que programas de mercado sejam executados sob o PORTOS-TF como processos em "foreground". Dessa forma, informações obtidas por um processo de aquisição de dados, por exemplo,

podem ser armazenadas em arquivos no formato do DOS para posterior manuseio através de pacotes de mercado.

A.3. Aspectos da Implementação:

A.3.1. O Relógio de Tempo-Real:

Computadores do tipo IBM-PC apresentam uma rotina original de tratamento de interrupções de relógio de tempo-real. O relógio de tempo-real gera uma interrupção que está mapeada no endereço 8 (oito) do vetor de interrupções do computador. Essa rotina executa um conjunto de procedimentos, dentre os quais está o rearme da 8259 (pastilha controladora de interrupções) através do envio de um EOI ("End Of Interrupt"). Um outro procedimento executado é a geração de uma interrupção de software de número 1Ch (1C hexadecimal). A rotina de atendimento a essa interrupção executa apenas uma instrução de retorno. O objetivo dessa chamada é permitir que o usuário defina, nesse endereço, uma rotina sua que necessite de uma base de tempo que, no caso, é fornecida pelo relógio de tempo-real.

A rotina de tratamento do relógio pelo PORTOS-TF foi colocada originalmente no lugar da interrupção de software 1Ch. Observou-se, contudo, que o sistema se perdia. Após alguns testes, verificou-se que a rotina original do computador mudava de pilha antes de chamar a interrupção 1Ch. Como uma interrupção do relógio de tempo-real pode causar um escalonamento (por tempo), a pilha tomada como sendo do processo era na realidade a da rotina original da máquina. Como essa pilha era a mesma em todas as vezes que ocorria uma interrupção do relógio, independentemente do processo que estava no estado "rodando", essa pilha era danificada causando o descontrole do sistema.

A solução encontrada foi, então, colocar a rotina do PORTOS-TF antes da original. Dessa forma, o endereço da rotina original foi mudado para a posição da interrupção de software 61h (uma além da rotina de chamada de funções do núcleo do sistema operacional). Na posição da interrupção 8 (oito), foi assinalado o endereço da rotina de tratamento do relógio do PORTOS-TF. Essa rotina, por sua vez, passou a gerar uma interrupção de software de número 61h para que os procedimentos originais (inclusive o rearme da 8259) fossem efetuados.

A.3.2. Troca de Contexto:

A troca de contexto, isto é, o salvamento do contexto de um processo e posterior restauração daquele do processo seguinte na Lista dos Prontos é uma tarefa particular para cada tipo de processador diferente.

Os computadores do tipo IBM-PC são baseados em processadores da família do 8088, da Intel. Assim, existem computadores desse tipo equipados com as UCPs 8088, 80286 e 80386. Essa família de processadores apresenta uma arquitetura baseada em endereçamento segmentado. Cada endereço é composto por um par de registradores: um contendo o segmento e outro o deslocamento dentro do segmento. São os seguintes os registradores dos processadores dessa família:

. AX:
  acumulador;
. BX, CX e DX:
  uso geral;
. SI e DI:
  ponteiros para seqüências de bytes;
. DS:
  segmento da área de dados;

. ES:

endereço auxiliar de segmento;

. SS e SP:

segmento e deslocamento, respectivamente, do topo da pilha;

. BP:

ponteiro auxiliar para manuseiro de dados na pilha;

. CS e PC:

segmento e deslocamento, respectivamente, da próxima instrução a ser executada e

. "flags".

Quando o processador aceita um pedido de interrupção, ele empilha os "flags" o ponteiro de instruções e o segmento da área de código. Na entrada da rotina de tratamento dessa interrupção (seja de relógio, ou de entrada no núcleo), os demais registradores são empilhados. Finalmente, os registradores ponteiros da pilha ("SS" de segmento e "SP" de deslocamento) são salvos, no descritor do processo que foi interrompido, para posterior recuperação. O procedimento de restauração do contexto segue o caminho inverso.

A.3.3. Uso de Ponteiros e Endereços:

Dada à arquitetura de registradores apresentada no ítem anterior, ponteiros e endereços podem ser de dois tipo básicos: próximos ou distantes. Ponteiros ou endereços próximos são aqueles que apontam para dados ou funções estão dentro do mesmo segmento. Por conseguinte, são referenciados apenas pelo valor do deslocamento. Ponteiros e endereços distantes, em contrapartida, se referenciam a dados ou funções que estão localizados em quaisquer segmentos (inclusive o próprio). Dessa forma, estes são referenciados por um par de registradores do tipo

"segmento:deslocamento".

Por esse motivo, algumas definições de tipos se encontram em arquivos "dependentes da máquina". Em outro tipo de arquitetura, esses tipos poderão ser redefinidos.

O núcleo do PORTOS-TF foi todo compilado no modo "small" do compilador, isto é, todos os seus ponteiros e endereços são do tipo próximo. Como exceção tem-se apenas aqueles que se referenciam a dados ou funções pertencentes aos processos do sistema, uma vez que estes se encontram em segmentos diferentes daquele do núcleo.

Nas chamadas ao núcleo do sistema operacional, por exemplo, é passado como parâmetro um ponteiro para uma estrutura de parâmetros para o núcleo (vide capítulo III). No caso dos equipamentos do tipo IBM-PC, esse ponteiro é do tipo "distante" pois a estrutura que aponta está em outro segmento, o do processo chamador.

A.3.4. O "Buffer Pool" do Sistema:

Originalmente, o "buffer pool" do sistema operacional empregado na alocação dinâmica de áreas de estruturas e áreas de mensagens foi definido como sendo um vetor de caracteres de tamanho predeterminado. Ocorre que esse tamanho podia ser insuficiente para algumas aplicações, uma vez que o número de estruturas do tipo descritor de processos, nó de árvore, etc. depende do número de processos existentes no sistema em um determinado instante. A solução adotada foi, obviamente, permitir que esse tamanho fosse alterado em função da aplicação.

O tamanho de um vetor só pode ser alterado em tempo de compilação. Dessa forma, essa opção de definir

o "pool" através de um vetor de caracteres tornou-se inviável. A utilização da primitiva de alocação de área do DOS apresentava o problema de que essa área teria que ser referenciada através de ponteiros do tipo "far". Foi, então, escolhida a opção de se utilizar a área de "heap" deixada pelo compilador logo após a área de variáveis não iniciadas. Uma vez que esta área é alocada através da chamada da primitiva "malloc" da biblioteca do compilador, ela pode ser parametrizável em tempo de execução e não mais em tempo de compilação, como no caso original.

A.3.5. Escrita em Vídeo:

A biblioteca padrão da linguagem "C" possui uma série de funções de escrita em vídeo. O uso dessas funções, contudo, acarreta um aumento significativo do código do programa que as usa. Mais crítico do que isso é que elas utilizam a interrupção de software de número 10h do BIOS ("Basic I/O System") da máquina. O problema é que a primeira instrução da rotina de tratamento dessa interrupção é uma instrução de habilitação de aceitação de inperrupção pela UCP. Além disso, essa rotina é muito lenta. Na escrita de uma frase, por exemplo, ela é chamada várias vezes. Dessa forma, se o tempo de escrita for maior do que o período do relógio (que nos IBM-PC é de aproximadamente 55 ms) o código do núcleo será interrompido. Como ele não é reentrante, pois usa variáveis globais, o sistema poderá se perder.

Para solucionar esse problema e permitir que o código do núcleo do sistema operacional escrevesse na tela (o que era feito em duas ocasiões: durante procedimentos de depuração ou de exceção) foram desenvolvidas algumas funções. Elas empregam a política de escrever diretamente na memória de vídeo do computador, que fica a partir do endereço absoluto de memória B8000h. Esse

tipo de tela, em sua configuração original, é composta de 25 linhas e 80 colunas. Cada caractere é armazenado em dois bytes (um com o caractere propriamente dito e outro com seus atributos: cor do caractere, cor do fundo e indicação de se deve piscar ou não).

Essas rotinas, como pode-se notar, são extremamente dependentes não só da máquina como também da configuração desta. Desse modo, sempre que a configuração mudar, elas poderão ser reescritas.

APÊNDICE B

PRIMITIVAS DO NÚCLEO DO PORTOS-TF

Nesse apêndice são descritos os protótipos das rotinas de interface com o núcleo do PORTOS-TF. Essas rotinas estão escritas na linguagem "C".

No ítem B.1 são explicados os tipos definidos que não são padrões da linguagem "C". Esses tipos correspondem a tipos padrões do "C" (unsigned, char, etc.). Nesse ítem serão mostradas as correspondências usadas na implementação em computadores do tipo IBM-PC. Essa correspondência, todavia, é dependente da máquina e pode ser alterada quando o sistema operacional for migrado para outro tipo de equipamento.

No ítem B.2 são, então, apresentados os protótipos das primitivas acompanhados de sua função e descrição dos parâmetros e valor de retorno.

B.1. Explicação dos Tipos não-padrões Empregados:

- Tipo: BOOLEAN
  Significado: armazena valores "booleanos" (0 ou 1).
  Definição: enumeração (FALSE, TRUE).

- Tipo: FAR
  Significado: indica que o tipo a ele adjacente é do tipo "longo", ou seja, está em outro segmento diferente do corrente. Esse tipo não faz sentido em arquiteturas que não trabalham com o conceito de segmento.
  Definição: far.

- Tipo: SENTIDO_PORTO
  Significado: Indica o sentido do porto.
  Definição: enumeração (SAIDA, ENTRADA).

- Tipo: TIPO_FATIA_TEMPO
  Significado: indica o numero de pulsos do relógio que dura uma fatia de tempo de um processo.
  Definição: unsigned

- Tipo: TIPO_ID_PROC
  Significado: indica um numero de um processo.
  Definição: unsigned

- Tipo: TIPO_PILHA
  Significado: armazena o valor do ponteiro para o topo da pilha de um processo.
  Definição: struct contexto far

- Tipo: TIPO_PRI
  Significado: indica a prioridade de um processo.
  Definição: unsigned char

- Tipo: TIPO_PROCESSO
  Significado: indica o endereço inicial de um processo.
  Definição: void far (*func)()
- Tipo: TIPO_ROT_SERV
  Significado: indica o endereço inicial de uma rotina de serviço.
  Definição: void far (*func)()

- Tipo: TIPO_STATUS_COMUNICACAO

  Significado: Indica o resultado de uma operação de comunicação (envia, recebe ou responde).

  Definição: enumeração (OK,
  PORTO_INEXISTENTE,
  SENTIDO_INCOMPATIVEL,
  NAO_HA_MENSAGEM,
  BUFFER_INSUFICIENTE,
  PEDIDO_JA_ATENDIDO,
  FIM_TEMPO_ESPERA)

- Tipo: TIPO_TEMPO_ESPERA

  Significado: indica o numero máximo de pulsos do relógio que um processo aceita ficar bloqueado.

  Definição: unsigned

B.2. Protótipos da Primitivas do Núcleo:

- Nome: cria_processo

  Função: criar um novo processo.

  Protótipo:

```
TIPO_ID_PROC
cria_processo (
    TIPO_PROCESSO          proc,
    TIPO_PRI               prioridade,
    TIPO_FATIA_TEMPO       fatia,
    TIPO_PILHA             pilha,
    unsigned               dados );
```

  Parâmetros:

  . proc: endereço inicial do processo

  . prioridade: prioridade do processo

  . fatia: número de pulsos do relógio da fatia de tempo

  . pilha: endereço da base da área reservada para a pilha do sistema

  . dados segmento inicial da área de dados

  Valor de Retorno: identificador do processo criado. Um valor nulo indica que a operação não foi bem sucedida.

- Nome: mata_processo

  Função: eliminar um processo existente.

  Protótipo:

```
TIPO_ID_PROC
mata_processo(
    TIPO_ID_PROC   proc );
```

  Parâmetros:

  . proc: identificador do processo

  Valor de Retorno: identificador do processo eliminado. Um valor nulo indica que a operação não foi bem sucedida.

- Nome: eu

  Função: obter o número de um processo existente.

  Protótipo:

```
TIPO_ID_PROC
eu( );
```

  Parâmetros:

  NENHUM

  Valor de Retorno: identificador do processo chamador

- Nome: bloqueia_processo

  Função: bloquear um processo existente.

  Protótipo:

```
TIPO_ID_PROC
bloqueia_processo(
    TIPO_ID_PROC           proc,
    TIPO_TEMPO_ESPERA      tempo,
    TIPO_ROT_SERV          rot_serv );
```

  Parâmetros:

  . proc: identificador do processo

  . tempo: tempo máximo de espera

  . rot_serv: endereço da rotina de serviço

  Valor de Retorno: identificador do processo bloqueado. Um valor nulo indica que a operação não foi bem sucedida.

- Nome: libera_processo

  Função: liberar um processo existente.

  Protótipo:

```
TIPO_ID_PROC
libera_processo(
    TIPO_ID_PROC           proc );
```

  Parâmetros:

  . proc: identificador do processo

  Valor de Retorno: identificador do processo liberado. Um valor nulo indica que a operação não foi bem sucedida.

- Nome: pega_prioridade

  Função: obtém a prioridade de um processo existente.

  Protótipo:

```
TIPO_PRI
pega_prioridade(
        TIPO_ID_PROC                proc );
```

  Parâmetros:

  . proc: identificador do processo

  Valor de Retorno: prioridade do processo. Um valor nulo indica que a operação não foi bem sucedida.

- Nome: muda_prioridade

  Função: altera a prioridade de um processo existente.

  Protótipo:

```
TIPO_PRI
muda_prioridade(
        TIPO_ID_PROC                proc,
        TIPO_PRI                    prioridade );
```

  Parâmetros:

  . proc: identificador do processo

  . prioridade: nova prioridade do processo

  Valor de Retorno: prioridade anterior do processo. Um valor nulo indica que a operação não foi bem sucedida.

- Nome: pega_fatia

  Função: obtém a fatia de tempo de um processo existente.

  Protótipo:

```
TIPO_FATIA_TEMPO
pega_fatia(
        TIPO_ID_PROC                proc );
```

  Parâmetros:

  . proc: identificador do processo

  Valor de Retorno: fatia de tempo do processo. Um valor nulo indica que a operação não foi bem sucedida.

- Nome: muda_fatia

  Função: altera a fatia de tempo de um processo existente.

  Protótipo:

```
TIPO_FATIA_TEMPO
muda_fatia(
     TIPO_ID_PROC          proc,
     TIPO_FATIA_TEMPO      fatia );
```

  Parâmetros:

  . proc: identificador do processo

  . fatia: nova fatia de tempo do processo

  Valor de Retorno: fatia de tempo anterior do processo. Um valor nulo indica que a operação não foi bem sucedida.

- Nome: congela_fatia

  Função: suspende a contagem de tempo da fatia de tempo do processo corrente.

  Protótipo:

```
void
congela_fatia( );
```

  Parâmetros:

  NENHUM

  Valor de Retorno: NENHUM

- Nome: descongela_fatia

  Função: retoma a contagem de tempo da fatia de tempo do processo corrente.

  Protótipo:

```
void
descongela_fatia( );
```

  Parâmetros:

  NENHUM

  Valor de Retorno: NENHUM

- Nome: habilita_excecao

  Função: Autoriza o sistema operacional a gerar uma dada exceção.

  Protótipo:

  ```
  void
  habilita_excecao(
        unsigned   tipo );
  ```

  Parâmetros:

  . tipo: tipo da exceção

  Valor de Retorno: NENHUM

- Nome: desabilita_excecao

  Função: Impede o sistema operacional de gerar uma dada exceção.

  Protótipo:

  ```
  void
  desabilita_excecao(
        unsigned   tipo );
  ```

  Parâmetros:

  . tipo: tipo da exceção

  Valor de Retorno: NENHUM

- Nome: habilita_escalonamento

  Função: Autoriza o sistema operacional a efetuar um escalonamento.

  Protótipo:

  ```
  void
  habilita_escalonamento( );
  ```

  Parâmetros:

  NENHUM

  Valor de Retorno: NENHUM

- Nome: desabilita_escalonamento

  Função: Desautoriza o sistema operacional a efetuar um escalonamento.

  Protótipo:

```
void
desabilita_escalonamento( );
```

  Parâmetros:

  NENHUM

  Valor de Retorno: NENHUM

- Nome: desabilita_excecao

  Função: Impede o sistema operacional de gerar uma dada exceção.

  Protótipo:

```
void
desabilita_excecao(
      unsigned   tipo );
```

  Parâmetros:

  . tipo: tipo da exceção

  Valor de Retorno: NENHUM

- Nome: escalona

  Função: Cede o processador.

  Protótipo:

```
void
escalona( );
```

  Parâmetros:

  NENHUM

  Valor de Retorno: NENHUM

- Nome: cria_porto

  Função: Cria um novo porto.

  Protótipo:

```
BOOLEAN
cria_porto(
     unsigned        numero,
     TIPO_ID_PROC    proc,
     SENTIDO_PORTO   sentido );
```

  Parâmetros:

  . numero: número do porto
  . proc: identificador do processo ao qual pertence o porto
  . sentido: sentido do porto

  Valor de Retorno: Verdadeiro, se o porto foi criado. Falso se não.

- Nome: elimina_porto

  Função: Elimina um porto existente.

  Protótipo:

```
BOOLEAN
elimina_porto(
     unsigned        numero,
     TIPO_ID_PROC    proc );
```

  Parâmetros:

  . numero: número do porto
  . proc: identificador do processo ao qual pertence o porto

  Valor de Retorno: Verdadeiro, se o porto foi criado. Falso se não.

- Nome: liga_porto

  Função: Liga dois portos existentes.

  Protótipo:

```
BOOLEAN
liga_porto(
    unsigned        porto_1,
    TIPO_ID_PROC    proc_porto_1,
    unsigned        porto_2,
    TIPO_ID_PROC    proc_porto_2 );
```

  Parâmetros:

  . porto_1: número de um dos portos
  . proc_porto_1: identificador do processo ao qual pertence esse porto
  . porto_2: número do outro portos
  . proc_porto_2: identificador do processo ao qual pertence esse outro porto

  Valor de Retorno: Verdadeiro, se o porto foi criado. Falso se não.

- Nome: desliga_porto

  Função: Desliga dois portos existentes.

  Protótipo:

```
BOOLEAN
desliga_porto(
    unsigned        porto_1,
    TIPO_ID_PROC    proc_porto_1,
    unsigned        porto_2,
    TIPO_ID_PROC    proc_porto_2 );
```

  Parâmetros:

  . porto_1: número de um dos portos
  . proc_porto_1: identificador do processo ao qual pertence esse porto
  . porto_2: número do outro portos
  . proc_porto_2: identificador do processo ao qual pertence esse outro porto

  Valor de Retorno: Verdadeiro, se o porto foi criado. Falso se não.

- Nome: envia

  Função: Escreve uma mensagem em um porto.

  Protótipo:

```
TIPO_STATUS_COMUNICACAO
envia(
    unsigned            porto,
    char FAR            *mens,
    unsigned            tam_mens,
    char FAR            *resp,
    unsigned            tam_resp,
    TIPO_TEMPO_ESPERA   t_esp );
```

  Parâmetros:

  . porto: número do porto

  . mens: ponteiro para o "buffer" que contém a mensagem

  . tam_mens: tamanho da mensagem

  . resp: ponteiro para o "buffer" onde será copiada a resposta da mensagem

  . tam_resp: tamanho do "buffer" da resposta

  . t_esp: tempo de espera pela resposta

  Valor de Retorno: Indica o estado da operação (vide definição do tipo "TIPO_STATUS_COMUNICACAO").

- Nome: recebe

  Função: Lê uma mensagem de um porto.

  Protótipo:

```
TIPO_STATUS_COMUNICACAO
recebe(
    unsigned              porto,
    char FAR              *mens,
    unsigned              tam_mens,
    TIPO_TEMPO_ESPERA     t_esp );
```

  Parâmetros:

  . porto: número do porto
  . mens: ponteiro para o "buffer" onde será copiada a mensagem
  . tam_mens: tamanho da mensagem
  . t_esp: tempo de espera pela resposta

  Valor de Retorno: Indica o estado da operação (vide definição do tipo "TIPO_STATUS_COMUNICACAO").

- Nome: responde

  Função: Envia uma resposta a uma mensagem.

  Protótipo:

```
TIPO_STATUS_COMUNICACAO
responde(
    unsigned              porto,
    char FAR              *resp,
    unsigned              tam_resp )
```

  Parâmetros:

  . porto: número do porto
  . resp: ponteiro para o "buffer" onde está a resposta
  . tam_resp: tamanho da resposta

  Valor de Retorno: Indica o estado da operação (vide definição do tipo "TIPO_STATUS_COMUNICACAO").

- Nome: selec

  Função: Abre um bloco de seleção.

  Protótipo:

  ```
  void
  selec( )
  ```

  Parâmetros:

  NENHUM

  Valor de Retorno: NENHUM

- Nome: fim_selec

  Função: Fecha um bloco de seleção.

  Protótipo:

  ```
  TIPO_STATUS_COMUNICACAO
  fim_selec(
      TIPO_TEMPO_ESPERA      t_esp );
  ```

  Parâmetros:

  . t_esp: tempo de espera por uma mensagem.

  Valor de Retorno: Indica o estado da operação (vide definição do tipo "TIPO_STATUS_COMUNICACAO").

- Nome: porto_ultima_mensagem

  Função: Informa o porto onde chegou a mensagem mais recente.

  Protótipo:

  ```
  unsigned
  porto_ultima_mensagem( );
  ```

  Parâmetros:

  NENHUM.

  Valor de Retorno: número do porto que recebeu a mensagem.

- Nome: estado_sistema

  Função: Dar informações sobre o estado corrente do sistema.

  Protótipo:

```
void
estado_sistema(
    struct estado_sistema FAR     *estado );
```

  Parâmetros:

  . estado: ponteiro para uma estrutura onde serão copiadas as informações.

  Valor de Retorno: NENHUM

APÊNDICE C
COMENTÁRIOS SOBRE AS REFERÊNCIAS BIBLIOGRÁFICAS

Nesse apêndice, são apresentados comentários sobre os textos relacionados nas referência bibliográficas que têm como objetivo detalhar, um pouco mais do que o título, o escopo da publicação. No ítem C.1 são comentados os textos ligados a sistemas operacionais, enquanto que no ítem C.2 se acham os trabalhos referentes a tolerância a falhas. No ítem C.3, estão as demais referências.

C.1. Textos Relativos a Sistemas Operacionais:

BRINCH HANSEN (1973):

"Operating Systems Principles".

Livro introdutório aos sistemas operacionais.

CHERITON & outros (1979):

"THOTH, A Portable Real-Time Operating System".

Expõe os aspectos técnicos de um Sistema Operacional de Tempo-Real implementado com o objetivo de ser portátil. É dada ênfase à estruturação do sistema operacional quanto aos pontos dependentes e independentes do hardware, definindo-se uma "máquina padrão". Percebe-se que a linguagem de programação adotada é de fundamental importância para a definição dessa máquina.

DEITEL (1984):

"An Introduction to Operating Systems".

Livro introdutório aos sistemas operacionais. Aborda uma vasta gama de tópicos nesta área, terminando pelo estudo de casos de alguns sistemas operacionais comerciais.

FANTECHI & outros (1983):

"The Cnet INTERNODE Communication Mechanism".

Aborda o emprego portos na comunicação e sincronização entre processos.

FERREIRA (1985):

"Proposta de uma Arquitetura de um Sistema Operacional de Tempo-Real".

Propõe uma arquitetura de um Sistema Operacional de Tempo-Real baseada no conceito de Máquinas Virtuais.

HOARE (1972):

"Towards a Theory of Paralell Programming".

Elabora uma proposição sobre gerenciamento de regiões críticas e regiões críticas condicionais.

HOLT & outros (1978):

"Structured Concurrent Programming with Operating Systems Applications".

Aborda o tema da concorrência ilustrando sua conveniência e os problemas dela originados. Comenta a estruturação de um sistema operacional e discute uma implementação.

JOSEPH (1981):

"Schemes for Communication".

Aborda o emprego portos na comunicação e sincronização entre processos.

KIRNER (1986):

"Desenvolvimento de Suporte Básico para Sistemas Operacionais Distribuídos".

Discute a comunicação e sincronização entre processos, apresentando um projeto de um subsistema de comunicação.

KIRNER & MENDES (1988):

"Sistemas Operacionais Distribuídos: Aspectos Gerais e Análise de sua Estrutura".

Aborda aspectos conceituais relativos a Sistemas Operacionais Distribuídos, dando ênfase aos mecanismos de comunicação e sincronização entre processos e linguagens de programação concorrente.

KIRRMANN & KAUFMANN (1984):

"Poolpo - A Pool of Processors for Process Control Applications".

Descreve a implementação de um "pool" de

processadores, fortemente acoplados, para aplicação em controle de processos em tempo-real. Expõe as características do núcleo de tempo-real desenvolvido para essa arquitetura. Discute prioridade e seus efeitos na preempção dos processos. Adota a opção de execução cíclica dos processos.

KRAMER & outros (1983):

"CONIC: An Integrated Approach to Distributed Control Systems".

Aborda o emprego portos na comunicação e sincronização entre processos.

KRAMER & MAGEE (1985):

"Dynamic Configuration for Distributed Systems".

Após uma introdução sobre configuração estática e dinâmica de sistemas distribuídos ou não, apresenta as características necessárias e desejáveis de uma linguagem de programação e um sistema operacional para permitir a reconfiguração dinâmica de um sistema de software. Descreve, ainda, as primitivas disponíveis no CONIC para esse fim.

LISKOV (1979):

"Primitives for Distributed Computing".

Aborda o emprego portos na comunicação e sincronização entre processos.

MADNICK & DONOVAN (1974):

"Operating Systems".

Livro introdutório aos sistemas operacionais.

MAO & YEH (1980):

"Communication Port: A Language Concept for Concurrent Processing".

Aborda o emprego portos na comunicação e sincronização entre processos.

MENDES (1984):

"Programação Concorrente: Mecanismos de Comunicação e Sincronização de Processos".

Explora o tema de Programação Concorrente estudando mecanismos de comunicação e sincronização entre processos. É mostrada a aproximação crescente entre sistemas operacionais e as linguagens de programação, na medida em que estas vêm incorporando características providas pelo sistema operacional.

REID (1980):

"Control and Communication in Programmed Systems".

Aborda o emprego portos na comunicação e sincronização entre processos.

RUGGIERO & BRESSAN (1982):

"Um Modelo de Programação para Sistemas Distribuídos".

Aborda o emprego portos na comunicação e sincronização entre processos.

SANTOS (1987):

"Sistemas Distribuídos Aplicados à Automação Industrial".

Descreve a estrutura de um sistema de controle e os níveis de um sistema distribuído. Apresenta a implementação de um sistema operacional distribuído utilizando semáforos e troca de mensagens.

SCHWARZ (1981):

"Computadores em Tempo-Real: Uma Introdução aos Conceitos Básicos".

Apresenta uma introdução aos sistemas de controle de processos. Aborda questões de analise de sinais, aquisição de dados, confiabilidade e segurança de sistemas de computadores, software de tempo-real (sistema operacional e aplicativos) e fundamentos de sistemas de controle.

SEGRE (1987):

"Um Estudo de Ambientes de Programação Distribuída: Proposta de Extensões para MODULA-2".

Faz uma revisão em sistemas operacionais distribuídos, dando ênfase ao endereçamento de processos por portos. Discute configuração dinâmica de sistemas distribuídos e comenta uma implementação de um mecanismo de comunicação entre processos baseado em chamada remota de procedimentos.

SILBERSCHATZ (1981):

"Port Directed Communication".

Aborda o emprego portos na comunicação e sincronização entre processos.

SLOMAN, KRAMER & MAGEE (1986):

"The CONIC Toolkit for Building Distributed Systems".

Apresenta as linguagens de programação e de configuração do CONIC (que é um sistema distribuído para controle de processos em tempo-real). Explora a flexibilidade para reconfiguração dinâmica do sistema. Propõe um sistema de comunicação entre processos baseado em portos. Os módulos são gerados e compilados independentemente valendo-se da linguagem de programação. A linguagem de configuração é usada para ligar os portos entre si.

SOUZA (1987):

"Um Sistema Operacional de Rede".

Apresenta um sistema operacional de rede comercial que opera em conjunto com o sistema operacional nativo da estação.

TSICHRITZIS & BERNSTEIN (1974):

"Operating Systems".

Livro introdutório aos sistemas operacionais.

C.2. Textos Ligados a Tolerância a Falhas:

ANDERSON & KNIGHT (1981):

"Practical Software Fault Tolerance for Real-Time Systems".

Enfoca a questão de falhas de software como sendo "erros de projeto do software", impossíveis de se detectar "a priori". Propõe soluções baseadas em grafos de sincronização e na substituição do processo faltoso por outro funcionalmente equivalente, mas projetado independentemente.

ANDERSON & LEE (1981):

"Fault Tolerance, Principles and Practice".

Livro sobre técnicas de tolerância a falhas. Este livro é bastante utilizado como referência em diversos trabalhos nessa área.

ANDERSON & LEE (1982):

"Fault Tolerance Terminology Proposals".

Propõe uma série de termos a serem empregados na área de tolerância a falhas. (Ver também LAPRIE (1985)).

AVIŽIENIS & outros (1985):

"The UCLA DEDIX System: A Distributed Testbed for Multiple-Version Software".

Após discutir as técnicas de "forward" e "backward recovery", apresenta um sistema de

experimentação de mecanismos de tolerância a falhas que utiliza a técnica de N-versões. O algoritmo de decisão empregado é genérico, implementando (a) comparação bit-a-bit, (b) por cadeias de caracteres e (c) comparação de números inteiros e reais. O sistema é dividido em níveis hierárquicos e trabalha com decisões locais e globais.

BELLON & SAUCIER (1982):

"Protection against External Errors in a Dedicated System".

Estuda a contaminação de sistemas por erros externos a eles e propõe uma técnica de detecção e recuperação de falhas causadas pela contaminação.

CAMPBELL & RANDELL (1983):

"Error Recovery in Asynchronous Systems".

Propõe técnicas de "forward" e "backward error recovery" baseadas em ações atômicas e tratamento de exceções de software.

CAMPBELL, ANDERSON & RANDELL (1983):

"Practical Fault Tolerant Software for Asynchronous Systems".

Estuda o tratamento de exceções e "backward error recovery" através de operações atômicas e objetos recuperáveis.

COOPER (1986):

"Replicated Procedure Call".

Descreve uma implementação de chamada remota de procedimentos em uma arquitetura de módulos replicados. Na replicação é adotada a técnica de "N-versões".

CRISTIAN (1982):

"Exception Handling and Software Fault Tolerance".

Caracteriza mecanismos de exceção e mostra como podem ser empregados em mecanismos de tolerância a falhas.

CRISTIAN & outros (1985):

"Atomic Broadcast: from Simple Message Diffusion to Byzantine Agreement".

Apresenta três protocolos de difusão atômica de mensagens: o primeiro tolera falhas de omissão, o segundo falhas de tempo e o terceiro falhas gerais (ou falhas "bizantinas"). O protocolo se baseia no tempo absoluto indicado em cada estação levando em conta eventuais diferenças de sincronismo entre elas.

CRISTIAN (1987):

"Issues in the Design of Highly Available Computing Systems".

Discute o gerenciamento da disponibilidade de grupos de servidores. Faz uma abordagem resumida sobre o problema da manutenção da seqüência das mensagens enviadas para processos replicados. Esse tema é mais detalhado em CRISTIAN & outros (1985).

GUNNINGBERG (1983):

"Voting and Redundancy Management Implemented by Protocols in Distributed Systems".

Elabora uma proposição de um protocolo que implementa tolerância a falhas a partir de Redundância Modular Tripla ("Triple Modular Redundancy"). São discutidos protocolos de votação e recuperação de falhas.

KNIGHT, LEVESON & St. JEAN (1985):

"A Large Scale Experiment in N-Version Programming".

Este trabalho descreve uma experiência que conclui que as N-versões de um mesmo programa implementadas separadamente não são completamente independente. Essa constatação não inviabiliza, contudo, essa técnica. Ela apenas mostra que a confiabilidade de um sistema de N-versões não é tão alta quanto o valor teórico, igual à enésima potência da confiabilidade das versões isoladamente, pois esse valor pressupõe que haja total independência entre as versões.

KOKAWA & SHINGAI (1982):

"Failure Propagating Simulation and Nonfailure Paths Search in Network Systems".

Esse artigo propõe um método de simulação de propagação de erros em sistemas em rede. Através desse método é possível delimitar o alcance do dano causado ao sistema devido a essa propagação.

LAPRIE (1985):

"Dependable Computing and Fault Tolerance: Concepts and Terminology".

Propõe uma série de termos a serem empregados na área de tolerância a falhas. (Ver também ANDERSON, T. & LEE, P. A. (1982)).

LEITE & ARIBE (1988):

"Recuperação de Erros em Sistemas de Processos Concorrentes: uma Proposta".

Caracteriza falhas no software e problemas que existem para recuperá-los. Revê, ainda, métodos de recuperação de erros encontrados na literatura e propõe um novo modelo para tratamento de falhas de software.

LOMET (1977):

"Process Structuring, Syncronization, and Recovery Using Atomic Actions".

Apresenta o conceito de ação atômica e mecanismos empregados para sua implementação. Descreve, também, de uma forma simples e clara, a idéia e a estrutura de um bloco de recuperação.

LOQUES & KRAMER (1986):

"Flexible Fault Tolerance for Distributed Computer Systems".

Expõe uma técnica de tolerância a falhas em sistemas distribuídos implementada em CONIC. A

principal característica desta técnica é sua transparência em relação ao projeto dos módulos, isto é, cada módulo pode ser projetado ignorando-se a existência de mecanismos de deteção e recuperação de falhas. Só na configuração do sistema é que são especificados os pontos onde tais mecanismos devem atuar.

LOQUES (1988):

"Replicação Automática e Transparente de Módulos em Sistemas Distribuídos".

Descreve aspectos relativos à incorporação de mecanismos de tolerância a falhas de forma transparente à aplicação. Discute, ainda, a replicação ativa de elementos.

LOQUES, KRAMER & ANIDO (1988):

"Diverse and Selective Fault-Tolerance in a Distributed Environment".

Descreve a implementação em CONIC de diversos mecanismos de tolerância a falhas. Mecanismos como "hot-standby" ou "cold-standby" e replicação ativa e passiva de modulos são utilizados de forma seletiva, ou seja, apenas nos módulos em que são necessários.

MELLIAR-SMITH & RANDELL (1977):

"Software Reliability: The Role of Programmed Exception Handling".

Após apresentar algumas definições de conceitos, o artigo discute as causas de faltas e

erros tanto de hardware quanto de software. O texto propõe, ainda, o emprego de rotinas de tratamento de exceções em conjunto com esquemas viáveis de detecção e recuperação de erros de software.

POWELL & DESWARTE (1985):

"Candidate Solutions Proposed for OTV and DSP Missions".

Descreve as técnicas de tolerância a falhas propostas para dois projetos de veículos espaciais: o Veículo de Transferência Orbital (Ônibus Espacial) e a Sonda Espacial ("Deep Space Probe").

POWELL (1987):

"Fault-Tolerance in Delta-4".

Descreve as opções de projeto adotadas no DELTA-4 do programa ESPRIT, no que diz respeito a tolerância a falhas. Descreve mecanismos de replicação de módulos para situações (a) de falha seguida de interrupção da operação do módulo ("fail-stop") e (b) falha não controlada. São empregados mecanismos de votação por assinatura, ao invés de bit-a-bit, a fim de diminuir o "overhead" na comunicação.

RANDELL (1975):

"System Structure for Software Fault Tolerance".

Introduz conceitos, atualmente muito difundidos na área de tolerância a falhas, como: teste de aceitação, ponto de checagem ("checkpoint"), conversação e bloco de recuperação ("recovery block").

RANDELL, LEE & TRELEAVEN (1978):

"Reliability Issues in Computing System Design".

Este texto faz uma revisão bastante abrangente em vários tópicos de tolerância a falhas, entre eles: tipos de falhas, ações atômicas, níveis de abstração, detecção de erros (tipos e interfaces de checagem) e recuperação de erros ("backward error recovery", "foreward error recovery" e recuperação de erros em vários níveis). Descreve, ainda, alguns sistemas tolerantes a falhas.

SHRIVASTAVA (1979):

"Structuring Distributed Systems for Recoverability and Crash Resistance".

Discute a recuperabilidade de estados anteriores de um processo através da delimitação de regiões de recuperação. Propõe a hierarquia Mestre-Escravo para processos gerentes de recursos distribuídos, onde um mestre comanda diversos escravos, um em cada nó do sistema onde exista o recurso.

SHRIVASTAVA & PANZIERI (1981):

"The Design of a Reliable Remote Procedure Call Mechanism ".

Expõe a implementação de um mecanismo de chamada remota de procedimentos a partir de um sistema de troca de mensagens. Discute e sugere uma solução para o problema de chamadas órfãs, isto é, quando o processo chamador deixa de existir antes de receber a resposta do processo chamado.

WALTER, KIECKHAFER & FINN (1985):

"MAFT: A Multicomputer Architecture for Fault-Tolerance in Real-Time Control Systems".

Apresenta uma arquitetura distribuída para sistemas tolerantes a falhas. Essa arquitetura usa a técnica de votação, permitindo a reconfiguração do sistema em caso de detecção de falha. Vale notar que o artigo conta a origem do termo "concordância bizantina" (do inglês "Byzantine agreement"), que é amplamente usado na literatura.

C.3. Outras Referências:

HOROWITZ & SAHNI (1984):

"Fundamentos de Estruturas de Dados".

Texto introdutório ao tema "Estruturas de Dados".

IBM (1987):

"DOS 3.30 Technical Reference"

Manual.

KERNIGHAN & RITCHIE (1978):

"The C Programming Language".

Livro sobre a linguagem de programação "C". Kernighan e Ritchie são os autores da linguagem.

QUANTUM (1988):

"QNX - Reference Guide".

Manual.