| 1. Publicação nº | 2. Versão | 3. Data | 5. Distribuição |
|---|---|---|---|
| *INPE-4419-TDL/307* | | *Novembro 1987* | ☐ Interna ☒ Externa ☐ Restrita |

| 4. Origem | Programa |
|---|---|
| *PG/SDA* | *FRH/ECO* |

6. Palavras chaves - selecionadas pelo(s) autor(es)

*REDUNDÂNCIA DUPLA* *PONTOS DE RECUPERAÇÃO*
*DISPONIBILIDADE* *RECUPERAÇÃO RETROATIVA*

7. C.D.U.: *681.3.063*

8. Título *INPE-4419-TDL/307*

*UM SISTEMA DE SUPERVISÃO TOLERANTE A FALHAS*

10. Páginas: *157*

11. Última página: *C.7*

12. Revisada por

*Ricardo C. O. Martins*

9. Autoria *Maria de Fátima Mattiello Francisco*

Assinatura responsável

13. Autorizada por

*Marco Antonio Raupp*
*Dr. Diretor Geral*

14. Resumo/Notas

*Este trabalho propõe a incorporação de técnicas de tolerância a falhas a um sistema de controle de processos orientado para tempo real que se destina a atividades de supervisão. Estas técnicas visam prover maior disponibilidade ao Sistema de Supervisão se falhas simples do "hardware" ocorrerem. Uma arquitetura básica com redundância dupla é proposta. Em termos de "software" explora-se a técnica de recuperação de erro retroativa com o estabelecimento dos pontos de recuperação a cargo dos aplicativos de supervisão.*

15. Observações *Dissertação de Mestrado em Eletrônica e Telecomunicações, aprovada em junho de 1986.*

Aprovada pela Banca Examinadora
em cumprimento a requisito exigido
para a obtenção do Título de Mestre
em Eletrônica e Telecomunicações

Dr. Tatuō Nakanishi — Presidente

Dr. Ricardo Corrêa de O.Martins — Orientador

Dr. Eduardo Whitaker Bergamini — Co-Orientador

Dr. Orlando Gomes Loques Filho — Membro da Banca -convidado-

Engº Wilson Yamaguti, Mestre — Membro da Banca

Candidato: Maria de Fátima Mattiello

São José dos Campos, 30 de junho de 1986

AGRADECIMENTOS

Agradeço ao Dr. Ricardo C. Oliveira Martins a orientação dada ao presente trabalho, ao Dr. Eduardo W. Bergamini pela co-orientação, incentivo e interesse, e, em particular, ao Engº Antonio Francisco Júnior pela revisão técnica e apoio durante a execução deste trabalho.

*ABSTRACT*

*This paper proposes the incorporation of fault tolerant techniques in a process control oriented real time system to be applied in activities of supervision. These techniques should provide a higher availability of the Supervision System if simple fault of the hardware should occur. A basic architecture with dual redundancy is proposed. With regard to the software the backward-error-recovery technique is explored by establishing recovery points under the control of the supervision applicatives.*

SUMÁRIO

Pág.

## LISTA DE FIGURAS

## LISTA DE TABELAS

# CAPÍTULO 1

## INTRODUÇÃO

Neste trabalho o termo Sistema de Supervisão é caracterizado como um sistema de controle de processos que executa as tarefas de aquisição, processamento e atuação consideradas básicas neste tipo de aplicação. Para este fim, geralmente, considera-se um sistema de computação de uso geral ao qual são incorporadas: interfaces de aquisição de dados e de atuação, e uma interrupção periódica utilizada como referência de tempo para as aquisições de dados e outras tarefas periódicas executadas pelo sistema.

Dada a natureza da função a que se destina um Sistema de Supervisão, em geral ele deve possuir alta disponibilidade. Para atender este requisito ele necessita de recursos de segurança de modo a evitar colapso no controle dos processos devido a falhas no próprio sistema de computação.

Todas as técnicas empregadas na implementação de recursos de tolerância a falhas, adotadas neste trabalho, estão baseadas no uso de redundância. Entende-se aqui que recursos redundantes são elementos replicados do sistema que não seriam requeridos caso fosse garantido que o sistema estivesse livre de falhas.

Dado que a tecnologia atual permite que unidades de computadores sejam fabricadas com baixo custo, utilizando componentes integrados, torna-se economicamente atrativo e fisicamente possível o uso de unidades de computadores redundantes.

O objetivo deste trabalho é desenvolver recursos de tolerância a falhas que se incorporem ao Sistema Operacional padrão iRMX86, fechado a aplicação, utilizado para gerenciar e controlar as atividades de um Sistema de Supervisão. Diz-se que um sistema é fechado à aplicação quando ele não permite modificações do usuário. Considerando esta res

trição do Sistema Operacional iRMX86, serão investigados e projetados mecanismos de "hardware" e "software" para apoio à recuperação do Sistema de Supervisão em casos de falha simples de "hardware". O referido Sistema Operacional iRMX86 foi escolhido dado a sua característica modular e sua facilidade de expansão. Em adição, o iRMX86 é um Sistema Operacional voltado para o controle de processos em tempo real, fato que atende perfeitamente aos requisitos da aplicação de supervisão em vista.

Neste trabalho, a incorporação de recursos de tolerância a falhas ao Sistema Operacional iRMX86 consiste em considerar a sua expansão de forma a provê-lo de ferramentas para apoio à deteção de erros, juntamente com mecanismos para recuperação de erros com base no estabelecimento de pontos de recuperação.

No caso do Sistema Operacional iRMX86, a expansão pode ser feita de forma simples e eficiente, com a introdução de novos objetos (entidades gerenciadas por um Sistema Operacional, por exemplo: tarefas, caixas postais, semáforos, etc.) e chamadas ao sistema. Para tanto, é realizado um estudo dos conceitos utilizados na área de tolerância a falhas, mais especificamente das técnicas de recuperação de erro, efetivando-se a escolha daquelas que mais se adequam ao caso de Sistemas de Supervisão.

Pretende-se, desta forma, aumentar a disponibilidade do Sistema de Supervisão com a recuperação de falhas simples de "hardware", detectadas ao nível de circuitos ou mesmo ao nível dos processos aplicativos do sistema.

A tolerância a falhas em termos de recuperação de erros de projeto de "software" foge do escopo deste trabalho. Neste sentido, considera-se o fato de o sistema ser do tipo controle de processos e espera-se que o "software" de supervisão já tenha sido devidamente testado antes de entrar em operação.

A preocupação fundamental na incorporação das técnicas escolhidas para o Sistema Operacional iRMX86 é com a consistência dos pontos de recuperação, cuja análise é apresentada no decorrer deste trabalho.

Com relação ao conteúdo desta dissertação cabe citar que o Capítulo 2 introduz os conceitos e princípios básicos de tolerância a falhas com ênfase em deteção e recuperação de erros. Neste capítulo também são apresentados dois sistemas duais tolerantes a falhas, bastante conhecidos na literatura (TANDEM e Processadores ESS).

O Capítulo 3 preocupa-se em introduzir o ambiente de um Sistema de Supervisão tanto no que se refere à proposta de uma arquitetura básica e filosofia de operação, quanto na apresentação do Sistema Operacional, iRMX86, considerado como apoio à execução da aplicação.

No Capítulo 4 apresenta-se a técnica de recuperação de erro adotada, bem como novas camadas que deverão ser incorporadas à arquitetura do Sistema Operacional iRMX86 para torná-lo tolerante a falhas. Ainda neste capítulo são propostas duas ALTERNATIVAS para implementação dos mecanismos de apoio a recuperação do Sistema de Supervisão, um deles projetado basicamente por "software", enquanto o outro utiliza maior suporte de "hardware".

No Capítulo 5 os mecanismos propostos no capítulo anterior são analisados de forma comparativa, levando em consideração eficiência de processamento. Ainda neste capítulo é feita uma análise com a qual se garante que os mecanismos propostos apóiam uma recuperação consistente.

Finalmente, no Capítulo 6 são apresentadas as conclusões obtidas deste trabalho.

# CAPÍTULO 2

# TOLERÂNCIA A FALHA

Este capítulo cuida da apresentação dos conceitos básicos de tolerância a falha encontrados na literatura, salientando o emprego de redundância. Adicionalmente, procura-se dar ao leitor os princípios básicos de algumas técnicas de detecção e recuperação de erro que serão utilizadas no decorrer desta dissertação.

## 2.1 - CONCEITOS BÁSICOS

A terminologia associada à Tolerância a Falhas, adotada neste trabalho, aplica-se tanto a sistemas de "hardware" quanto de "software". Objetiva-se com isto a padronização dos termos básicos encontrados na literatura. Certas definições diferem um pouco daquelas adotadas em literatura mais antiga quando pouca importância era dada às inadequações de projeto.

De acordo com as definições de Randell et. al. (1978) um *sistema* é um conjunto de componentes e ligações entre eles, projetado para prover um ou mais serviços específicos. Os *componentes* de um sistema podem, eles próprios, ser, também, sistemas. A relação entre estes componentes do sistema é denominada *algoritmo do sistema*. Não existem restrições quanto ao fato de um componente vir a ser utilizado apenas por um único sistema. Ao contrário, ele pode ser um componente utilizado por vários sistemas, devendo existir um algoritmo específico que o relaciona a cada um dos sistemas.

A *confiabilidade* de um sistema é uma medida do sucesso do seu desempenho na realização de um procedimento específico, que implementa o algoritmo. Quando o procedimento do sistema é desviado daquele especificado, diz-se que ocorreu uma *falha* ("failure"). Assim uma falha é um evento e a confiabilidade do sistema é inversamente dada pela freqüência de tais eventos.

A *disponibilidade* é definida como a fração de tempo que um sistema satisfaz suas especificações quando ele é observável.

O serviço provido por um sistema é considerado como provido por um ou mais *ambientes*. Dentro de um sistema particular, o ambiente de um componente é caracterizado por ele e por todos os outros componentes com os quais ele se relaciona diretamente.

Diz-se que o estado de um sistema é um *estado errôneo* quando ele é tal que o processamento da própria especificação do sistema, pelos seus algoritmos, leva o sistema a uma falha não-conseqüente de um defeito posterior. O termo *erro* é usado para designar a parte do estado do sistema que está incorreta. Assim, um erro é um ítem de informação, e os termos erro, detecção de erro e recuperação de erro são usados como equivalentes casuais para estado errôneo, detecção de estado errôneo e recuperação de estado errôneo, respectivamente.

Um *defeito* ("fault") é a causa algorítmica (física ou abstrata) de um erro. Desta forma, a falha de um componente do sistema pode ser um defeito mecânico (elétrico ou eletrônico) do ponto de vista do sistema como um todo.

Um *erro* em um componente ou no projeto de um sistema será associado a um defeito do sistema. Assim sendo, um defeito no componente de um sistema reflete um erro no estado interno do componente, da mesma forma que um defeito no projeto de um sistema é um erro num estado interno do projeto (Anderson and Lee, 1981).

Um defeito pode ser classificado quanto à característica de sua ocorrência em três tipos:

transitório: é aquele que ocorre uma única vez, após o que deixa o sistema em uma condição livre de erro;

intermitente: é um defeito que se repete em intervalos de tempo;

permanente: é aquele que persiste constantemente, sem interrupção.

Não é muito simples atribuir a uma dada falha o defeito específico que a provocou. A ocorrência de uma falha durante o processamento indicaria a presença de um erro. Porém, tal falha poderia não permitir a identificação das informações errôneas a ela associadas, e a precisão do momento em que o estado tornou-se errôneo. Não raro, tal identificação é feita baseando-se em julgamentos subjetivos.

Considerações sobre os problemas de confiabilidade de sistemas de computação mais complexos sugerem a adoção de conceitos do tipo *ações atômicas*. Antes porém de introduzí-los é dada a noção de *estrutura dinâmica*.

A atividade de um sistema pode ser estruturada de diferentes formas, dependendo apenas dos aspectos dessa atividade que se tem interesse em salientar ou ignorar. Um conceito básico e bem estabelecido, usado para descrever alguns aspectos da estrutura dinâmica de um sistema, é o conceito de *processo*, também referenciado neste trabalho por *tarefa*. Processos são caracterizados como partes independentes de códigos executáveis que implementam certas atividades de um sistema. Diferentes processos, juntamente com suas interrelações, constituem a estrutura dinâmica do sistema.

Desta forma, a estrutura dinâmica de um sistema envolve aspectos de seqüenciamento (fluxo de controle) dos seus processos, além dos aspectos de comunicação (fluxo de informações) entre eles.

A Figura 2.1 apresenta os aspectos de *fluxo de controle* da estrutura dinâmica denominados criação, manutenção e encerramento de processos.

Os aspectos de *fluxo de informação* da estrutura dinâmica de um sistema cuidam da troca de informações entre processos do sistema.

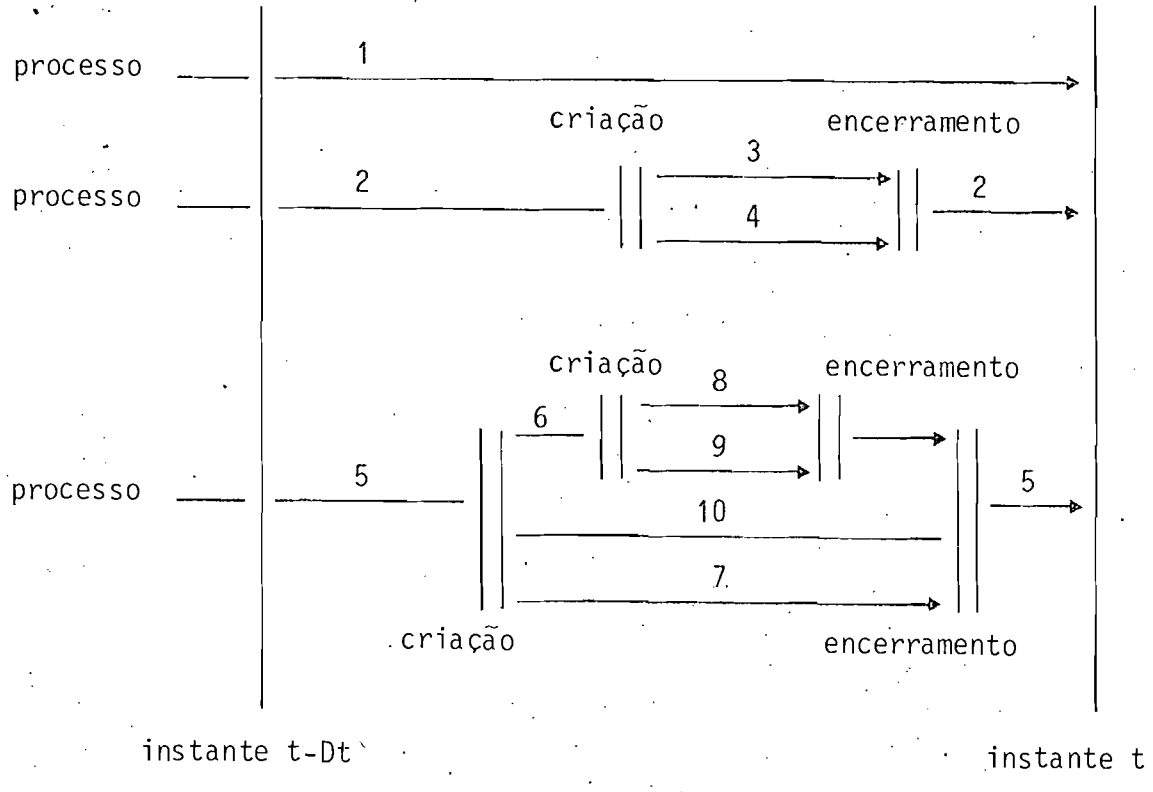


Fig. 2.1 - Aspectos de fluxo de controle de estrutura dinâmica.

⟶ processo em execução

Os problemas de fluxo de informação entre processos são, no mínimo, tão importantes quanto o fluxo de controle quando considerados em confiabilidade, particularmente, na determinação dos possíveis danos causados por um defeito (Randell et al., 1978).

Neste contexto, o conceito de ações atômicas permite expressar a estrutura dinâmica de um sistema em termos de fluxo de informações.

*Ações atômicas* caracterizam uma forma de correlacionar certas atividades do sistema, executadas pelos seus processos. São utilizadas pelo projetista na especificação daquelas interações entre processos (fluxo de informações) que devem ser evitadas, para que possa ser mantida a integridade do sistema. Uma melhor compreensão de ação atômica é obtida com a análise de suas propriedades.

As propriedades das ações atômicas são:

1 - não há fluxo de informação, em qualquer direção, entre o processo (ou grupo de processos) e o resto do sistema;

2 - uma ação atômica poderia envolver vários processos, desde que:

   a - um processo envolvido em uma única ação atômica pudesse temporariamente criar um ou mais processos;

   b - dois ou mais processos pudessem cooperar diretamente em uma ação compartilhada caso a atividade conjunta fosse atômica em relação ao restante dos processos do sistema.

A Figura 2.2 mostra estas possibilidades com base na Figura 2.1. As linhas ovais completas indicam ações atômicas, isto é, cada linha oval delimita os processos envolvidos em uma ação atômica. As linhas interrompidas (instante t) representam ações atômicas ainda em progresso.

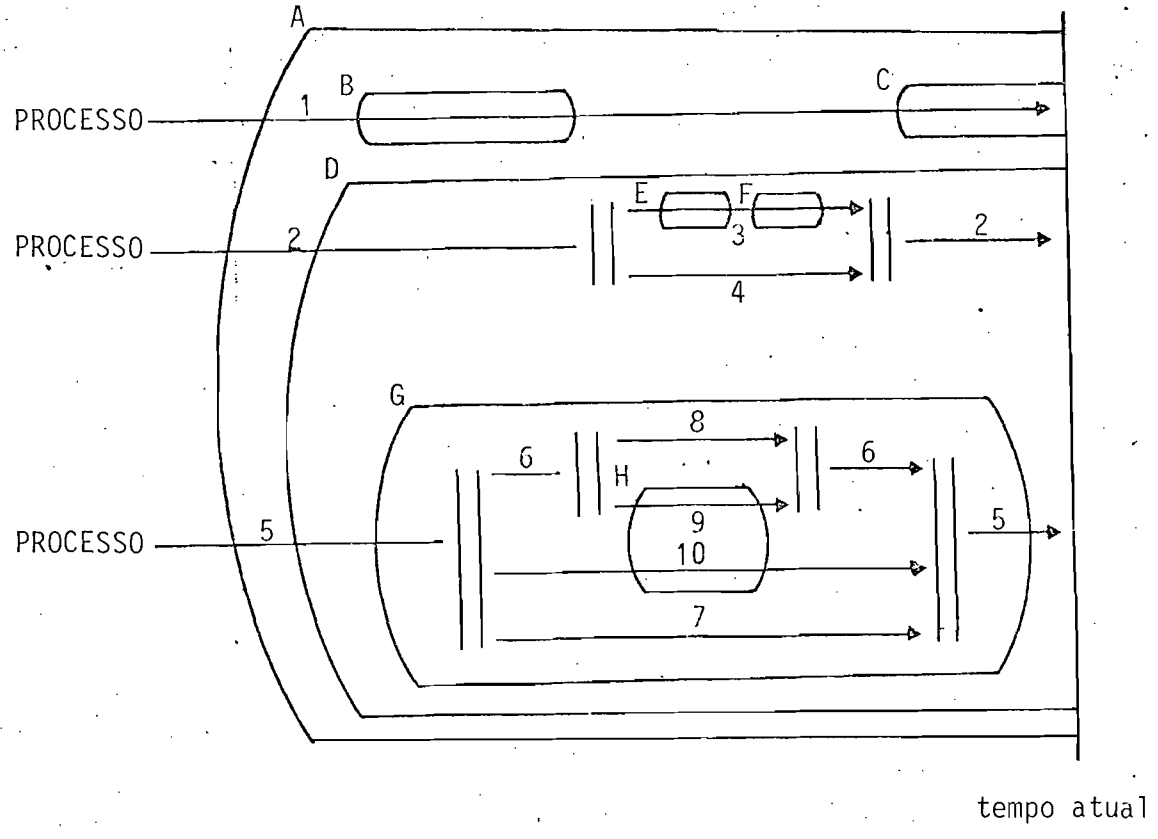


Fig. 2.2 - Processos e suas ações atômicas (A,B,C, ... H).

Apesar de as ações atômicas terem sido descritas como uma simples forma de o projetista indicar os pontos que caracterizam a integridade do sistema, elas são relevantes para uso nas técnicas que tratam de tolerância a falhas. Isto se deve ao fato de que as ações atômicas provêm uma forma de impedir a propagação de erros, e neste trabalho são utilizadas para assegurar uma boa recuperação do sistema caso haja ocorrência de falhas (vide Seção 2.6.2).

## 2.2 - PRINCÍPIOS DE TOLERÂNCIA A FALHAS

Para que sistemas tolerantes a falhas possam ser projetados e implementados de forma completa, evitando que a ocorrência de defeitos implique falhas de sistema, Anderson e Lee (1981) identificaram

quatro fases para a implementação de tolerância de falhas. São elas: (i) deteção de erros, (ii) análise e confinamento, (iii) recuperação de erro, e (iv) tratamento e continuação

(i) *Deteção de erro:* Para que um defeito no sistema possa ser tolerado, primeiramente, seus efeitos terão de ser detetados. Assim, enquanto um defeito não puder ser diretamente detetado pelo sistema, sua manifestação poderá gerar erros em outros lugares. Desta forma, o ponto inicial para aplicação das técnicas de tolerância a falhas é na deteção de um estado errôneo. Alguns mecanismos serão utilizados neste trabalho para deteção de erros. Eles serão apresentados a seguir, na Seção 2.5.

(ii) *Análise e Confinamento:* Quando um erro é detetado, nada pode garantir que o único estado errôneo do sistema seja aquele descoberto. Pode-se suspeitar de que outros estados do sistema também estejam errôneos. Devido a atrasos comuns entre a manifestação de um defeito e a deteção de suas conseqüências errôneas, informações inválidas podem ter sido geradas, levando a erros ainda não detetados. Assim, antes que qualquer cuidado seja tomado no sentido de detetar o erro, sugere-se que seja avaliada a extensão dos estados do sistema que possam ser danificados. Esta avaliação depende de decisões do projetista do sistema sobre como evitar a propagação do erro (confinamento) durante a execução, utilizando para isto medidas e mecanismos específicos. A avaliação também depende das técnicas utilizadas para detecção de erro (localização do defeito).

(iii) *Recuperação de erro:* Seguindo as fases anteriores, técnicas de recuperação devem ser utilizadas. Estas técnicas ajudam a transformar o estado atual errôneo num estado bem definido e livre de erro, para que o sistema possa continuar operando normalmente. Sem tais transformações, fatalmente o sistema prosseguiria operando com falha. Portanto, a recuperação de erro é um dos aspectos mais importantes de tolerância e falhas.

(iv) *Tratamento e Continuação:* Apesar de a fase anterior permitir que o sistema retorne a um estado livre de erro, podem ser aplicadas técnicas de modo a capacitar o sistema a dar continuidade ao serviço requerido por sua especificação, garantindo a não-recorrência imediata do erro que foi recuperado. Um problema no tratamento do erro é que a sua detecção não serve necessariamente para identificá-lo; a relação entre erros e defeitos pode ser complexa até mesmo em sistemas simples. Freqüentemente ocorre que diferentes defeitos geram o mesmo erro. Assim, o primeiro aspecto observado para o tratamento de falha consiste na localização do defeito efetuada na detecção. Somente depois é que se deve pensar em *reparar* o defeito ou mesmo *reconfigurar* o resto do sistema para evitá-lo. Na maioria dos sistemas é requerida a recuperação de erros, porém o reparo dos defeitos que causaram estes erros, apesar de muito desejável, não é necessariamente essencial à continuidade da operação.

Cabe notar que não é necessário que as quatro fases estejam implementadas em um sistema de computação tolerante a falhas. Esta decisão faz parte do projeto do sistema quando se podem avaliar os seus quesitos de tempo real, para poder ou decidir implementar mecanismos de tolerância a falhas.

É de interesse que as fases para tolerância a falhas, implementadas em sua totalidade ou parcialmente em um sistema tenham a sua execução invocada automaticamente de forma separada daquelas atividades que são normais ao sistema.

## 2.3 - SISTEMAS DUAIS TOLERANTES A FALHAS

Sistemas tolerantes a falhas devem ser intrinsecamente capazes de detectar falhas transitórias e permanentes de seus circuitos e programas. Uma das definições clássicas de sistemas de computação tolerantes a falhas estabelece que tais sistemas devem preservar a execu

ção correta de programas e de suas funções de entrada e saída, mesmo na presença de falhas, quaisquer que sejam as fontes de erro (Avizienis, 1976). Assim, pela definição de Siewiorek e Swarz (1982), *computação tolerante a falhas* implica a execução correta de algoritmos específicos, mesmo na presença de defeitos, de modo que as conseqüências dos defeitos em um sistema podem ser superadas pelo uso de *redundância*. Esta redundância pode ser tanto temporal (por meio de execuções repetidas ), quanto física ("hardware" ou "software" replicado). Cabe notar que neste trabalho serão consideradas apenas as redundâncias físicas duais tanto de "hardware" quanto de "software".

Como em todo projeto de sistemas, as suas especificações delimitam o espaço de atuação do projeto e, conseqüentemente, as técnicas que podem ser usadas. No mesmo nível de especificação, sistemas tolerantes a falhas são categorizados pela alta disponibilidade ou pela alta confiabilidade.

Equipamentos com alta confiabilidade, tolerante a falha, são extremamente importantes em aplicações que envolvem risco de vida humana como em centrais elétricas nucleares, sistemas para monitorar pacientes em hospitais, sistemas de transporte; em situações onde o equipamento não pode ser reparado, como equipamento que opera em localidades remotas e sistemas aeroespaciais (computadores de bordo); ou mesmo aqueles sistemas cujo reparo é proibitivamente caro. Também, em algumas aplicações industriais, a falha do equipamento eletrônico em processos de controle de produção pode ter graves conseqüências econômicas ou mesmo catastróficas, em questões de segurança.

No caso de sistemas que exigem alta disponibilidade deseja-se que seus equipamentos estejam disponíveis para executarem suas funções de computação o maior tempo possível, quando em serviço. Atividades do tipo manutenção preventiva e reparo reduzem o tempo de disponibilidade do sistema. De acordo com Sieworek e Swarz (1982), *disponibilidade* é tipicamente usada como uma figura de mérito em sistemas cujos serviços podem ser atrasados ou suspensos por períodos curtos de tempo, sem conseqüências sérias.

A título de exemplo, serão apresentados a seguir dois sistemas de computação que se caracterizam por permitirem alta disponibilidade de serviço. São eles: TANDEM e PROCESSADORES ESS.

### 2.3.1 - TANDEM

O TANDEM 16 (Sieworek and Swarz, 1982) foi o primeiro sistema comercialmente disponível, com expansão modular, projetado especificamente para alta disponibilidade. Os objetivos de projeto deste sistema incluem:

- Operação "sem parada" mesmo na presença de falhas. Neste caso o sistema é reconfigurado colocando os componentes com defeito fora de serviço. Componentes reparados são configurados de volta no sistema, sem que ocorra interrupção no uso dos outros componentes.

- Nenhuma falha simples de "hardware" deve comprometer a integridade dos dados do sistema.

- A expansão modular é obtida com a adição de capacidade de processamento, memória e periféricos, sem que o "software" aplicativo seja modificado.

O TANDEM é composto de até 16 computadores interconectados por 2 barramentos de 16-bits paralelos cada (DYNABUS). A arquitetura de cada processador TANDEM, apresentada na Figura 2.3 (Serlin, 1984), possui memória própria e canais de entrada/saída (e/s). Todos os controladores dispõem de portas de e/s duais ("dual port") para acesso alternativo por 2 canais de e/s, tipicamente de 2 processadores distintos, no caso de defeito de um dos processadores ou da unidade de e/s.

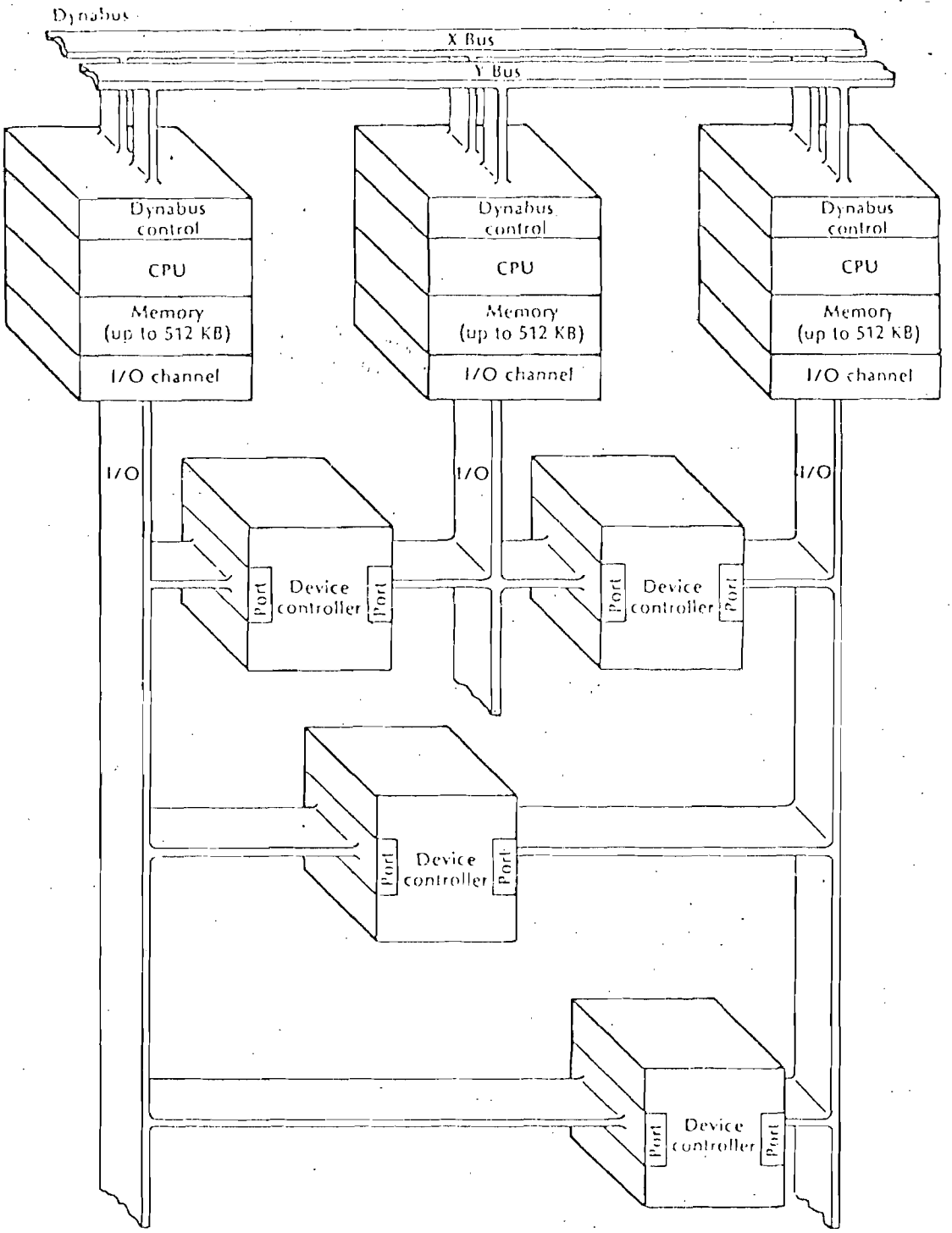


Fig. 2.3 - Arquitetura do TANDEM
FONTE: Serlin (1984).

Os acionadores de disco ("disk drives") são também do tipo com portas duais, sendo cada acionador conectado a 2 controladores. Assim, os dados do disco permanecem acessíveis mesmo na decorrência de falhas de um processador e um controlador ao mesmo tempo.

Caso, opcionalmente, duas cópias idênticas (espelhamento) do disco sejam consideradas, até mesmo com um defeito em um dos acionadores, a base de dados poderia ainda ser recuperada. Com espelhamento de disco, o sistema automático mantém 2 cópias idênticas dos arquivos especificados, em duas unidades independentes.

Cada processador é energizado individualmente e assim pode ser conectado para reparo independentemente dos outros componentes do sistema.

Controladores consomem energia de ambos processadores aos quais eles se conectam; a capacidade de cada alimentador é tal que o controlador pode continuar funcionando mesmo sem alimentação de um dos processadores.

Como mecanismos para tolerância a falhas, o "hardware" ainda inclui:

- "checksums" nas mensagens do DYNABUS;
- paridade nos caminhos de dados;
- código de correção de erro na memória;
- temporizador do tipo cão de guarda ("watch dog timer").

Cada processador no sistema contém uma cópia do sistema operacional GUARDIÃO, que mantém tabelas locais que refletem o estado de todos os recursos disponíveis no sistema. Desvios repetitivos ("loops" infinitos) e caída do processador são detetados pela ausência de mensa

gens do tipo EU ESTOU BEM, emitidas periodicamente por um processador e divulgadas para os outros processadores pelo DYNABUS.

O "software" construído sobre esta estrutura de "hardware" é um sistema orientado por *processo* com toda comunicação manipulada por *mensagens*. Esta abstração permite a transparência dos limites físicos existentes entre processadores e periféricos. Qualquer e/s ou recurso do sistema pode ser acessado por um processo, sem nenhum obstáculo quanto ao local onde o recurso ou processo possa residir.

A integridade dos dados é mantida pelo mecanismo *par de processos* de entrada/saída: um processo de e/s é designado primário, enquanto o outro reserva. Todas as mensagens de manipulação de arquivos que se encontram naquela unidade de e/s são liberadas para o processo primário de e/s. Então o primário envia uma mensagem com informações de ponto de teste ("checkpoint") para o reserva, para que este possa acessá-la no caso de defeito do processador primário ou mesmo defeito no caminho de acesso a e/s.

Arquivos também poderão ser duplicados fisicamente em dispositivos distintos controlados por um par de processos de e/s, em diferentes processadores. Todas mensagens de modificação do arquivo são liberadas para os dois processos de e/s. Na ocorrência de uma falha física ou no isolamento do processo primário, o arquivo reserva é atualizado e tornado disponível.

Não só os processos que cuidam de e/s podem utilizar o mecanismo par de processos. Programas de aplicação SEM PARADA ("non stop"), feitos pelos próprios usuários, podem empregar tal mecanismo. Maiores informações sobre o sistema operacional e programas de aplicação SEM PARADA poderão ser encontradas em Bartlett (1977).

Este sistema TANDEM 16 foi utilizado por aplicações do tipo: registro hospitalar, transações bancárias e transações bibliotecárias.

## 2.3.2 - PROCESSADORES ESS

Os Sistemas de Chaveamento Eletrônico (ESS), desenvolvidos pelo "Bell Laboratories" desde 1953, são utilizados por grande número de sistemas digitais tolerantes a falhas. Desde aquela época já foram desenvolvidos os sistemas ESS nº 1, ESS Nº 2, ESS nº 3, ESS nº 4, utilizados extensivamente pelas companhias de operação da "Bell System" para prover o serviço telefônico comercial. Estes sistemas servem a todos os tipos de serviços telefônicos:

- ESS nº 1, de grande capacidade, serve para serviços metropolitanos;

- ESS nº 2, de capacidade média, foi projetado para serviços suburbanos;

- ESS nº 3, muito utilizado em pequenos serviços rurais;

- ESS nº 4, para tráfego telefônico de longa distância, codificado digitalmente.

Os sistemas ESS manipulam o roteamento de chamadas telefônicas através de centrais de serviços. Estes sistemas visam alta disponibilidade: duas horas de parada em 40 anos (3 minutos por ano) (Sieworek and Swarz, 1982).

Os objetivos de projeto destes sistemas permitem que não mais que 0,01% das chamadas telefônicas sejam processadas incorretamente (Toy, 1978).

No centro de cada ESS encontra-se um processador simples de alta velocidade. Para estabelecer um ambiente de chaveamento confiável, os componentes do sistema são redundados, incluindo a duplicação do próprio processador. Sem esta redundância, a falha de um único componente no processador poderia causar uma falha completa do sistema. Nes

te caso, com a duplicação, o processador reserva assume o controle e continua o serviço telefônico enquanto o outro é reparado. Caso ocorra uma outra falha durante o tempo de reparo de um dos processadores, então naturalmente o sistema deixa de funcionar. Esta técnica tem sido utilizada por todos os sistemas ESS (Toy, 1978).

No caso do sistema ESS nº 3, por exemplo, ele também opera com circuitos redundantes; porém, diferentemente dos sistemas ESS nº 1 e ESS nº 2, não realiza comparação entre os resultados obtidos pelas duas unidades redundantes. Neste sistema a deteção de erro é provida por um circuito de autoverificação incorporado ao processador. Dois processadores são utilizados, em operação normal, o processador ativo ("on line") é responsável pela execução e processamento das chamadas telefônicas; enquanto o de reserva permanece no estado PARADO. A operação de escrita atua sobre as memórias dos dois processadores para manter a memória da unidade reserva atualizada. No caso da operação de leitura, o acesso é permitido apenas para a memória da unidade ativa, exceto quando ocorre um erro de paridade na leitura. Como o ESS 3A utiliza técnicas de microprogramação em seu processador central, a ocorrência do erro de paridade na leitura resulta em uma interrupção na execução do microprograma, que passa a ler a palavra de memória da unidade de reserva, de modo a deixar o erro transparente ao meio externo.

O ESS nº 3 possui 20 canais principais de e/s, cada um com 20 subcanais, permitindo ao processador o controle e acesso a até 400 unidades periféricas.

Os processadores possuem 16 registros de uso geral, e os dados nestes registros, bem como na memória principal, são codificados utilizando 2 bits de paridade. A verificação é feita a cada transferência de dados de uma posição para outra no processador.

A unidade central de controle é verificada periodicamente por dois temporizadores, que funcionam como CÃO DE GUARDA. Desta maneira, se um tempo específico foi excedido ("time-out"), um erro é detetado e o sistema inicia o procedimento de recuperação.

Neste sistema a recuperação pode ser automática ou manual. A *recuperação automática* tem início assim que o erro é reconhecido pelo processador ativo. Os sinais de erro são armazenados no registro de erro (RE) para que possa ser feita uma diagnose posterior. Além disso, os sinais de erro são divididos em 3 grupos, cada um causando a execução de um conjunto de ações diferentes por parte do sistema.

O primeiro grupo de erro está associado aos dispositivos de e/s. Este tipo de erro causa uma interrupção pela qual o processador exerce um controle completo na maneira de como determinar a causa exata do problema. Se o erro for uma falha transitória, a informação sobre este erro é armazenada para uma análise posterior. Caso o erro se deva a uma falha de "hardware" de um dos componentes do processador, então o programa que serve a interrupção inicia uma reconfiguração, chaveando para a unidade reserva através do canal especial de manutenção.

O segundo tipo de erro envolve falhas que ocorrem na unidade de reserva do sistema. Estas falhas influenciam diretamente na operação da unidade ativa. Por exemplo, o procedimento de escrita na memória é feito tanto na ativa quanto na reserva e ambas enviam sinal de confirmação para o processador completar a operação. Se a resposta for gerada apenas pela memória ativa, um certo período de tempo é esperado para a chegada da resposta da memória reserva. Se este sinal não chegar dentro deste período, um circuito especial de TEMPO ESGOTADO gera uma interrupção (sinal de erro) e o processador ativo continua sua operação normal, mantendo isolada a unidade com defeito.

O terceiro tipo de erro envolve falha de "hardware" do processador ativo. Para ilustrar este tipo de erro basta retomar o exemplo do caso anterior: acesso às memórias do sistema. Se o sinal de confirmação de escrita é recebido pela unidade reserva e não pela ativa, um sinal de erro é gerado para forçar o chaveamento do sistema para a unidade reserva. Então o sistema sai do ar momentaneamente e um sinal de reinicialização ("restart") na unidade reserva inicializa seu processador, que continua o atendimento das chamadas telefônicas, afetando,

talvez, apenas a chamada que estava sendo processada durante a transição de estado.

A *recuperação manual* é necessária quando o sistema não é capaz de recuperar-se por procedimento automático.

### 2.3.3 - CONSIDERAÇÕES SOBRE REDUNDÂNCIA

Como pode ser visto nos sistemas de alta disponibilidade apresentados, o ponto fundamental em que se baseia a maior parte das técnicas utilizadas para tolerância a falha é a redundância (nos casos citados redundância dupla). Assim, o mecanismo básico utilizado para obtenção da maior disponibilidade dos sistemas foi a duplicação de componentes.

De acordo com Anderson e Lee (1981), todas as técnicas de tolerância a falha dependem do emprego efetivo e utilização de redundância, isto é, de elementos extras no sistemas que são redundantes no sentido em que se tornam dispensáveis ao sistema caso este, garantidamente, estivesse livre de falhas.

Antigamente, os métodos utilizados para a introdução de redundância num sistema de computação eram classificados de acordo com sistemas que proviam tolerância somente diante da ocorrência de defeitos de componentes de "hardware", o que naturalmente separava a redundância de "hardware" da redundância de "software". A redundância de "software" era, então, provida exclusivamente pelos programas que efetuavam a integração da redundância de "hardware" no sistema tolerante a falha. Considerava-se a não-existência de falhas de projeto, por exemplo, os sistemas de "software" eram considerados livres de falhas.

Redundância de "hardware" tem sido categorizada como do tipo estático (mascaramento) ou dinâmico. Na *redundância estática* os componentes redundantes são usados para prover tolerância a falhas no sentido de mascarar os efeitos da falha de um componente dentro do sis

tema. Em contraste, *redundância dinâmica* é empregada justamente para atender a capacidade de deteção de erro no sistema. Por exemplo, o uso de código de correção de erro em unidades de memória pode ser uma aplicação de redundância estática, enquanto um código de paridade simples é uma aplicação de redundância dinâmica, visto que a tolerância de uma falha de memória requer, por exemplo, redundância no código do sistema operacional.

A classificação de redundância como estática ou dinâmica é absolutamente dependente da estrutura do sistema e não simplesmente do uso da redundância. Diferentes visões abstratas de um sistema tolerante a falhas podem gerar diferentes classificações de redundância dentro daquele sistema. Por exemplo, um teste de paridade acoplado ao sistema operacional que mantém cópias reservas de páginas de memória poderia ser visto pelo usuário como um tipo de redundância estática.

A classificação citada é bastante apropriada para redundância de "hardware". Isto se deve à tendência dos limitantes físicos em ditar a visão adotada pelos sistemas de "hardware", e portanto suas redundâncias. Em sistemas de "software", onde a estruturação tende a ser mais abstrata, esta classificação não é comumente adotada, embora possa ser igualmente aplicada.

Uma das consequências da incorporação de redundância num sistema é que seu tamanho e complexidade são aumentados, implicando fatores que podem levar a uma redução na confiabilidade do sistema, o que seria uma contradição. Já que redundância é necessária para tolerância a falhas, sua incorporação, e uso dentro do sistema, deve ser cuidadosamente estruturada e controlada para minimizar qualquer aumento desnecessário na complexidade. Uma solução para isto consiste em separar aqueles componentes do sistema que realmente necessitam de tolerância a falhas dos restantes. Tal separação capacita a identificação de alguns componentes críticos para as atividades de tolerância a falhas.

## 2.4 - SISTEMAS MULTINÍVEIS

Por serem os Sistemas de Computação digital bastante complexos, uma forma conveniente de torná-los mais compreensíveis consiste em estruturá-los em níveis. A evolução de um nível conceitual primitivo para um nível conceitual mais alto é feita através de uma série de abstrações. Cada abstração contém somente informações importantes para aquele nível e dispensa informações desnecessárias referentes aos níveis inferiores.

De acordo com Randell et al. (1978), quando se olha um sistema ou suas atividades, realizadas por um de seus conjuntos de componentes, e se concentra na relação entre eles, está-se deliberadamente fazendo uma abstração do sistema como um todo.

Em termos de estruturação, pode-se dizer que o sistema foi estruturado num único nível de abstração, ou *estruturação horizontal*. Ao considerar maiores detalhes do sistema (ou outras partes menores), observa-se um nível de abstração inferior, que mostra como os componentes e suas inter-relações são implementadas. Outros detalhes de componentes e suas inter-relações podem justificar novos níveis de abstração.

Para a identificação de um conjunto de níveis de abstração e para a definição da relação entre eles torna-se desejável uma estruturação do sistema, neste caso referenciada por *estruturação vertical*. Esta estruturação descreve como os componentes são construídos ao contrário da estruturação horizontal que descreve como os componentes se interagem. A importância dos níveis de abstração deve refletir a facilidade com que se pode entender e controlar partes específicas, sem necessidade de considerar todos os detalhes. Neste caso, torna-se indispensável uma especificação bem documentada das características externas de cada nível. Tais especificações podem ser concebidas como *interfaces interpretativas* (ou interfaces de abstração) quando especificam a relação vertical entre os níveis e como *interfaces de comunicação* quan

do especificam a relação entre componentes de um mesmo nível. Em ambos os casos, a interface, quando bem escolhida, permite que o projetista ignore o trabalho das outras partes do sistema, fora da interface.

O exemplo da Figura 2.4 (Randell et al., 1978) apresenta um sistema completo interpretativo multinível. Ele tem uma estrutura vertical constituída por 4 níveis, implementados cada um por um *interpretador*. Cada interpretador é programado usando um conjunto de facilidades aparentemente atômicas (indivisíveis) providas pela interface de abstração, e tem a função de oferecer um conjunto abstrato de facilidades, definido pela interface de abstração do nível imediatamente superior. Devido à completa especificação e documentação das interfaces de abstração, o projetista da implementação de qualquer nível normalmente precisará de pouco ou quase nenhum conhecimento do projeto como um todo, ou mesmo da existência de outros níveis.

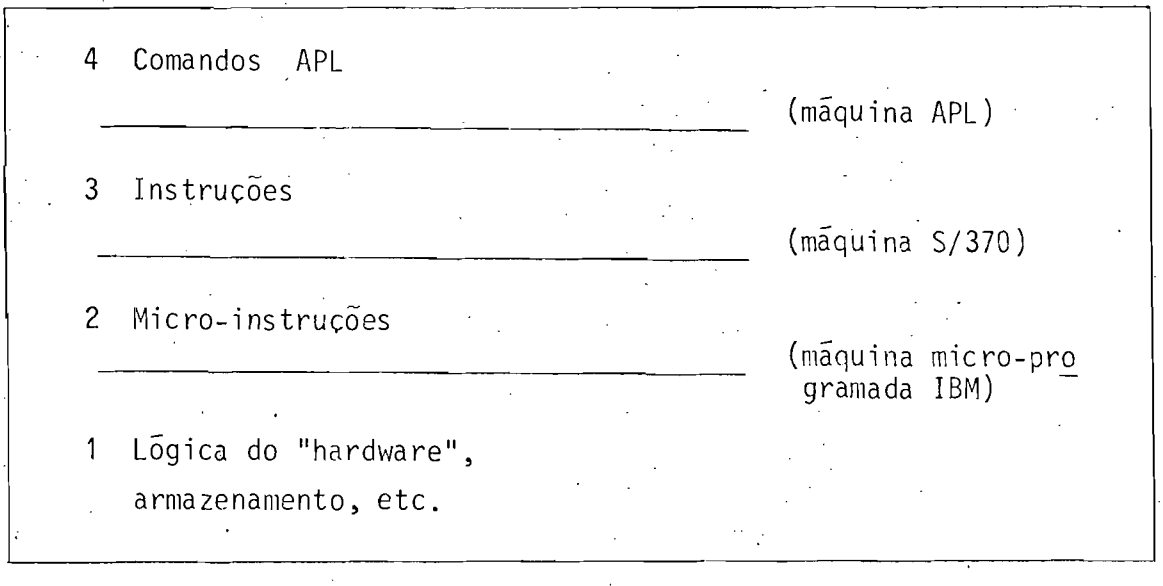


Fig. 2.4 - Sistema interpretativo multiníveis.

O objetivo do enquadramento de um sistema de computação na estrutura vertical é a tentativa de previnir falhas (ou mais comumente, certos tipos de falhas) decorrentes da invalidação da abstração que um nível deveria prover. A maior parte da estruturação vertical de sistemas visa a provisão de base para implementação de tolerância a falha.

### 2.4.1 - MANIPULAÇÃO DE EXCEÇÕES

Além da estruturação do sistema de computação em níveis hierárquicos, outros fatores que auxiliam muito na implementação de tolerância a falha é o bom enquadramento das quatro fases para tolerância a falha (veja Seção 2.2) e a separação das atividades anormais do sistema das atividades normais, sendo as anormais automaticamente invocadas quando requisitadas (Anderson and Lee, 1981). Entende-se por atividades anormais aquelas que cuidam dos erros para tolerância a falhas.

As atividades anormais (*exceções*) estão associadas a respostas em situações onde os componentes não desempenham suas funções corretamente, isto é, respostas anormais são normalmente sinalizadas quando ocorre uma falha no sistema. Esta sinalização é apropriada para a inicialização do processo de tolerância a falha, ou seja, para a *manipulação da exceção*.

Exceções e facilidades para manipulação de exceções formam a base da implementação de tolerância a falha. Uma visão mais prática para sistemas de computação consiste em considerar uma *interface interpretativa* entre o "hardware" e o "software" de um sistema. Com isto tem-se um melhor entendimento do uso de exceções e dos mecanismos de manipulação destas exceções. E mais, uma vez que, frequentemente, recursos de tolerância a falhas em "hardware" são implementados por técnicas de "software", a manipulação de exceções em componentes de "hardware", sinalizadas por uma interface interpretativa, pode ser de vital importância para a tolerância a falha num sistema (Anderson and Lee, 1981).

Exemplos típicos de exceções sinalizadas por uma interface interpretativa:

- divisão por zero;

- estouro aritmético;

- instrução ilegal;

- violação de proteção.

Medidas podem ser tomadas para que uma exceção possa ser encaminhada corretamente para o seu manipulador. Desta forma, a sinalização de uma exceção resultará na chamada ao manipulador. Há vários métodos pelos quais isto poderia ser feito. A mecanização de um deles seria: a interface interpretativa sinalizar uma variável ("set flag") interrogada por um programa. As desvantagens desta mecanização são:

- não existe maneira de forçar o programa a interrogar a variável e invocar o manipulador;

- não é eficiente ter de testar as exceções.

O mais usual e desejável é que a interface interpretativa inicialize um *mecanismo de exceção* que, por sua vez, invoca automaticamente um manipulador. Desta forma, é forçada uma mudança do fluxo de controle do programa. Exemplo, um mecanismo de interrupção é comumente usado para este fim em muitos computadores.

Idealmente, uma interface interpretativa sinaliza exceções de falha para indicar a presença de falhas no componente e sinaliza exceções de interface no caso de violação da interface de abstração (indicação de falha no projeto do sistema). Além disso, uma interface interpretativa pode prover um *mecanismo de exceção* que executa instruções que permitem:

(i) que novas exceções possam ser declaradas;

(ii) que exceções sejam sinalizadas ou assinaladas com parâmetros, de modo a permitir a transferência de informações para um manipulador;

(iii) que manipuladores de exceções possam ser habilitados ou desabilitados.

A implementação destes recursos pela interface interpretativa é relativamente estratégica. Embora a classificação de exceções sinalizadas pela interface interpretativa, como exceções de interface ou exceções de falha, seja interessante, pode não ser simples para a interface interpretativa distinguí-las. A forma usada para fazer esta distinção é caracterizada pelo *nome da exceção*. Não se pode garantir que a ocorrência de um defeito no sistema causará a sinalização de uma exceção por um de seus componentes. Portanto torna-se necessária a inclusão de medidas de deteção de erro no sistema. Testes de diagnóstico de um componente poderão ser usados para revelar defeitos de componentes, enquanto que testes do estado do sistema poderão detetar erros causados por defeitos de projeto.

Com referência a um sistema interpretativo simples é muito desejável que a utilização de um mecanismo que habilita a deteção da falha invoque um manipulador de exceção específico por ocasião da deteção de estado errôneo. Por exemplo, o interpretador provê uma instrução do tipo:

SINALIZE (nome exceção, outros parâmetros).

Manipulação de exceção é um conceito muito aceito em sistemas de "software" (não apenas para uso em tolerância a falhas). Poucos sistemas de "software" estruturados num único nível consideram a utilização de manipulação de exceções. Ela é mais adequada em Sistemas Multiníveis.

### 2.4.2 - MANIPULAÇÃO DE EXCEÇÕES EM SISTEMAS MULTINÍVEIS

Neste sistemas é permitido que a exceção seja sinalizada para o manipulador de exceção de um determinado nível diferente daquele onde o erro possa ter ocorrido. Desta forma, torna-se necessária a utilização de uma interface interpretativa capaz de executar a sinalização entre níveis (Anderson and Lee, 1981).

A existência de múltiplos níveis de abstração pode levar à propagação de exceções de um nível para outro. Por exemplo, suponha-se que um programa P utilize uma função F do nível inferior (exemplo: acesso a disco) e que durante a execução desta função a interface interpretativa sinalize uma exceção. Se a manipulação desta exceção já estivesse prevista pela própria função F, então nenhuma dificuldade ocorreria. Entretanto, se nada tivesse sido previsto, a alternativa seria a interface interpretativa pesquisar o contexto de manipulação de P para descobrir qual seria o manipulador apropriado. Esta alternativa não é muito utilizada já que a sinalização de uma exceção deve ter mais significado para a função onde a falha ocorreu do que para o programa invocador da função. Assim, uma regra útil a observar consiste em prover que exceções sinalizadas não se propaguem automaticamente. Um fato que tem sentido ocorrer é incorporar ao manipulador de exceções de F a sinalização de uma exceção "diferente" para P; esta tarefa deve ser de responsabilidade do manipulador de F e não da interface interpretativa.

## 2.5 - DETEÇÃO DE ERRO

O ponto fundamental de toda estratégia para tolerância a falha é a deteção de um estado errôneo. O sucesso de um sistema tolerante a falha consiste na efetividade das técnicas de deteção de erro escolhidas.

Na prática existem limitações quanto à extensão das detecções, providas por um sistema, uma vez que ela está intimamente vinculada às redundâncias existentes, o que significa aumento de custo e sobre

carga de processamento decorrente dos testes extensivos utilizados no processo de deteção (Anderson and Lee, 1981).

Na realidade, discute-se deteção de erro em termos de:

*medidas* - procedimentos incorporados ao sistema para realização de testes de seus componentes;

*mecanismos* - ferramentas implementadas em uma interface interpretativa para apoiar a detecção de erro durante a execução de programas.

O objetivo de tais medidas e mecanismos é a sinalização de exceções aos seus manipuladores que, então, executam a invocação automática das outras fases para tolerância a falha (vide Seção 2.2).

A seguir apresentam-se algumas medidas e mecanismos considerados de relevância para o presente trabalho.

As *medidas* consideradas são as seguintes:

(1) Testes de Tempo - É uma medida bastante comum, que pode ser aplicada tanto em sistemas de "hardware" quanto de "software". Se a especificação de um componente do sistema inclui restrições de tempo para prover seu serviço, então deve-se aplicar um teste de tempo para verificar se elas foram cumpridas ou não. Caso as restrições não tenham sido cumpridas, então o teste de tempo pode sinalizar uma *exceção de tempo esgotado* ("time-out") indicando uma provável falha do componente. Anderson e Lee (1981) dizem que estes testes são muito usados para indicar a presença de falhas, mas não sua ausência, e se mostram úteis no apoio a outros testes de um sistema. Quando estes testes são usados em sistemas de "software", baseados em um temporizador do tipo CÃO DE GUARDA ("watch dog timer"), o requisito imposto é que periodicamente o "software" deve reiniciali

zar ("reset") o CÃO DE GUARDA indicando que ele está operando satisfatoriamente. Caso esta atualização não ocorra devidamente (exemplo: desvio repetitivo ("loop" infinito), então o CÃO DE GUARDA esgota seu tempo de espera e sinaliza uma exceção ao sistema. O CÃO DE GUARDA é um mecanismo da interface interpretativa. No caso do processador ESS nº 1 (Seção 2.3.2), este tipo de teste é usado para garantir que o controle geral do sistema não será perdido em decorrência de um mau funcionamento do "hardware" ou do "software". Um teste similar, com um CÃO DE GUARDA implementado por "software", é utilizado para evitar que recursos do sistema sejam perdidos ou fiquem inutilizados por programas com falha. Um "bit" de indicação de tempo esgotado é acionado ("set") na estrutura de dados associada a um determinado recurso, por um *programa auditor*, quando tal recurso está em uso. Esta sinalização é desfeita ("reset") quando o recurso estiver disponível. O programa auditor pode então verificar se o recurso foi utilizado por um tempo maior que o máximo especificado. Uma técnica similar é empregada em programas de aplicação para detetar armazenamentos ("buffers") perdidos. O teste de tempo é a medida de deteção de erro utilizada em maior escala pelo sistema TANDEM 16 (Seção 2.3.1).

(2) Teste de Consistência ("Reasonableness") - Os testes de consistência cuidam da verificação da consistência dos estados de vários objetos (abstratos) do sistema. Isto é feito baseado na utilidade destes objetos, como previstos pelo projetista do sistema. Um teste comum para aceitação é o *teste de limites* que pode determinar se o valor de um objeto particular é consistente com os limites estipulados. Muitos destes testes de consistência aplicados em sistemas de "software" são semelhantes a teste de TIPO ("type"), normalmente associados a programas escritos em linguagens de alto nível que apoiam tipos de dados abstratos. Enquanto muita falha de "software" pode ser detetadas durante a compilação dos programas, testes feitos durante a execução destes programas também são essenciais. Por

exemplo, testar se um valor gerado está dentro dos limites do "array" é possível somente durante a execução do programa. Portanto, não se pode afirmar que os testes executados durante a compilação de um programa sejam suficientes para garantir que não ocorrerão por ocasião da sua execução. É verdade também que pode ocorrer falha no interpretador ou no compilador do programa. Muitos tipos de testes feitos durante a execução de programas têm de ser realizados por medidas no programa, ainda que possam ter sido gerados automaticamente pelo compilador. Neste caso, apesar de os testes serem aplicáveis durante a execução, eles são utilizados somente durante a fase de testes dos programas, após o que eles são omitidos. Isto se dá quando o sistema é colocado em operação normal. O fato é que tais medidas causam sobrecarga intolerável no tempo de execução, razão pela qual elas são descartadas em operação normal. Já que a supressão destes testes é feita em detrimento do emprego efetivo da tolerância a falha no sistema, seria desejável que a própria arquitetura da máquina provesse mecanismos que evitassem a necessidade da incorporação de tais medidas (ineficientes) ao sistema. Testes explícitos de consistência incluídos em sistemas de "software" são denominados ASSERÇÕES (ou "assert statements"). Uma ASSERÇÃO é uma expressão lógica que implementa um teste de consistência em objetos de um programa, que é realizado em tempo de execução. Diz-se que a expressão lógica é verdadeira ("true") se o estado não é errôneo e sinaliza uma exceção caso contrário (Anderson and Lee, 1981). Por exemplo, quando um programa está sendo executado por uma interface implementada por "hardware", vários testes construídos no "hardware" podem ser complementados por asserções em "software". Estas asserções podem estar embutidos nos próprios processos ou no núcleo do sistema. Um teste executado em "hardware" pode, por exemplo, revelar um erro pela deteção de algum uso inválido da interface, por exemplo, devido a uma divisão por zero, violação de proteção ou tentativa de executar um código de operação inválida. Uma asserção de "software" pode revelar um erro

pela deteção de uso ilegal de dados do programa; por exemplo, teste de vetores, testes de limites de matrizes ou testes de relação inválida entre variáveis. Assim, quando um erro é detetado, quer por "hardware" ou "software", uma interrupção é sinalizada e o controle do programa passa para um manipulador de exceções do sistema (Anderson and Knight, 1983).

Quanto aos mecanismos de deteção de erro, cabe lembrar que são implementados com o uso de uma interface interpretativa para realização efetiva das medidas apresentadas anteriormente. Os mecanismos de deteção são os seguintes:

(1) *Recurso para Teste de Tempo* - Consiste basicamente em um temporizador CÃO DE GUARDA (ou "watch dog timer") implementado por "hardware" ou "software" para permitir a realização de testes de tempo;

(2) *Recurso para Teste de Consistência* - É caracterizado por um conjunto de objetos abstratos disponíveis com suas operações associadas. Já que todas as interações são feitas em termos destes objetos e operações, a interface interpretativa pode prover um *teste de interface* que consiste em testar se a operação requerida é legítima, se os operandos são válidos e compatíveis com a operação. Para Anderson e Lee (1981), o Teste de Interface é uma forma muito atraente de detecção de erro, já que a deteção é executada automaticamente, em paralelo com a execução da operação requisitada - não podendo ser suspensa por um programador. Infelizmente, as interfaces interpretativas de propósito geral prevêem apenas objetos fixos vinculados ao "hardware" (exemplo: estouro aritmético, violação de proteção) ao invés de darem apoio também à deteção de erros em programas. Idealmente seria desejável que o sistema operacional possuisse extensões para a declaração de novos objetos com limitantes específicos (definição de asserções) para a deteção de erros. Adicionalmente, seria desejável que facilidades para extensão

das interfaces interpretativas fossem oferecidas pelo sistema operacional para permitir que testes de interfaces (automáticos) pudessem validar o uso destes novos objetos e sinalizar as exceções apropriadas.

## 2.6 - RECUPERAÇÃO DE ERRO

A seção anterior cuidou da apresentação, num certo grau de detalhe, da fase de deteção de erro, caracterizada como recurso para obter tolerância a falhas. Esta fase é considerada passiva no sentido que não se destina a efetuar nenhuma mudança no sistema. Em contraste, a fase de *recuperação de erro* é ativa, pois deve mudar o sistema para que falhas ocorridas e suas consequências sejam toleradas ou compensadas. A recuperação de erro é a ferramenta utilizada para eliminar os erros de estado do sistema, geralmente pelo uso de informações redundantes mantidas por ele. Por isso tem recebido bastante atenção dos pesquisadores e projetistas de sistemas.

Quando um erro é detetado num sistema, uma exceção é sinalizada invocando o manipulador da exceção. Este, naturalmente, implementa alguns métodos para eliminar os erros de estado do sistema.

Os métodos para tratamento de erros já detetados são usualmente classificados em: técnicas de recuperação preditiva ("forward recovery") e técnicas de recuperação retroativa ("backward recovery"). Tais técnicas se destinam a colocar o sistema num estado livre de erro, de modo que seu processamento possa continuar como se nada tivesse ocorrido.

A *recuperação retroativa* envolve o retorno de um ou mais processos de um sistema para um estado anterior que se espera estar livre de erro, de onde ele deve continuar as próximas operações do sistema. Em contraste, a *recuperação preditiva* baseia-se na manipulação de parte do estado atual do sistema para produzir um novo estado que, então, deverá ser livre de erro (Randell et. al., 1978).

A *recuperação retroativa* depende da provisão de *pontos de recuperação*, isto é, uma maneira pela qual um estado de execução (ou situação) de um processo num dado instante pode ser registrado e futuramente restaurado ("restoration"). Isto possibilita a recuperação de erros não-previstos, podendo portanto a recuperação retroativa ser implementada por um mecanismo em qualquer nível de abstração do sistema.

A *recuperação preditiva*, no entanto, é projetada como parte integrante do sistema que ela serve, sendo portanto realizada por medidas embutidas nos próprios processos aplicativos. Por este fato, esta técnica caracteriza-se por sua dependência da aplicação.

A Tabela 2.1 mostra um quadro comparativo das características das duas técnicas apresentadas.

Os mecanismos de recuperação retroativa de erro são caracterizados mais facilmente em termos da estratégia que eles adotam para registrar e preservar dados nos pontos de recuperação. Duas estratégias, com características substanciais, estão disponíveis: ponto de teste ("checkpoint") e trilha de auditoria ("audit trail").

Um *ponto de teste* ("checkpoint") é conhecido como a forma mais direta de garantir que o estado de um processo possa ser restaurado. Esta estratégia consiste simplesmente na obtenção de uma cópia completa do estado dos processos quando um ponto de recuperação é estabelecido. Assim, quando uma restauração ("restoration") de estado for necessária, o estado errôneo é simplesmente descartado e substituído pelo estado armazenado no último ponto de recuperação anterior a ocorrência do erro.

TABELA 2.1

CARACTERÍSTICAS DAS TÉCNICAS DE RECUPERAÇÃO DE ERRO

| RECUPERAÇÃO RETROATIVA | RECUPERAÇÃO PREDITIVA |
|---|---|
| independência de avaliação dos danos provocados. | dependente dos danos e da predição. |
| capaz de fornecer recuperação para defeitos não-previstos. | só é capaz de recuperar erros previstos. |
| conceito geral aplicável a todos os sistemas. | projetada especificamente para um sistema particular. |
| facilmente implementada como mecanismo. | impossível de ser implementada como mecanismo. |

A segunda estratégia, *trilha de auditoria* ("audit trail"), não registra nenhum dado de recuperação quando um ponto de recuperação é estabelecido. Ao invés, a atividade do processo, subsequente ao estabelecimento do ponto de recuperação, passa a ser cuidadosamente monitorada e um registro suficientemente detalhado daquela atividade é preservado num histórico para possibilitar as devidas mudanças no estado do processo quando uma restauração daquele ponto de recuperação for necessária. Assim, a restauração de um ponto de recuperação, neste caso, é realizada pelo processamento inverso dos eventos registrados no histórico, através de sucessivas mudanças no estado do sistema, como se desfazendo os efeitos de cada evento.

De forma geral, para um ponto de recuperação recente, o uso de trilha de auditoria em oposição a ponto de teste reduz a sobrecarga associada à recuperação dos dados registrados e o tempo necessário para a restauração ("restoration"), porém, obviamente ambas crescerão com o aumento da distância (em tempo) deste ponto de recuperação (Anderson and Lee, 1981).

A conhecida estratégia de dispensa para recuperação ("recovery cache") é considerada neste trabalho como um caso especial de ponto de teste ("checkpoint") já que se caracteriza por uma cópia não-completa do estado dos processos do sistema. É copiada apenas a parte do estado que foi modificada (Anderson and Lee, 1981).

As técnicas até aqui apresentadas se preocuparam com os dados armazenados nos pontos de recuperação. Porém, nada foi dito a respeito de como estabelecer e descartar os pontos de recuperação.

É possível construir um mecanismo de recuperação de erro retroativo que opere independentemente dos processos do usuário. Neste caso, o estabelecimento e o descarte de pontos de recuperação deveriam estar sob o controle da interface interpretativa. Por exemplo, a interface interpretativa poderia, automaticamente, estabelecer pontos de recuperação em intervalos de tempo regulares. Uma política deveria ser formulada de modo a capacitar a *interface interpretativa* para a determinação de quais pontos de recuperação que deveriam ser descartados, objetivando sempre a manutenção da capacidade de recuperar. Apesar de um esquema de recuperação deste tipo, completamente automático, ser vantajoso no sentido de transparência aos processos do usuário, ele tem de ser padrão, não podendo prover recuperação adaptada à estrutura destes processos.

Uma forma de estabelecer pontos de recuperação adaptáveis à estrutura da aplicação consiste na utilização de *ciclos de controle*. De acordo com Martins e De Paula (1983), um ciclo de controle é o tempo necessário para que seja completado um conjunto básico de tarefas do sistema, sendo definido como um múltiplo inteiro do ciclo básico de controle do sistema (base de tempo do sistema) que é um intervalo de tempo com início e fim bem determinados, nos quais os estados do sistema e os processos ativos também estão bem determinados. Em geral, esta técnica é mais adequada a ambientes de sistemas de computação para controle de processos. Estes sistemas são caracterizados pelo seu comportamento bem determinado, cujos processos são definidos como cíclicos. Diz-se que um

*processo é cíclico* se existe um período de tempo (T) para o qual a sequência de chamadas a primitivas do sistema operacional se repete. Em sistemas dessa natureza, uma interface interpretativa pode se encarregar de estabelecer e descartar os pontos de recuperação de acordo com os ciclos de controle do sistema.

Uma terceira forma de estabelecer pontos de recuperação pode ser implementada através dos aplicativos do sistema. Neste caso, duas situações são consideradas, embora com diferença sutil: na primeira delas o programador dos processos da aplicação é provido de um conjunto de primitivas do sistema operacional que incorporam em seu código mecanismos de recuperação de erro. Assim, quando utilizadas pelo programador permitem, implicitamente, o estabelecimento de pontos de recuperação sem que o usuário tenha de lidar diretamente com os mecanismos de recuperação (Rossi and Simone, 1984). Na segunda situação, no entanto, a interface interpretativa provê o programador da aplicação com primitivas específicas para recuperação de erro que dão a ele completa liberdade sobre o estabelecimento de pontos de recuperação. Assim, estas primitivas são utilizadas pelo programador de acordo com a sua visão da aplicação. Apesar de o bom emprego desta técnica estar muito sujeito à experiência do programador, tem-se a vantagem de poder incorporá-la a códigos de programas antigos, codificados quando a aplicação não dispunha ainda de nenhum recurso de tolerância a falhas (Loques, 1984a)

### 2.6.1 - RECUPERAÇÃO EM PROCESSOS CONCORRENTES

Quando se trata de um sistema com concorrência de processos, devem ser tomados cuidados quando um destes processos sinaliza uma exceção. É necessário considerar que a provisão de tolerância a falha para aquele processo pode afetar outros processos no sistema. Como na seção anterior, será dada maior atenção à técnica de recuperação retroativa de erro devido a sua generalidade e mecanização.

Para melhor apresentar e aplicar as técnicas de recuperação de erro em processos concorrentes, estes foram classificados em três categorias: (i) independentes, (ii) competidores e (iii) cooperantes. Diz-se que os processos concorrentes são *independentes* se o conjunto de objetos abstratos acessados por cada processo forem disjuntos, considerando que a atividade de cada processo deve ser privada completamente dos outros processos. Neste caso, a recuperação é independente para cada processo e os mecanismos adotados podem ser os mesmos que implementam recuperação em processos sequenciais.

Processos concorrentes são denominados *competidores* se existirem objetos no sistema, acessados por mais de um processo, com o objetivo de melhor utilização destes objetos (recursos escassos), porém com a garantia de que nenhum fluxo de informação entre processos (Seção 2.1) resultará deste compartilhamento. Como a comunicação entre os processos competidores foi totalmente excluída pela própria definição, poder-se-ia adotar uma técnica de recuperação relativamente simples ao se considerar, apropriadamente, a abstração de que os processos competidores sejam independentes. Porém, na prática, existe a necessidade de mecanismos mais sofisticados que aqueles utilizados em recuperação de erro em processos sequenciais. Estes mecanismos são conseguidos com a estrutura de sistemas multiníveis, através da comunicação entre processos individuais e a extensão da interface interpretativa (requisições para aquisição e liberação de objetos partilhados). Este recurso é implementado por métodos convencionais de controle de alocação de recursos, como é executado, na prática, por muitos sistemas operacionais, em termos de monitores.

A terceira e última categoria consiste nos processos *cooperantes*, os quais tem acesso compatilhado e objetos que são usados diretamente para comunicação entre processos, não sendo imposta nenhuma restrição quanto ao fluxo de informação. Infelizmente, o fluxo de informação entre processos implica a dependência dos processos. Portanto a recuperação de erro em processos cooperantes exige muito cuidado, pois refazer o processamento entre o ponto de recuperação e o ponto onde foi

detetado o erro implica refazer todas as comunicações no intervalo citado para garantia de recuperação dos processos que se comunicaram entre si.

### 2.6.2 - EFEITO DOMINÓ E RECUPERAÇÃO CONSISTENTE

Em particular, a tentativa individual de processos cooperantes na realização de recuperação de erro retroativa pode resultar em problemas conhecidos como *efeito dominó*. Resumidamente, o efeito dominó se deve ao estabelecimento de pontos de recuperação de forma desordenada, o que provoca uma programação descontrolada de restaurações ("restoration") na recuperação de erro. Este efeito é ilustrado na Figura 2.5 (Anderson and Lee, 1981).

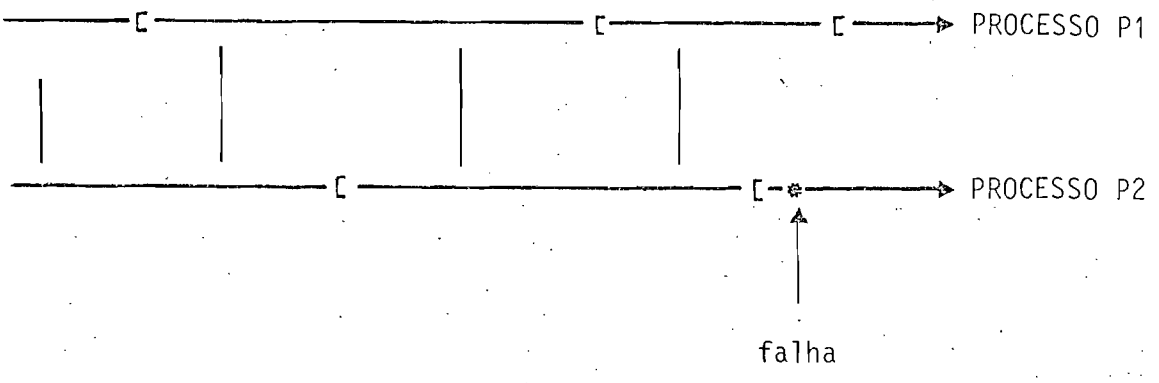


Fig. 2.5 - Efeito dominó.

Os processos $P_1$ e $P_2$, independentemente, estabelecem pontos de recuperação; cada '[' marca um ponto de recuperação ativo no qual o estado do processo correspondente pode ser restaurado ("restored"). As linhas verticais indicam a ocorrência de comunicação entre os processos.

Nenhuma dificuldade é encontrada se uma exceção é sinalizada pelo processo $P_1$, visto que a restauração de seu ponto de recuperação mais recente prove recuperação consistente. Porém se o processo $P_2$ sinaliza uma exceção, a recuperação de erro retroativa torna-se mais difícil, pois seu ponto de recuperação mais recente predece a uma comunicação com $P_1$. Para efeito de recuperação consistente não somente $P_2$ deve ser restaurado para um estado anterior, mas também $P_1$. O mesmo fato ocorre então com $P_1$ e assim, sucessivamente, um retorno descontrolado é desencadeado, com pontos de recuperação sendo restaurados. Desta forma uma falha num único processo pode resultar em recuperação em vários, senão em todos os processos cooperantes, anteriores.

Para evitar o efeito dominó em processos cooperantes e garantir uma recuperação consistente, restrições devem ser feitas na concorrência, na comunicação ou na capacidade de recuperação.

Russel (1980) apresenta três métodos para evitar o efeito dominó, que são apresentados a seguir.

O primeiro deles consiste em abandonar a recuperação estritamente retroativa que se faz pelo uso de recuperação do tipo PONTO DE TESTE ("checkpoint"). Isto é, deve-se evitar que certas interações sejam refeitas, passando a utilizar informações sobre as características do sistema (compensações). Com isto, na obtenção de um estado válido, algumas interações poderiam dispensar a propagação de restaurações ("restoration"), evitando assim, o efeito dominó.

O segundo método cuida de coordenar as ações de uma recuperação retroativa limitando as interações entre processos numa *conversão*. Este método utiliza o conceito de ações atômicas apresentado na Seção 2.1. Randell (1975) diz que um conjunto de processos executados num sistema de computação estão engajados em uma conversação quando se comunicam entre si e quando não interagem com processo algum que esteja fora do conjunto. Cada processo estabelece um ponto de recuperação quando entra na conversação, o que pode ser feito em instantes de tempo dis

tintos. Todos os processos, no entanto, devem deixar a conversação no mesmo instante, pois se um erro for detetado em um dos processos do conjunto, todos os processos deste conjunto deverão retornar ao início da conversação. Como mostra a Figura 2.6 três processos estão engajados em uma conversação. A linha tracejada delimita a ação atômica decorrente da conversação. As linhas pontilhadas indicam que os processos não estão sendo executados. As linhas contínuas em negrito indicam que os processos estão sendo executados, e os símbolos '<' e '>' respectivamente indicam o início e o término da execução destes processos.

O terceiro método para evitar o efeito dominó considera que todas as interações entre processos são *dirigidas*, isto é, que os dados são enviados unidirecionalmente *de* um processo fonte *para* um outro processo destino específico numa comunicação *síncrona do tipo PEDIDO-RESPOSTA*. Este método é utilizado por Loques (1984a) em sistemas distribuídos.

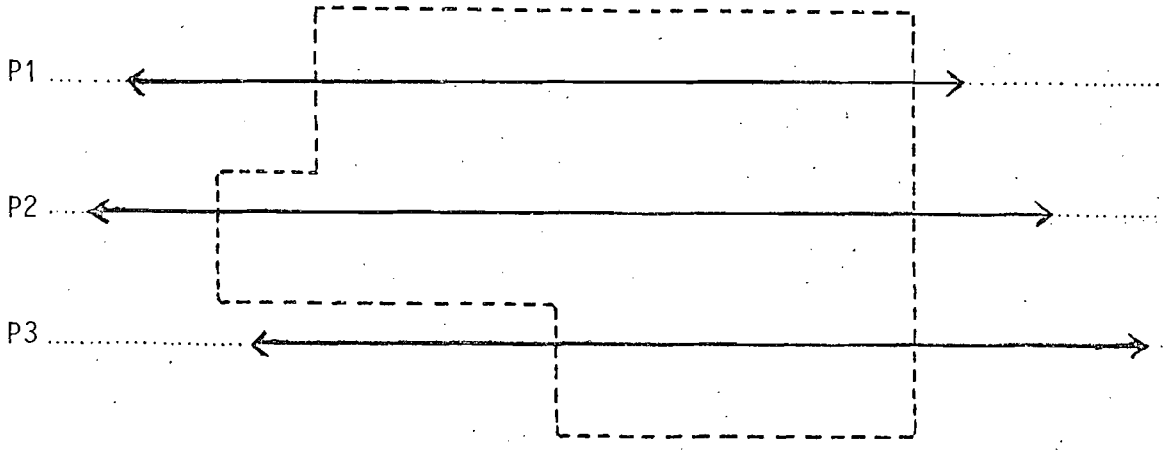


Fig. 2.6 - Conversação.

A garantia de *recuperação consistente* em um sistema de computação com processos concorrentes pode ser obtida pela observação da definição precisa de *consistência* formulada por Anderson e Lee (1981):

seja o subconjunto de processos $P_1,\ldots,P_n$ e suponha que pontos de recuperação tenham sido estabelecidos nos instantes $t_1,\ldots,t_n$ respectivamente. Este conjunto de pontos de recuperação é *consistente* num instante 't' posterior se as seguintes condições para os processos $P_i$ e $P_j$ do subconjunto são satisfeitas:

(i) no período $t_i$ a $t_j$ os processos $P_i$ e $P_j$ não se comunicam;

(ii) no período $t_i$ a 't' o processo $P_i$ não se comunica com nenhum outro processo que não estiver no subconjunto.

Assim, para um conjunto de pontos de recuperação ser consistente, cada processo $P_i$, depois de estabelecer seu ponto de recuperação só pode comunicar-se com aqueles processos do conjunto que já estabeleceram seus pontos de recuperação.

# CAPÍTULO 3

## SISTEMA DE SUPERVISÃO

Este capítulo cuida da caracterização de um Sistema de Supervisão, especifica uma Arquitetura Básica para o referido Sistema, a qual é provida de redundância dupla, e apresenta de forma breve o Sistema Operacional iRMX86 adotado, detalhando apenas alguns aspectos do NÚCLEO deste sistema, considerados relevantes para um completo entendimento do presente trabalho.

### 3.1 - CARACTERÍSTICAS FUNDAMENTAIS

Neste contexto, os Sistemas de Supervisão apresentam a estrutura típica de um sistema de controle de processos por computador, na qual são identificadas as três tarefas básicas da aplicação: aquisição, processamento e atuação.

A tarefa de *aquisição* tem a função de coletar, através de uma interface de aquisição, os valores de um conjunto de variáveis, os quais deverão ser analisados e controlados. Esta análise é feita pela tarefa de *processamento*, responsável pela verificação da condição de obediência dos valores das variáveis às restrições a elas impostas. Qualquer anomalia detetada desencadeará a execução de rotinas de regulamentação que se incumbem de *atuar* no sistema controlado para que a variação detetada possa ser corrigida e o sistema continue operando devidamente.

Quando as informações obtidas da interface de aquisição são simplesmente apresentadas ao operador, sem que ele atue diretamente no processo, o sistema é classificado apenas como um *sistema de supervisão sem controle*.

Quando o operador, a partir de informações obtidas através da supervisão, emite comandos para atuar diretamente sobre o pro

cesso, o sistema é classificado como um *sistema de supervisão com controle manual*.

Naturalmente, o sistema é classificado como de *supervisão com controle automático*, quando a atuação é feita diretamente no processo, através de algoritmos específicos, sem a intervenção direta do operador.

As funções de aquisição, processamento e atuação correspondem à parte do sistema de controle de processos que implementa uma aplicação específica. De modo geral, esta parte aplicativa é implantada em um ambiente de computação mais amplo que consta essencialmente de "hardware" que contém processadores, memórias de massa e outros periféricos, e de um "software" que consiste em um sistema operacional de tempo real e de bibliotecas de programas de uso geral, voltadas para controle de processos.

Usualmente, o "hardware" de um Sistema de Computação para Supervisão é composto de circuitos de uso geral, aos quais são incorporadas as interfaces de aquisição de dados e de atuação. Como nestes sistemas, normalmente, é importante o conceito de tempo absoluto, um *relógio de tempo real* torna-se parte essencial do equipamento, atuando como referência de tempo para as aquisições de dados e para outras tarefas periódicas executadas pelo sistema.

É importante salientar que um Sistema de Computação para Supervisão com Controle Automático deve apresentar alta disponibilidade. Para tanto, deve possuir medidas de segurança de modo a evitar colapso no controle dos processos devido a falhas no próprio sistema de computação. Recentemente, o uso de mais de um processador para esta finalidade, em operação simultânea ou cooperativa, passou a ser frequente. Desta forma, a alta disponibilidade destes sistemas é obtida com a redundância de "hardware", adicionada aos mecanismos de "software" para tolerância a falhas, incorporados ao sistema operacional e utilizados pelos programas aplicativos de supervisão.

## 3.2 - ARQUITETURA BÁSICA

O "hardware" proposto neste trabalho para um Sistema de Supervisão procura atender fins gerais de supervisão. Portanto nenhuma ênfase será dada a circuitos específicos de aquisição e atuação.

Os objetivos deste Sistema de Supervisão, além da modularidade e simplicidade, incluem:

- Operação sem interrupção, mesmo na presença de falhas. Para isto ele deve possuir capacidade de reconfiguração de modo a colocar os componentes com defeito fora de operação até que eles sejam reparados.

- Nenhuma falha simples de "hardware" deve comprometer a integridade dos dados do sistema.

- Na ocorrência de uma falha, não se deseja que o sistema seja reinicializado a ponto de ser desligado; o que sugere a incorporação de técnicas automáticas de recuperação de erro no sistema.

Para atender aos requisitos de tolerância à falha do projeto, propõe-se uma arquitetura básica semelhante à dos sistemas duais apresentados na Seção 2.3, no que se refere à utilização de *redundância dupla*. Como pode ser visto na Figura 3.1, a arquitetura básica considerada é composta de duas Unidades de Processamento (referidas simplesmente por UNIDADES) interligadas entre si por dois barramentos paralelos idênticos (A e A') de alta velocidade (com Acesso Direto a Memória) que possuem linhas de dados e linhas de controle.

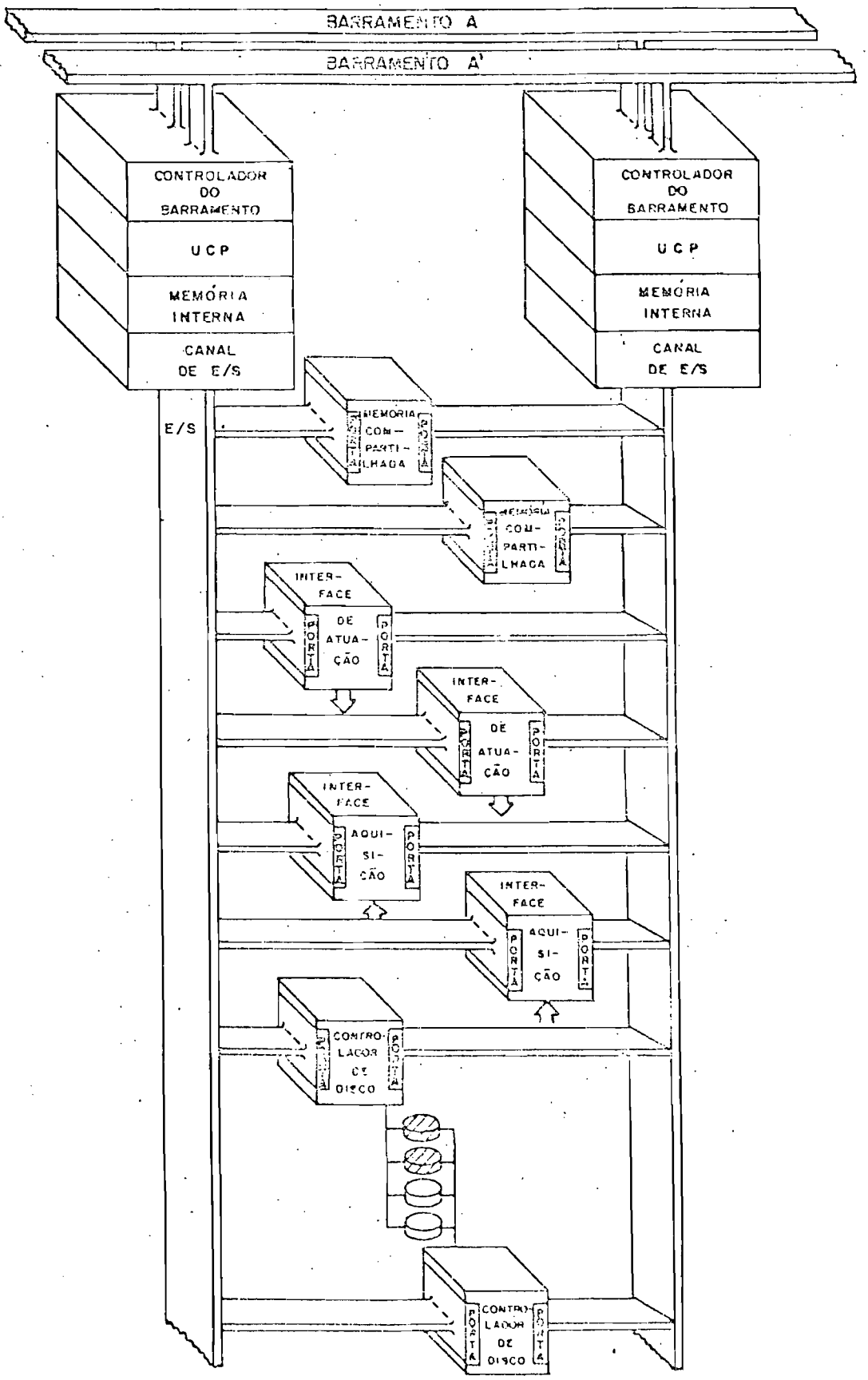


Fig. 3.1 - Arquitetura básica do sistema de supervisão.

Cada UNIDADE possui:

- um processador (INTEL 86,88 - 16 bits);
- um controlador do barramento;
- uma memória interna de leitura/escrita (RAM);
- um canal de entrada/saída (E/S).

O canal de E/S da UNIDADE é utilizado como via de acesso a recursos compartilhados do tipo portas duais. Um destes recursos, propostos como parte de arquitetura básica, consiste em uma *memória compartilhada*. Esta memória é fisicamente constituída por dois módulos de memórias de leitura/escrita (MEMRAM) idênticas às memórias internas. Para tolerar falha simples da memória compartilhada esta é duplicada.

O canal de E/S também é utilizado para acessar as *interfaces de aquisição e atuação* cujos controladores também devem ser do tipo portas duais e duplicados.

Analogamente ao Sistema TANDEM 16 (Seção 2.3.1) é possível que sejam configurados, nesta arquitetura básica, controladores de disco e terminais de vídeo do tipo *portas duais*. No caso do disco, seus acionadores também devem ser do tipo portas duais; cada acionador conectado a dois controladores. Isto permite que os dados do disco permaneçam acessíveis mesmo na ocorrência de falha simultânea de um processador e um controlador.

Também deve ser considerada a existência de espelhamento automático (Seção 2.3.1) do disco, isto é, cada par de discos deve possuir exatamente as mesmas informações de modo que qualquer operação de escrita é executada simultaneamente nos dois discos.

Supõe-se que o "hardware" ainda inclua, como mecanismos para tolerância a falhas:

- temporizador do tipo CÃO DE GUARDA (Seção 2.5);
- paridade na transferência de dados.

Cabe citar que, dependendo da complexidade da aplicação de supervisão que se deseja implementar, uma configuração mais simples deste sistema pode ser utilizada, consistindo na Arquitetura Básica proposta sem os controladores de disco. Em outros casos, porém, em que a complexidade da aplicação de supervisão é grande, é importante que o sistema apresente equipamentos mais diversificados, tais como impressoras para saída de relatórios, fitas magnéticas e discos para o armazenamento de grande quantidade de dados, consoles para diálogo entre o sistema e os operadores, além de outros recursos.

### 3.2.1 - FILOSOFIA DE OPERAÇÃO

Considerando a arquitetura básica apresentada (Figura 3.1), propõe-se que na operação de supervisão uma UNIDADE funcione no MODO DE OPERAÇÃO ATIVO e a outra no MODO DE OPERAÇÃO PASSIVO. A UNIDADE ATIVA deve executar todas as funções de supervisão, enquanto a UNIDADE PASSIVA deve funcionar simplesmente como reserva da ATIVA. Assim, no caso de falha da ATIVA, a PASSIVA assume completamente suas funções. Este tipo de redundância provê a tolerância a falhas simples de "hardware", sem acarretar degradação no sistema.

A alocação dos processos de supervisão na memória pode ser estática. Neste caso ela deve ser feita em ambas as UNIDADES, ATIVA e PASSIVA, de forma idêntica, durante a inicialização do sistema.

Cada UNIDADE deve conter uma cópia do Sistema Operacional adotado (Seção 3.3).

Para funcionar como reserva, a UNIDADE PASSIVA deve ter conhecimento atualizado da situação dos processos (Seção 2.6) executados pelo processador da UNIDADE ATIVA. Isto é facilmente obtido via memória compartilhada.

A falha de uma UNIDADE é sentida pela outra na ausência de uma mensagem simples de controle, tipo EU ESTOU BEM, que deve ser enviada periodicamente de uma UNIDADE para a outra, e reciprocamente, através de uma linha de controle do barramento duplicado que une as duas UNIDADES (ATIVA e PASSIVA).

Quando a ausência desta mensagem for sentida pela UNIDADE PASSIVA, ela considera que a UNIDADE ATIVA falhou e desencadeia um procedimento para recuperação do sistema, tornando-se ATIVA.

Caso a ausência desta mensagem seja sentida pela UNIDADE ATIVA, nada deverá ocorrer em termos de reconfiguração do Sistema de Supervisão.

Em ambos os casos, a UNIDADE com defeito deve ser reparada e reconectada ao Sistema de Supervisão o quanto antes possível. Esta reconexão é bastante simples pois a UNIDADE reparada deve ser sempre reconfigurada como PASSIVA.

Uma proposta para implementação desta lógica é apresentada no Capítulo 4. Ela faz uso do temporizador CÃO DE GUARDA, ao nível de "hardware", e de manipuladores de exceções (Seção 2.4.2) ao nível de "software".

A arquitetura básica, juntamente com a filosofia de operação adotada, tem como característica fundamental a redundância dupla, necessária para atender os quesitos de disponibilidade de um Sistema de Supervisão.

## 3.3 - SISTEMA OPERACIONAL

Sobre o "hardware" apresentado na seção anterior deve operar um "software" básico que tem como função principal manter disponível os diversos recursos do sistema e viabilizar a execução dos programas de supervisão.

De modo geral, um sistema operacional voltado para aplicação em controle de processos pode ser visto como uma infra-estrutura básica que rege a utilização de recursos computacionais, permitindo ao operador comunicar-se com o sistema e interferir em sua atividade. Além disso, o sistema operacional deve ser também capaz de permanecer sempre disponível para reconhecer e atender a solicitações a estímulos externos, em tempo real.

O "software" básico considerado neste trabalho para apoiar programas aplicativos de supervisão é o Sistema Operacional iRMX86. Este pacote de "software" foi projetado para ser usado com os computadores INTEL iSBC 86/88 e com outros computadores baseados nos processadores iAPX 86,88. Os motivos que levaram à escolha deste sistema operacional foram os seguintes:

- é um Sistema Operacional *padrão*, disponível no mercado;
- é um Sistema Operacional *modular*, formado por uma coleção de subsistemas que podem ser configurados de acordo com a necessidade da aplicação;
- é um Sistema Operacional *multitarefas para controle de processos em tempo real*;
- é um Sistema Operacional *expansível*, que permite a criação de novos tipos de objetos e chamadas do sistema para manipulação destes objetos.

A seguir serão apresentadas a arquitetura básica do Sistema Operacional iRMX86 com detalhes de suas facilidades, consideradas relevantes para o presente trabalho.

A partir de agora a palavra *tarefa* será usada como sinônimo do termo *processo*, conceituado na Seção 2.1, até então utilizado nesta dissertação. Isto se deve a uma simples compatibilização de terminologias, visto que o iRMX86 adota o termo *tarefa*.

### 3.3.1 - ARQUITETURA DO SISTEMA OPERACIONAL iRMX86

A arquitetura básica do Sistema Operacional iRMX86, tal como representada na Figura 3.2 consiste no NÚCLEO e nos seguintes níveis: nível do SISTEMA BÁSICO DE ENTRADA/SAÍDA, nível do SISTEMA ESTENDIDO DE ENTRADA/SAÍDA, nível do CARREGADOR DA APLICAÇÃO e nível de INTERFACE COM OPERADOR.

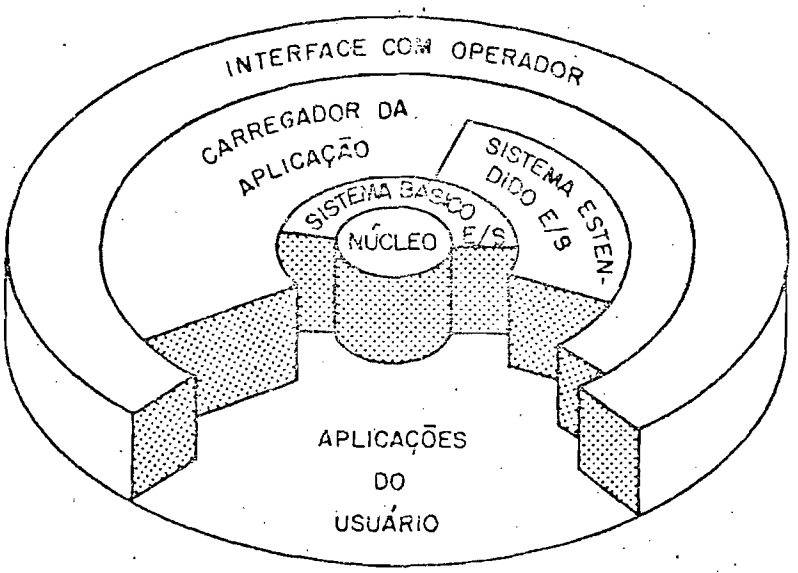


Fig. 3.2 - Arquitetura do Sistema Operacional iRMX86.

O *NÚCLEO* constitui o nível central do Sistema Operacional. Ele aloca recursos do processador, permite a comunicação entre processos e controla a alocação de memória, além de gerenciar as interrupções. Não é opcional.

O *SISTEMA BÁSICO DE ENTRADA/SAÍDA (BIOS)* provê a aplicação da capacidade de acessar arquivos em dispositivos periféricos. Isto é, as tarefas da aplicação se comunicam com arquivos associados aos dispositivos, não com os dispositivos propriamente ditos. Isto permite que as tarefas manipulem todos os arquivos de forma padronizada, inde

pendentemente dos dispositivos em que eles residem. A BIOS oferece controle total das operações de e/s para as aplicações, especialmente aquelas com acesso de e/s aleatórias. Este nível é um subsistema opcional do Sistema Operacional iRMX86, podendo ser excluído da configuração do Sistema Operacional se não for necessário.

O *SISTEMA ESTENDIDO DE ENTRADA/SAÍDA (EIOS)* é uma expansão do nível anterior, de fácil uso, especialmente em aplicações que usam e/s sequencial. Requer que a BIOS seja configurada. Possui a capacidade de associar nomes lógicos a dispositivos físicos de e/s. Isto permite que um usuário acesse dispositivos de e/s sem ter de escrever seus próprios procedimentos para especificá-los. O uso da EIOS é opcional.

O *CARREGADOR DA APLICAÇÃO* permite que a aplicação leia programas do disco na memória e execute-os. Também permite que um programa seja quebrado em segmentos chamados "overlays", para que todo programa não tenha de estar na memória de uma única vez. Oferece aos programadores maior flexibilidade no uso da memória pelos programas aplicativos. O sistema pode carregar programas em qualquer lugar disponível da memória, de tal forma que eles podem ser executados, mesmo que ocupem espaço de memória principal maior que o disponível. Este nível é também um subsistema opcional mas, se for utilizado requer que, no mínimo, o SISTEMA BÁSICO DE ENTRADA/SAÍDA seja também configurado.

Finalmente, o nível *INTERFACE COM OPERADOR* pode controlar o sistema de aplicação por comandos do terminal. É possível criar novos comandos para controlar o sistema de aplicação e incluí-los no conjunto de comandos comumente utilizados na operação normal. Tal como os dois níveis dos SISTEMAS DE E/S (BÁSICOS e ESTENDIDO), a INTERFACE COM OPERADOR é também um recurso opcional. Entretanto, se incluído requer que todos os outros níveis inferiores sejam configurados.

No âmbito deste trabalho considera-se suficiente e também de interesse apenas a configuração do NÚCLEO do Sistema Operacional iRMX86. Esta restrição procura atender aplicações simples de super

visão sem inviabilizar aquelas mais complexas que fazem uso dos outros níveis do Sistema Operacional iRMX86. Cabe ressaltar aqui que a configuração do último nível: INTERFACE COM OPERADOR é bastante conveniente em Sistemas de Supervisão, pois ela implementa uma linguagem de controle que deve permitir um fácil acesso aos recursos do sistema, com o uso de comandos de baixa complexidade e de fácil aprendizado. É interessante notar que o número de comandos não deve ser muito grande, pois é conveniente para o operador que ele nunca tenha dúvidas quanto à ocasião em que deve empregar cada um dos comandos, adequadamente. Um outro recurso oferecido pela INTERFACE COM O OPERADOR é que este deve, sempre que cometer erros, ser informado pelo sistema, de maneira clara e precisa, preferivelmente com mensagens fornecidas por extenso. Isto evita que o operador seja obrigado a consultar tabelas de código de mensagens. Além disso, as listagens emitidas pelo sistema (tais como relatórios) e os dados que são fornecidos a ele (tais como parâmetros e comandos) devem ser apresentados de maneira clara e legível, sem dar margem a dúvidas quanto ao seu significado.

### 3.3.2 - ASPECTOS DO NÚCLEO DO iRMX86

Alguns aspectos do NÚCLEO do Sistema Operacional iRMX86, considerados relevantes para este trabalho, são apresentados nesta seção.

O Sistema Operacional iRMX86 compõe-se de um conjunto de processos básicos e estruturas de dados que apresentam características do tipo particionamento de programas em TAREFAS, escalonamento de tarefas e comunicação entre tarefas. Estas características são apoiadas pelo NÚCLEO do iRMX86 e utilizadas pelos demais níveis deste Sistema Operacional, assim como pela aplicação.

#### 3.3.2.1 - MULTITAREFAS

O iRMX86 usa multitarefas para simplificar o desenvolvimento de aplicações que processam *eventos de tempo real* como é o caso de sistemas de controle de processos em tempo real, particularmente de Sistemas de Supervisão.

A essência de aplicaçõess de tempo real consiste na habilidade de processar inúmeros eventos que ocorrem em instantes de tempo aleatório. Estes eventos são *assíncronos* porque eles podem ocorrer a qualquer instante de tempo e são potencialmente *concorrentes* porque um evento pode ocorrer enquanto outro está sendo processado.

Em aplicações de supervisão existe a necessidade intrínseca de processar eventos múltiplos, concorrentes e assíncronos. O sistema deve conhecer quais os eventos ocorridos e a ordem em que ocorreram, além de quais ainda não foram processados num dado instante. A complexidade, obviamente, aumenta com o número de eventos a serem monitorados.

Multitarefas é uma técnica que elimina a necessidade de monitorar a ordem em que os eventos ocorreram. Ao invés de escrever um único programa para processar N eventos, escrevem-se N programas de modo que cada um deles processe um único evento. Assim, cada um destes N programas forma uma TAREFA do iRMX86, que é o único tipo de objeto ativo do Sistema Operacional iRMX86.

### 3.3.2.2 - ARQUITETURA ORIENTADA PARA OBJETOS

A arquitetura do iRMX86 é orientada para *objetos*, os quais são entidades especiais gerenciadas pelo NÚCLEO do Sistema Operacional e utilizadas pelo programador da aplicação para fazer uso dos recursos providos pelo Sistema Operacional. Os tipos de objetos providos pelo Sistema Operacional iRMX86 são os seguintes:

1 - *Tarefas* - É o único tipo de objeto do iRMX86 dito ativo, pois só através dele o usuário pode invocar as primitivas (chamadas ao sistema) providas pelo iRMX86. As tarefas implementam as atividades de supervisão do Sistema de Supervisão considerado.

2 - *"Jobs"* - Caracteriza o ambiente onde residem os objetos (tarefas, caixas postais, semáforos, segmentos e outros "jobs") definidos pela aplicação. Todo "job" possui um *diretório de objetos* e um *poço de memória* (área onde os objetos podem ser criados pelas tarefas daquele "job"). Os "jobs" apoiam a multiprogramação, pois são independentes entre si. Uma aplicação pode consistir em um ou mais "jobs" que, apesar de serem independentes, podem compartilhar recursos. Cada "job" possui suas próprias tarefas e seu próprio diretório de objetos. Apesar de os objetos poderem ser compartilhados entre "jobs", um objeto deve estar contido no diretório de um único "job". Cabe ao programador decidir quais as tarefas que devem estar no mesmo "job". Em geral duas ou mais tarefas pertencem a um mesmo "job" se elas:

- têm propósito similar ou relacionado;
- compartilham muitos recursos;
- têm tempo de vida similar.

Para melhor controle do NÚCLEO do iRMX86, os "jobs" do sistema estão organizados numa estrutura de árvore (Árvore de "JOB"), como é mostrado na Figura 3.3.

Cabe notar que o JOB raíz é provido pelo NÚCLEO do iRMX86 e os JOBs de primeiro nível (jobs nível 1) são definidos pela aplicação, na configuração do sistema, para que eles sejam criados automaticamente na inicialização do sistema. Os demais "jobs" e suas tarefas podem ser criados dinamicamente durante a operação do sistema. Estes "jobs" são ditos descendentes do "jobs" de primeiro nível.

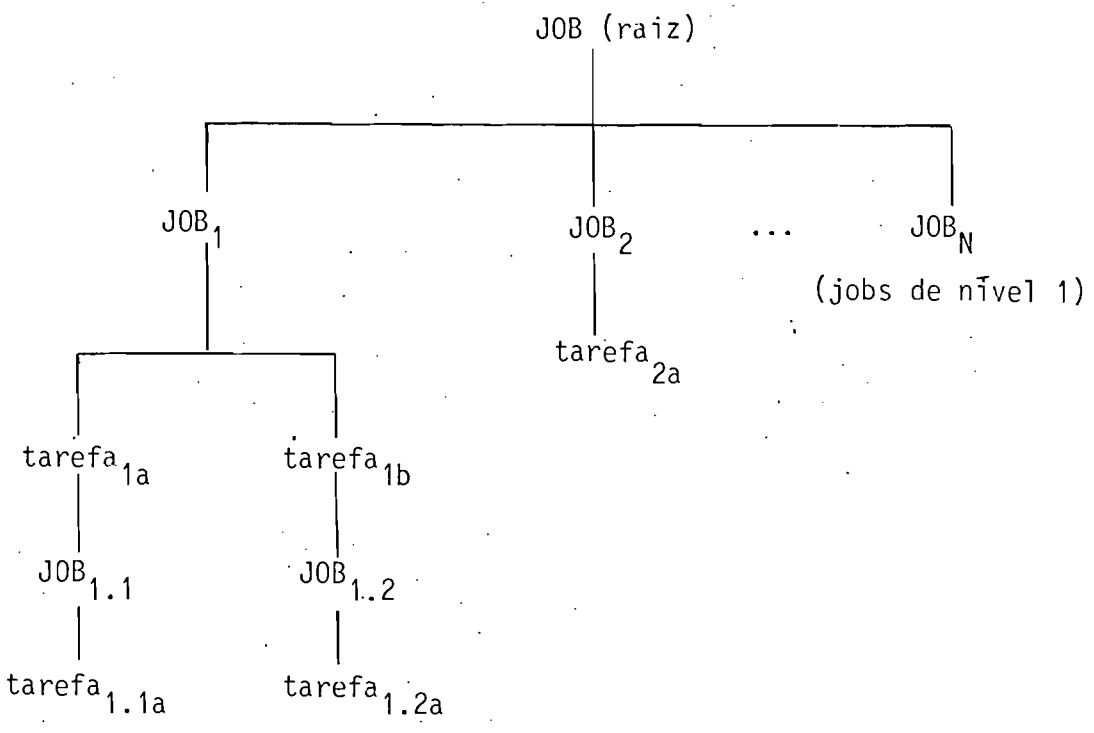


Fig. 3.3 - Árvore de "JOBs".

3 - *Segmentos* - São pedaços de memória utilizados pelas tarefas para comunicação e armazenamento de dados. Um SEGMENTO é uma sequência contínua de parágrafos de 16 bytes, com seu endereço base divisível por 16 (Figura 3.4). Os SEGMENTOS são alocados às tarefas através de chamadas ao NÚCLEO de forma que, quando uma tarefa requer um segmento, ela invoca uma primitiva (chamada ao sistema) do NÚCLEO (CREATE$SEGMENT) que cuida de criar este segmento a partir do poço de memória do JOB ao qual a TAREFA pertence. Caso não haja memória disponível, o NÚCLEO tentará emprestá-la de antecessores do JOB, utilizando para isto a hierarquia de estrutura da árvore de "jobs" do sistema.

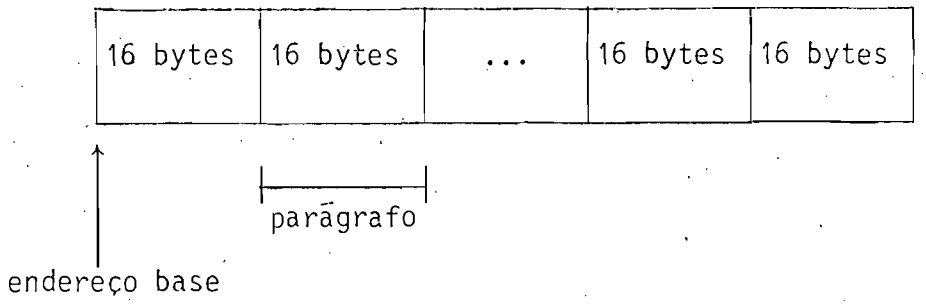


Fig. 3.4 - Segmento de memória.

4 - *Caixa postal* - É um objeto utilizado para comunicação entre tarefas. Uma tarefa fonte usa uma CAIXA POSTAL para enviar a identificação de um objeto (segmento) para outra tarefa que se diz destino. Cada CAIXA POSTAL possui duas filas: uma, FILA DE TAREFAS, contém as tarefas que estão esperando para receber objetos (segmentos); e outra, FILA DE SEGMENTOS, contém os objetos (segmentos) que foram enviados pelas tarefas fonte, mas que ainda não foram recebidos pelas tarefas-destino.

5 - *Semáforos* - Capacitam as tarefas a enviar sinais para outras tarefas. Esta capacidade não apoia a comunicação de dados mas facilita a exclusão mútua, a sincronização e a alocação de recursos. Cada SEMÁFORO possui apenas uma fila associada, FILA DE TAREFAS, que contém as tarefas que estão esperando pela sinalização do semáforo. Esta fila pode ser organizada por prioridade ou da forma FIFO ("First-In/First-Out"). O esquema adotado é especificado na criação de cada semáforo.

6 - *Região* - É um objeto do iRMX86 usado pelas tarefas para guardar uma coleção específica de dados compartilhados. Uma vez que uma tarefa ganha o acesso aos dados compartilhados, através de uma REGIÃO, esta tarefa não pode ser suspensa nem descartada por outras tarefas até que ela devolva a REGIÃO. A devolução é feita via o NÚCLEO do Sistema Operacional iRMX86 através da chamada ao sistema SEND$CONTROL.

7 - *Extensões e composições* - Referem-se aos objetos que permitem a incorporação, ao Sistema Operacional iRMX86, de novos tipos de objetos e chamadas ao sistema, mais adequados a funções específicas da aplicação. Os procedimentos que devem ser adicionados ao Sistema Operacional para apoiar uma função requerida são denominados Extensões do Sistema Operacional. Um *Gerente de Tipo* é uma destas extensões capaz de criar um novo tipo de objeto. Apesar de um dado Gerente de Tipo estar associado a apenas um tipo de objeto, ele pode criar vários objetos deste tipo (composições).

Cada tipo de objeto definido pelo Sistema Operacional iRMX86 possui um conjunto de *chamadas ao sistema* (primitivas). Estas chamadas estão resumidamente apresentadas no Apêndice A desta dissertação. Como exemplo, as possíveis chamadas ao sistema permitidas a objetos do tipo TAREFA são as seguintes:

- CRIA objeto;
- DESCARTA objeto;
- ENVIA segmentos para outras tarefas;
- RECEBE segmentos de outras tarefas;
- OBTÉM informações sobre objetos;
- CATALOGA objeto com novos descritores;
- DESCARTA objeto dos catálogos.

### 3.3.2.3 - ESCALONAMENTO BASEADO EM PRIORIDADE

O Sistema Operacional iRMX86 utiliza escalonamento preemptivo baseado em prioridade para decidir qual tarefa deve receber o controle do processador num dado instante. Esta técnica garante que se uma tarefa de maior prioridade que aquela que está sendo executada torna-se pronta, a mais prioritária ganha o controle do processador imediatamente. O escalonamento *preemptivo por prioridade* facilita o processamento de interrupções pois as prioridades das tarefas podem ser

associadas à importância relativa dos eventos que elas processam. Isto permite que os eventos mais importantes sejam processados antes daqueles menos importantes.

Para arbitrar a concorrência das tarefas no uso do processador, o NÚCLEO mantém, para cada tarefa, um estado de execução e uma prioridade (valor inteiro entre 0 e 255).

Uma tarefa deve sempre se encontrar em um dos 5 estados de execução:

DORMENTE - quando a tarefa estiver esperando por uma mensagem, semáforo ou região (RECEIVE$MESSAGE, RECEIVE$UNITS, RECEIVE$CONTROL). Também uma tarefa pode se colocar neste estado por um tempo específico através de uma chamada ao sistema ("SLEEP").

SUSPENSO - uma tarefa entra neste estado quando ela própria solicita sua suspensão, ou quando outra tarefa a suspende, ou mesmo quando uma tarefa que trata de uma interrupção específica de "hardware" sinaliza que ela, a tarefa, está pronta para servir à interrupção associada (WAIT$INTERRUPT). Associado a cada tarefa existe uma campo de *profundidade de suspensão* que reflete quantas vezes ela foi suspensa. Cada operação "SUSPEND" deve ser contraposta por uma "CONTINUE" para que a tarefa possa mudar para outro estado.

DORMENTE-SUSPENSO - uma tarefa só entra neste estado se ela estiver no estado dormente e for suspensa por outra tarefa. Neste caso, se a tarefa devesse dormir durante um certo tempo e este tempo termina, então ela automaticamente é colocada no estado SUSPENSO. Da mesma forma, se outra tarefa emite um comando "CONTINUE" para uma tarefa que se encontra no estado DORMENTE-SUSPENSO, tirando-a da condição de SUSPENSA, esta tarefa passa para o estado DORMENTE.

PRONTO - uma tarefa encontra-se neste estado quando ela não estiver em nenhum dos estados anteriores.

EM EXECUÇÃO - a única tarefa que se encontra neste estado num dado instante é aquela que se encontrava no estado PRONTO, com maior prioridade. Isto é, aquela que possui o controle do processador.

A Figura 3.5 mostra a transição de estados de execução permitida pelo NÚCLEO. Cabe notar que o NÚCLEO não aloca o processador para tarefas na forma de fatias de tempo ("time slice"). Ao invés, a ocorrência de evento é que causa a transição de estado de uma tarefa, o que significa que o Sistema Operacional iRMX86 é orientado pela ocorrência de eventos.

Toda tarefa é criada no estado PRONTO. Uma tarefa EM EXECUÇÃO volta ao estado PRONTO se uma outra tarefa de maior prioridade torna-se pronta (preempção). Se uma tarefa foi colocada no estado DORMENTE por um determinado tempo e este se esgotou, naturalmente ela passará para PRONTO. Ou se ela estava DORMENTE porque, por exemplo, esperava por um semáforo e este semáforo foi-lhe concedido, então ela passará ao estado PRONTO.

Uma tarefa passa de SUSPENSO para PRONTO quando ela é tirada de suspensão por outra tarefa ou quando a tarefa de atendimento de uma interrupção quer "avisar" à tarefa de interrupção que ela pode tratar daquela interrupção, pois ela acabou de ocorrer (SIGNAL$INTERRUPT) (vide mais detalhes em 3.3.2.6).

Como o algoritmo de escalonamento utilizado pelo iRMX86 é preemptivo por prioridade, cabe notar que no caso específico de um Sistema de Supervisão a tarefa de *aquisição* de dados deve receber maior prioridade para evitar que os dados de aquisição possam ser perdidos, a tarefa de *atuação* deve ter prioridade intermediária e as prioridades mais baixas devem ser atribuídas às tarefas de *processamento*.

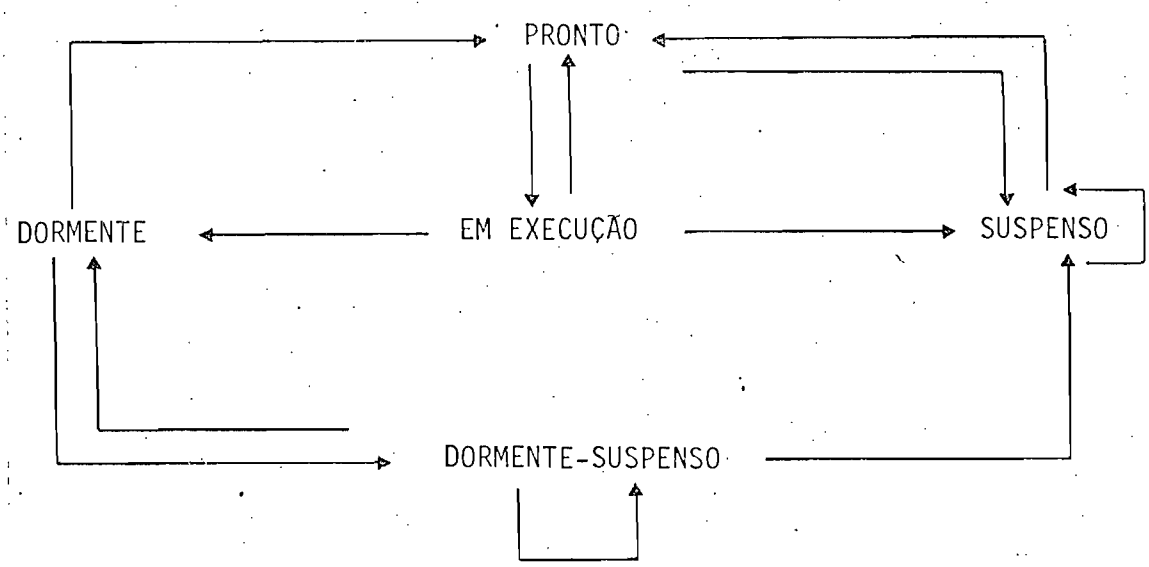


Fig. 3.5 - Transição de estados de execução.

3.3.2.4 - COORDENAÇÃO ENTRE TAREFAS

O Sistema Operacional iRMX86 é do tipo *multitarefas*, provendo técnicas simples para coordená-las. Estas técnicas permitem: a exclusão mútua, a sincronização e a comunicação entre tarefas.

Como se sabe multitarefas é uma técnica usada para simplificar o projeto de aplicações de tempo-real que monitoram múltiplos eventos concorrentes e assíncronos. Assim, o sistema multitarefas permite que os projetistas focalizem sua atenção no processamento de um único evento ao invés de considerar a ocorrência de todos eventos em ordem aleatória. Entretanto, o processamento de certos eventos podem estar relacionados entre si. Por exemplo, o processamento do evento A, quando ele ocorre, pode precisar saber quantas vezes o evento B ocorreu. Este tipo de processamento requer que certas tarefas possam ser capazes de se coordenarem entre si.

Assim, uma tarefa pode interagir com outras de três maneiras:

(i) *Exclusão Mútua*

A *exclusão mútua* implementa o partilhamento de recursos entre tarefas de modo a garantir que num dado instante não mais que uma tarefa pode usar o recurso. A exclusão mútua é provida pelo objeto *semáforo*. Um semáforo é um contador inteiro que as tarefas podem manipular através de chamadas ao núcleo. O semáforo pode assumir apenas valores inteiro não-negativos. As tarefas podem modificar o valor do semáforo usando chamadas ao núcleo do tipo SEND$UNITS ou RECEIVE$UNITS. Quando uma tarefa envia N unidades (> = 0) para um semáforo, o valor do contador é aumentado de N. Quando uma tarefa usa a chamada ao sistema do tipo RECEIVE$UNITS para requsitar M unidades (> = 0) de um semáforo, uma das duas coisas acontece:

(1) - se o contador do semáforo for maior ou igual a M, o núcleo reduz o contador de M e continua a executar a tarefa;

(2) - caso contrário, contador do semáforo menor que M, o núcleo coloca outra tarefa em execução, e a tarefa requisitante *espera* no estado DORMENTE pelo semáforo até que o contador seja igual a M ou maior.

Um semáforo deve ser criado com valor inicial 1. Antes que qualquer tarefa possa utilizar o recurso compartilhado, ela deve receber uma unidade do semáforo. Também, assim que a tarefa termine de usar o recurso, ela deve enviar uma unidade para o semáforo. Isto garante que em qualquer momento não mais que uma tarefa pode usar o recurso simultaneamente e que qualquer outra tarefa que deseja usar o recurso tem que esperá-lo retornar ao semáforo.

(ii) *Sincronização*

A *sincronização* entre tarefas é obtida pelo uso de *semáforos*. Neste caso, o semáforo deverá ser criado com valor inicial zero. Assim, uma tarefa A, que precisa ser executada somente depois que a tarefa B termine, pode ser sincronizada. A tarefa A é naturalmente

suspensa ao requisitar uma unidade do semáforo (RECEIVE$UNITS) e permanece neste estado até que a tarefa B envie uma unidade para o semáforo.

A coordenação entre tarefas provida pelo Sistema Operacional iRMX86 é flexível e simples de usar. Os *semáforos* e *caixas postais* podem acomodar uma grande variedade de situações. Cada aplicação não tem, em princípio, número limitado de caixas postais ou de semáforos. Pode-se criar tantos objetos quantos forem necessários.

(iii) *Troca de Informações*

A *comunicação entre tarefas* é feita por *caixas postais*, manipuladas pelo núcleo. Isto é, as tarefas usam caixas postais para enviar informações para outras tarefas (estas informações deverão estar num *segmento*) e usam a chamada ao sistema "SEND$MESSAGE" para enviar o segmento para a caixa postal destino. A outra tarefa então invoca a chamada ao sistema "RECEIVE$MESSAGE" para obter o acesso ao segmento contido naquela caixa postal. As caixas postais permitem a comunicação entre tarefas assíncronas.

Como já citado anteriormente, neste capítulo, cada caixa postal tem associada a ela duas filas:

*Fila de tarefas* - fila que contém as tarefas que esperam para receber objetos (segmentos);

*Fila de segmentos* - fila que contém os objetos (segmentos) que foram enviados pelas tarefas fontes, porém ainda não recebidos pelas respectivas tarefas destinos.

O NÚCLEO garante que todas as tarefas que estão esperando objetos na FILA DE TAREFAS os recebem assim que eles estiverem disponíveis. Isto significa que estes objetos nem são colocados na FILA DE SEGMENTOS. Assim sendo, num dado instante, para um dado objeto (segmento), no mínimo uma das duas filas da caixa postal encontra-se vazia.

A FILA SEGMENTOS de uma caixa postal é processada na forma FIFO ("First-In/First-Out"). Entretanto a FILA TAREFAS pode ser também por prioridades (a tarefa de maior prioridade ocupa a primeira posição da fila). Esta decisão deve ser tomada quando uma tarefa cria uma caixa postal. A tarefa especifica qual tipo de FILA TAREFAS a caixa postal deve ter.

Quando uma tarefa envia um sinal para uma caixa postal, através de um SEND$MESSAGE uma das situações acontece:

(1) - se nenhuma tarefa estiver esperando naquela caixa postal, o objeto é colocado por último na FILA SEGMENTOS (que pode estar vazia). Assim, o objeto permanece na fila até que ele atinja a primeira posição e seja repassado a uma tarefa;

(2) - se, por outro lado, existir tarefa esperando, a tarefa receptora que se encontra no estado DORMENTE sairá deste estado tornando-se PRONTA. Caso a tarefa receptora estivesse no estado DORMENTE-SUSPENSO, ela passaria ao estado SUSPENSO.

3.3.2.5 - GERENCIAMENTO DE MEMÓRIA

O Sistema Operacional iRMX86 apoia a alocação dinâmica de memória. Ocasionalmente uma tarefa necessita de memória adicional. Através de chamadas ao NÚCLEO para alocar e descartar segmentos, a tarefa pode satisfazer suas necessidades.

Como citado anteriormente, neste capítulo, cada "job" possui um *poço de memória*, que consiste na memória disponível para aquele "job" e seus descendentes. Quando um "job" é criado, a memória para seu poço é alocada do poço de memória do seu pai. Assim, existe uma *hierarquia de poços de memória*, estruturada em árvore, idêntica à estrutura para a hierarquia de "jobs". Toda memória que um "job" empresta do seu pai permanece no poço do pai mesmo pertencendo ao poço de memória do filho. Tal memória, entretanto, está disponível apenas para as tare

fas do "job" filho. A Figura 3.6 ilustra a relação entre as hierarquias de "jobs" e a memória. Nesta figura, os tamanhos mostrados correspondem ao tamanho máximo daqueles poços.

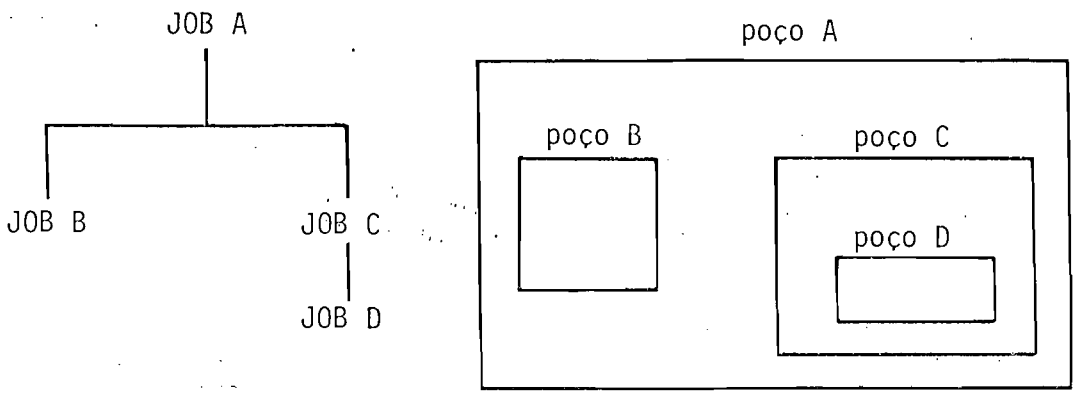


Fig. 3.6 - Hierarquias de "jobs" e memórias.

Para criar JOBs e TAREFAs o iRMX86 oferece respectivamente duas chamadas ao sistema: CREATE$JOB e CREATE$TASK. Estas chamadas possuem alguns parâmetros de entrada que permitem que o usuário aloque seus JOBs e TAREFAs em posições de memória específicas. Pode-se citar como exemplo destes parâmetros na criação de uma TAREFA:

- endereço do início da TAREFA;
- tamanho do segmento que contém a TAREFA;
- endereço do segmento de dados da TAREFA;
- endereço do início da pilha da TAREFA;
- tamanho do segmento que contém a pilha da TAREFA.

Para especificar o tamanho do poço de memória de um JOB são utilizados dois parâmetros: POOLMAX e POOLMIN (em parágrafos de 16 bytes). Inicialmente, o tamanho do poço é igual ao POOLMIN, e toda a memória alocada para as TAREFAS do JOB pertencem a este poço.

O poço de memória de um "job" consiste em duas classes de memória: alocada e não-alocada. A memória é considerada não-alocada a menos que ela tenha sido requisitada, implícita ou explicitamente, pelas tarefas do "job" ou tenha sido emprestada a um "job" filho.

Diz-se que uma requisição de memória é explícita quando a tarefa cria um objeto do tipo SEGMENTO (CREATE$SEGMENT).

Uma requisição é implícita quando a tarefa cria outro tipo de objeto diferente de SEGMENTO.

Deve-se citar que toda vez que um objeto é descartado, a área de memória que ele ocupava é devolvida ao poço de memória do "job" ao qual o objeto pertencia (DELETE$objeto).

O Sistema Operacional iRMX86 permite que uma tarefa obtenha algumas informações a respeito do poço de memória do "job" ao qual ela pertence (GET$POOL$ATTRIB). Uma consulta deste tipo fornece as seguintes informações:

- "Poolmax";
- "Poolmin";
- Tamanho do poço inicial;
- Número de parágrafos disponíveis (não-alocados);
- Número de parágrafos já alocados.

3.3.2.6 - PROCESSAMENTO DE INTERRUPÇÕES

O núcleo ainda é responsável pelo gerenciamento de interrupções. Este aspecto do núcleo é muito importante pois interrupções e seu processamento são essenciais para a computação de tempo real. Em geral, os eventos externos ocorrem assincronamente em relação à execução interna das aplicações no sistema. No Sistema Operacional iRMX86, uma *interrupção* que sinaliza a ocorrência de um evento externo desvia, implicitamente, o controle do programa para um local de memória especificado na seção de memória conhecida por TABELA DO VETOR

DE INTERRUPÇÃO. De lá, o controle é redirecionado para um procedimento chamado *Manipulador de Interrupção*. Neste caso uma das duas situações seguintes pode acontecer:

(1) - se a *manipulação da interrupção* toma pouco tempo e não faz nenhuma chamada ao sistema, diferente daquelas permitidas, o próprio manipulador de interrupção processa a interrupção;

(2) - de outra forma, o manipulador de interrupção invoca uma *Tarefa de Interrupção* para tratamento da interrupção específica. Depois de ela ter sido tratada, o sistema operacional retorna o controle para a tarefa da aplicação de maior prioridade, no momento.

Existem três recursos principais que direcionam o processamento de interrupções: (1) a tabela de vetores de interrupções; (2) os níveis de interrupções e (3) a desabilitação dos níveis de interrupção.

A *Tabela de Vetores de Interrupção* é composta de 256 vetores numerados de 0 a 255. Alguns destes vetores de interrupção são reservados, não estando disponíveis para ser definidos pelas tarefas da aplicação.

Com relação a níveis de interrupção, o núcleo gerencia um ambiente expandido de interrupções (em cascata), no qual até 7 linhas de entrada de um mestre podem ser conectadas a escravos. A oitava linha do mestre é conectada diretamente ao relógio do sistema. Já que cada escravo pode gerenciar até 8 interrupções, isto permite que o Sistema Operacional gerencie até 56 fontes externas (níveis de interrupções), além daquela do relógio do próprio sistema.

Em termos de desabilitação dos níveis de interrupções, ocasionalmente pode-se desejar impedir que certos sinais de interrupção causem uma interrupção imediata. Para estes casos, o iRMX86 permi

te que cada nível de interrupção possa ser desabilitado. Em certas circunstâncias o próprio núcleo desabilita tais níveis. Porém, as tarefas também podem desabilitar ou habilitar níveis de interrupção através das chamadas DISABLE e ENABLE, respectivamente. O nível de interrupção reservado para a base de tempo do sistema não pode ser desabilitado nem habilitado.

Se um sinal de interrupção chega num nível que está habilitado, o Sistema Operacional transfere o controle para o endereço contido na tabela de vetor de interrupção correspondente àquele nível de interrupção. Caso o nível esteja desabilitado, o sinal de interrupção é bloqueado até que o nível seja habilitado, quando então o sinal será reconhecido pela UCP (Unidade Central de Processamento). Entretanto, se a fonte do sinal parar de transmití-lo, ele não mais será reconhecido e aquela interrupção específica não será mais atendida, até nova ocorrência.

Como já foi citado anteriormente, o Sistema Operacional iRMX86 permite que uma *tarefa de interrupção* seja associada a cada nível de interrupção de modo que um *manipulador de interrupção* pode ou não ter associado a ele uma *tarefa de interrupção*. Quando a tarefa de interrupção existe, ela deve se encontrar no estado SUSPENSO, à espera da interrupção associada, até que ela ocorra.

Cabe citar que não é permitido o aninhamento ("nesting") de atendimento de interrupção dentro de manipuladores de interrupção.

A tarefa de interrupção tem prioridade relativa ao nível da interrupção associada a ela, mantendo esta relação com as prioridades das outras tarefas do sistema.

### 3.3.2.7 - EXPANSIBILIDADE

A *expansibilidade* permite que novos tipos de objetos sejam criados e adicionados às chamadas do Sistema Operacional. Caso mais de um "job" de uma aplicação requisite uma função não-provida

pelo Sistema Operacional iRMX86, o projetista da aplicação pode construir seus próprios tipos de objetos e chamadas ao sistema para manipulá-los. Desta forma, estes novos objetos são incorporados ao Sistema Operacional. A vantagem da expansibilidade consiste na possibilidade de criar um Sistema Operacional que atenda precisamente as necessidades da aplicação sem perder os benefícios da arquitetura orientada para objetos já oferecidos pelo iRMX86. Tais benefícios incluem a facilidade de enviar estes novos objetos para caixas postais e a facilidade de colocá-los nos diretórios de objetos próprios de cada "job".

Maiores detalhes sobre como criar novos tipos de objetos e adicioná-los ao Sistema Operacional serão descritos na Seção 3.4.

3.3.2.8 - MANIPULAÇÃO DE EXCEÇÕES

Com respeito ao último aspecto citado, o de *manipulação de exceções*, o Sistema Operacional iRMX86 permite que uma aplicação especifique um procedimento de manipulação de erro para cada tarefa.

Como se sabe, *manipulação de erro* é o processo que permite detetar e reagir a condições não-esperadas. O iRMX86 apoia a manipulação de erros na realização de teste de condição nas chamadas ao sistema (se as chamadas foram feitas corretamente, com parâmetros certos, então a chamada é reconhecida pelo sistema). Apesar de não poder detetar qualquer tipo de erro, o Sistema Operacional iRMX86 protege a aplicação de certos tipos de erros. Os conceitos envolvidos no esquema de manipulação de erro do iRMX86 são: códigos de condição e manipuladores de exceções.

1 - *Códigos de condição* - Quando uma tarefa faz uma chamada ao sistema, o iRMX86 executa a função requerida. Qualquer que seja o resultado desta chamada (com sucesso ou não) o Sistema Operacional gera um código de condição. Este código indica se a chamada ao sistema foi bem sucedida ou falhou e, em caso de falha, o código de condição mostra qual a condição que impediu o sucesso. Na Tabela 3.1 estão listadas as condições excepcionais detetadas pelo Sistema Operacional iRMX86.

TABELA 3.1

CONDIÇÕES EXCEPCIONAIS DETECTADAS PELO iRMX86

| CATEGORIA/mnemônico | SIGNIFICADO |
|---|---|
| CONDIÇÃO AMBIENTAL/ | |
| E$TIME | O limite de tempo se esgotou sem que a requisição da tarefa tenha sido satisfeita. |
| E$MEM | Não há memória disponível suficiente para satisfazer a requisição da tarefa. |
| E$LIMIT | Uma tarefa tentou executar uma operação que, se tivesse sido bem sucedida, teria violado um limite do núcleo. |
| E$CONTEXT | A chamada ao sistema estava fora de contexto, ou o núcleo foi chamado para executar uma operação inexistente. |
| E$EXIST | Um sinal de parâmetro tinha um valor diferente do sinal de um objeto existente. |
| E$STATE | Uma tarefa tentou executar uma operação que causaria uma transição de estado imprevista pelo núcleo. |
| E$NOT$CONFIGURED | A chamada ao sistema não faz parte da presente configuração do "software". |
| E$INTERR$SATURATION | Uma tarefa de interrupção atingiu a quantidade máxima permitida de requisições SIGNAL$INTERRUPT. |
| E$INTERR$OVERFLOW | Uma tarefa de interrupção superou a quantidade máxima permitida de requisições SIGNAL$INTERRUPT. |
| ERRO DO PROGRAMADOR/ | |
| E$ZERO$DIVIDE | Uma tarefa tentou dividir um número por zero. |
| E$OVERFLOW | Um estouro de interrupções acorreu. |
| E$TYPE | Um sinal de parâmetro referido por um objeto existente não é do tipo requerido. |
| E$PARAM | Parâmetro (que não é sinal nem "offset") com um valor não permitido. |

(continua)

Tabela 3.1 - Conclusão

| CATEGORIA/mnemônico | SIGNIFICADO |
|---|---|
| E$BAD$CALL | Uma tarefa escreveu sobre a biblioteca da interface ou tentou uma interrupção de "software" restrita. |
| E$ARRAY$BOUNDS | O "hardware" ou a linguagem detetou um estouro de "array". |
| E$NPD$ERROR | Erro detetado no processador aritmético 8087. |

2 - *Manipuladores de exceções* - Um manipulador de exceção é um procedimento que o Sistema Operacional iRMX86 pode invocar quando uma tarefa recebe um código de condição, indicando falha de uma função requerida. Assim como uma tarefa é criada, ela é também associada a um manipulador de exceção ("exception handling"), que pode ser utilizado para tratamento de falha, como poderá ser visto no Capítulo 4. O manipulador de exceção pode ser escrito pelo programador ou provido pelo Sistema Operacional ("default"). A facilidade de escrever o manipulador de exceção permite o controle de aplicação, quando ela recebe um código de condição. O manipulador pode recuperar erro, descartar a tarefa que contém o erro, prevenir o operador do erro, ou ignorar o erro. Assim, um manipulador de exceção trabalha da seguinte maneira: o Sistema Operacional gera um *código de condição* para cada chamada ao sistema. Se o código indicar sucesso, então nada se tem a fazer. Se o código indicar uma *condição excepcional*, o código excepcional pode ser processado dentro do próprio procedimento que faz a chamada ou por um *manipulador de exceção* invocado pelo Sistema Operacional. A técnica usada é uma característica de cada tarefa, sendo estabelecida quando a tarefa é incorporada ao sistema.

## 3.4 - GERENTE DE TIPO

Como citado na seção anterior, a *expansibilidade* permite que seja criado um Sistema Operacional mais adequado às necessidades da aplicação. Isto é possível incluindo no Sistema Operacional iRMX86 novos objetos e chamadas ao sistema. Um *Gerente de Tipo* é uma extensão do Sistema Operacional que provê este tipo de serviço.

Um Gerente de Tipo deve ser escrito para cada novo tipo de objeto a ser introduzido no Sistema Operacional. A responsabilidade de um Gerente de Tipo inclui:

- implementação do novo tipo, criando objetos deste novo tipo;
- provisão de um mecanismo para descartar objetos deste novo tipo;
- provisão de chamadas ao sistema que as tarefas da aplicação podem invocar, opcionalmente, para criar, manipular e descartar objetos do novo tipo.

A criação de um novo objeto compreende dois passos:

(i) - criar o tipo;

(ii) - criar objetos para aquele tipo.

A chamada ao sistema CREATE$EXTENSION cria o tipo. Isto é, retorna um sinal que representa uma licença para criar objetos daquele novo tipo. Então, CREATE$COMPOSITE cria objetos do novo tipo.

Deve-se notar que uma vez que um Gerente de Tipo cria um novo objeto, este gerente passa a administrar o tipo. Ele é o único Gerente de Tipo que pode criar objetos compostos a partir daquele tipo.

Existem duas chamadas ao sistema disponíveis para manipulação de objetos compostos:

INSPECT$COMPOSITE - retorna uma lista de sinais de componentes para um objeto composto;

ALTER$COMPOSITE - troca um sinal, na lista de componentes de um objeto composto, por outro sinal ou "null".

Duas chamadas ao sistema estão disponíveis exclusivamente para descartar objetos compostos:

DELETE$COMPOSITE - descarta um objeto composto específico, mas não os seus componentes;

DELETE$EXTENSION - descarta um tipo de extensão específico do objeto, podendo descartar seus componentes ou enviá-los para uma caixa postal de descarte.

Para escrever um Gerente de Tipo deve-se considerar que ele consiste em duas partes:

Parte de Inicialização - cria o tipo e opcionalmente uma caixa postal DESCARTE para a qual o sistema envia sinais para objetos daquele tipo, quando descarta "jobs" ou o próprio tipo.

Parte de Serviço - provê chamadas ao sistema através das quais tarefas podem criar e manipular objetos do tipo.

### 3.4.1 - PARTE DE INICIALIZAÇÃO

Esta parte deve ser escrita como uma tarefa e deve ser garantido que ela é executada o mais cedo possível na vida do sistema. Para tanto esta tarefa deve ser parte da tarefa de inicialização de um "job" de primeiro nível da árvore de "jobs" (Seção 3.3.2).

### 3.4.2 - PARTE DE SERVIÇO

A Parte de Serviço do Gerente de Tipo é escrita como uma Extensão do Sistema Operacional. Isto corresponde à escrita de procedimentos ("procedures") que implementam as novas chamadas ao sistema e à inicialização do vetor de interrupções. Estas *novas chamadas ao sistema* serão utilizadas pelas tarefas da aplicação para operar o novo tipo de objeto.

Na realidade, existem quatro tipos de procedimentos que o projetista deve escrever para compor uma *Extensão do Sistema Operacional:* Procedimento de Interface, Procedimento de Entrada, Procedimento de Função e Procedimento RQ$ERROR. Os três primeiros tipos citados devem ser procedimentos do tipo reentrante.

A Extensão do Sistema Operacional iRMX86 com uma nova função que possa ser invocada pelos programas da aplicação, como uma chamada ao sistema (primitiva), envolve a escrita de no mínimo dois tipos de procedimentos citados acima: Procedimento de Interface e Procedimento de Função.

O *Procedimento de Interface* tem a função de conectar a nova chamada ao Sistema Operacional já existente.

O *Procedimento de Função* é responsável por executar a nova função requisitada pela tarefa da aplicação quando faz a nova chamada ao sistema.

Para que as novas chamadas ao sistema sejam conservadas, um *Procedimento de Entrada* deve ser utilizado. Ele serve como um multiplexador de extensões do Sistema Operacional para apoiar mais que uma nova chamada ao sistema. Assim sendo, ele tem o propósito de rotear a chamada do Procedimento de Interface para o correspondente Procedimento de Função. É um procedimento opcional. Porém, sua vantagem consiste no aproveitamento de uma única entrada no Vetor de Interrupção para apoiar múltiplas chamadas ao sistema (vide Figura 3.7).

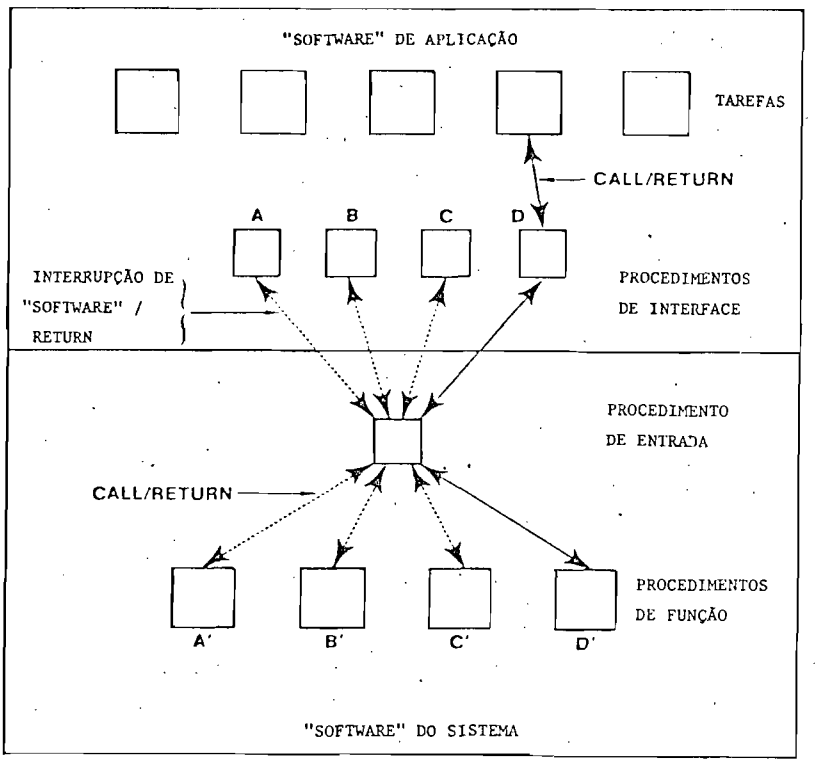


Fig. 3.7 - Extensão do Sistema Operacional com Procedimento de Entrada.

O *Procedimento de Interface* tem o propósito básico de usar uma interrupção de "software" para transferir o controle da tarefa que fez a chamada ao sistema para um *Procedimento de Entrada* (ou na sua ausência, para um *Procedimento de Função*). A estruturação destes procedimentos pode ser vista na Figura 3.7.

Caso haja um Procedimento de Entrada, o Procedimento de Interface deve comunicar-lhe o código que identifica o Procedimento de Função que ele deverá chamar. O Procedimento de Interface faz isto carregando o código num registro previamente determinado ou na pilha da tarefa invocadora.

Assim, o Procedimento de Entrada, quando invocado, extrai o código deste registro ou da pilha.

Uma segunda função do *Procedimento de Interface* é a de informar a tarefa invocadora (ou seu manipulador de exceção) sobre a ocorrência de alguma condição excepcional. Isto é feito da seguinte forma: o Procedimento de Entrada (ou o Procedimento de Função se não existir o Procedimento de Entrada) comunica a exceção encontrada para o Procedimento de Interface que, por sua vez desvia para o Procedimento RQ$ERROR "default" (desvio 4 na Figura 3.8), contido na biblioteca da interface do núcleo.

O *Procedimento de Entrada* tem o propósito de rotear a chamada para um Procedimento de Função. Além disso, deve avisar o mecanismo de manipulação de exceção no caso de ocorrência de um erro. Este aviso pode ser feito de duas maneiras, dependendo se a Extensão do Sistema Operacional tem seu próprio manipulador de exceção ou se a tarefa deseja manipular a exceção. No segundo caso, um *Procedimento RQ$ERROR* deve estar associado à tarefa. Este procedimento deve ser ligado ao Procedimento de Entrada.

O *Procedimento de Função* tem por função principal executar as ações requisitadas pela tarefa invocadora. Caso não haja um Procedimento de Entrada, o Procedimento de Função deverá informar ao Procedimento de Interface o estado de execução da tarefa invocadora.

Finalmente o *Procedimento RQ$ERROR*, referido pelos Procedimentos de Interface e Entrada, é invocado pelos Procedimentos de Interface do núcleo e pelos subsistemas do Sistema Operacional no caso da ocorrência de uma condição excepcional. Por exemplo, se a tarefa da aplicação faz uma chamada ao sistema, SEND$MESSAGE, e resulta uma condição excepcional, o NÚCLEO retorna o erro para a biblioteca de interface do núcleo que está ligada à tarefa da aplicação. O procedimento na biblioteca então chama RQ$ERROR para processar o erro.

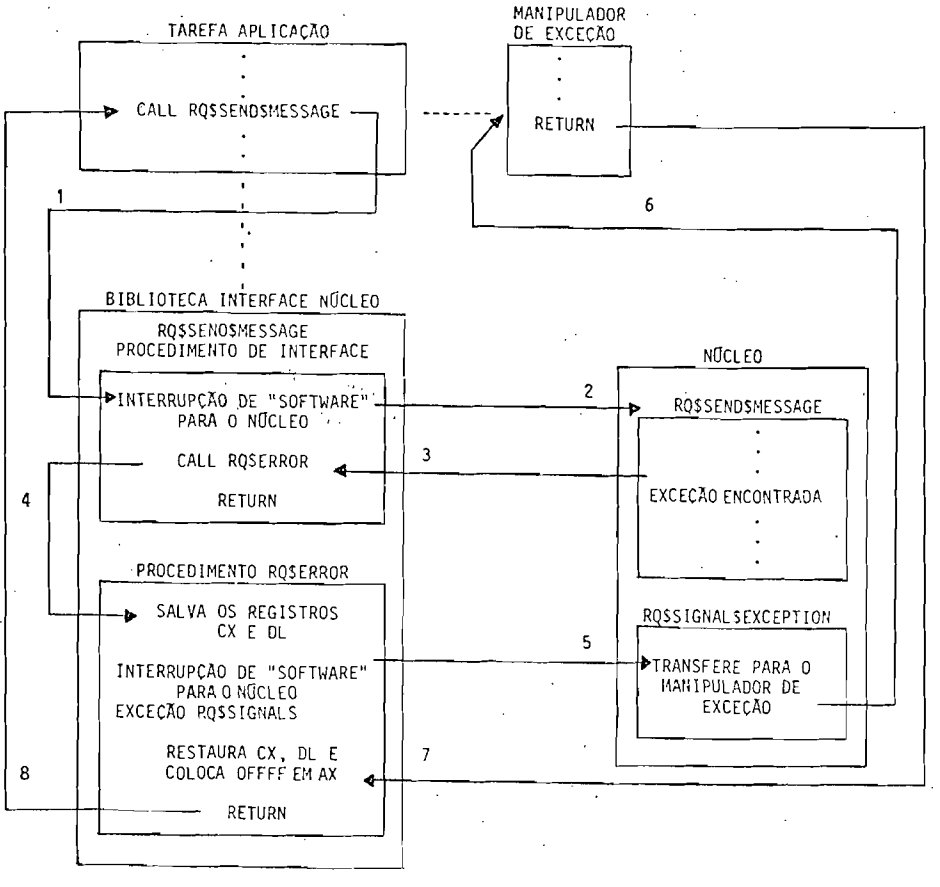


Fig. 3.8 - Fluxo de controle para um manipulador de exceção.

Este mecanismo também é válido para os demais subsistemas que compõem a arquitetura do Sistema Operacional iRMX86 (tais como SISTEMA DE ENTRADA/SAÍDA) e para *extensões do Sistema Operacional* que fazem chamadas ao sistema. Por exemplo, se o SISTEMA DE ENTRADA/SAÍDA chama SEND$MESSAGE e resulta uma condição excepcional, o NÚCLEO retorna o erro da sua biblioteca de interface associada ao SISTEMA DE ENTRADA/SAÍDA.

Como já foi citado, a biblioteca de interface do núcleo contém um procedimento RQ$ERROR "default", cuja função é chamar um SIGNAL$EXCEPTION para informar a tarefa invocadora (ou seu manipulador de exceção) da exceção ocorrida. Esta versão do procedimento RQ$ERROR

deveria estar ligada às tarefas da aplicação para garantir que seus manipuladores de exceções sejam chamados na ocorrência de condições excepcionais.

A Figura 3.8 ilustra o fluxo de controle de uma tarefa da aplicação para um manipulador de exceção quando a tarefa se encontra numa condição excepcional.

O Sistema Operacional iRMX86 utiliza este mecanismo de retorno de exceções para dar aos subsistemas e extensões do Sistema Operacional flexibilidade na manipulação de suas próprias exceções. Caso eles desejam que suas condições de exceção sejam processadas de uma forma especial, uma versão própria do procedimento RQ$ERROR pode ser provida para cuidar deste processamento especial.

Desta forma, como criador de uma Extensão do Sistema Operacional, tem-se a opção de ligar estas extensões tanto a um procedimento RQ$ERROR "default" quanto a um procedimento RQ$ERROR próprio.

Quanto à *inicialização do vetor de interrupção* deve-se citar que antes que o procedimento de interface transfira o controle com sucesso para uma Extensão do Sistema Operacional, o vetor de interrupção deve ser inicializado com os endereços dos Procedimentos de Entrada (e de Função). A chamada ao sistema, do tipo SET$OS$EXTENSION, tem esta finalidade de inicialização.

Como citado anteriormente, um objeto de qualquer tipo provido pelo iRMX86 pode ser descartado através da chamada ao sistema DELETE$objeto. Entretanto extensões do Sistema Operacional podem usar a chamada ao sistema DISABLE$DELETION para tornar um objeto imune a este tipo de descarte. Uma chamada ENABLE$DELETION subseqüente remove esta imunidade.

# CAPÍTULO 4

## SISTEMA DE SUPERVISÃO TOLERANTE A FALHAS

Objetivando prover o Sistema de Supervisão até aqui apresentado, de tolerância a falhas, propõe-se neste capítulo a utilização de algumas técnicas de deteção e recuperação de erro já mencionados no Capítulo 2. Estas técnicas serão incorporadas ao sistema como extensão do Sistema Operacional iRMX86 e deverão ser apoiadas por recursos específicos de "hardware" também propostosno decorrer deste capítulo.

### 4.1 - TÉCNICAS ADOTADAS PARA TOLERÂNCIA A FALHAS

Para atender os objetivos do Sistema de Supervisão apresentados na Seção 3.2 de forma a prover este sistema da capacidade de tolerar falhas, foram selecionadas técnicas não-dependentes da aplicação. As técnicas adotadas baseiam-se exclusivamente na arquitetura mostrada na Seção 3.2 e no Sistema Operacional iRMX86 considerado.

#### 4.1.1 - SERVIÇO "HOT-STANDBY" SIMPLIFICADO

Com a arquitetura e a filosofia de operação descrita na Seção 3.2, o sistema pode prover um serviço implementado por duas *instâncias idênticas* de cada tarefa, uma em cada unidade de processamento. Ainda que tais instâncias estejam preparadas para executar o processamento da aplicação, num dado instante, apenas uma delas está autorizada para processar mensagens e cooperar com outras tarefas do sistema. Esta instância é denominada INSTÂNCIA ATIVA. A outra, a INSTÂNCIA PASSIVA, não executa nenhum processamento da aplicação. Sua função fica limitada a uma cópia atualizada da informação de estado de execução (situação) da tarefa, produzida pela INSTÂNCIA ATIVA. Loques (1984a) chama este *serviço de "hot-standby"*.

No caso do Sistema de Supervisão sendo considerado neste trabalho, por serem os recursos memória compartilhada e disco do tipo

portas duais, dispensam-se quaisquer atividades de uma INSTÂNCIA PASSIVA. Não é necessário que ela execute nem mesmo a função de manter uma cópia atualizada da situação da tarefa, pois os mecanismos de apoio a recuperação de erro podem executar esta função. O único requisito é que estas informações sejam armazenadas numa *memória estável* (Seção 4.2.1) acessável pelas duas INSTÂNCIAS da tarefa (ATIVA e PASSIVA).

Todas as INSTÂNCIAS ATIVAS das tarefas de supervisão devem ser alocadas na UNIDADE ATIVA da arquitetura básica, enquanto as INSTÂNCIAS PASSIVAS devem ficar alocadas, como reserva, na UNIDADE PASSIVA. Cabe lembrar que a alocação das INSTÂNCIAS ATIVA E PASSIVA de uma tarefa deve ser feita na inicialização do Sistema de Supervisão, de forma idêntica nas duas unidades de processamento (mesmo endereçamento absoluto).

Como todas as INSTÂNCIAS PASSIVAS ficam alocadas numa mesma UNIDADE (PASSIVA), então poder-se-ia concluir que esta UNIDADE poderia ficar desligada. Entretanto, é conveniente que a UNIDADE PASSIVA permaneça o tempo todo pronta para assumir a operação da UNIDADE ATIVA, até mesmo com o código das INSTÂNCIAS já carregados e inicializados na sua memória interna. Além do mais, esta UNIDADE deve executar o processamento necessário para o envio da sinalização "EU ESTOU BEM" para a UNIDADE ATIVA ("reset" CÃO DE GUARDA, apresentado em maiores detalhes no decorrer deste capítulo). Assim sendo, considera-se este *serviço "HOT-STANDBY" simplificado*.

No caso de falha da UNIDADE ATIVA, todas as INSTÂNCIAS PASSIVAS são tornadas ATIVAS e o processamento é continuado pela UNIDADE PASSIVA que, a partir de então, tornar-se ATIVA.

Caso a falha ocorra na UNIDADE PASSIVA, as INSTÂNCIAS da UNIDADE ATIVA não são afetadas.

Após a recuperação de uma UNIDADE com defeito, ela deve ser reincorporada ao sistema. Isto implica a reinicialização das INSTÂNCIAS

PASSIVAS das tarefas de supervisão nesta UNIDADE, o que deve ser feito independentemente da operação das INSTÂNCIAS ATIVAS.

### 4.1.2 - CONFINAMENTO DE ERRO

As condições excepcionais consideradas neste trabalho são basicamente aquelas detetáveis pelo Sistema Operacional iRMX86 (Seção 3.3.2). O Sistema Operacional caracteriza-se como uma *interface interpretativa* que implementa ferramentas, isto é, *mecanismos* para apoiar a detecção de erros pelos programas em execução (Seção 2.5). No caso, o Sistema Operacional iRMX86 apoia testes de consistência e testes de tempo.

Quanto à manipulação destas exceções considera-se aqui que qualquer erro sinalizado pelo Sistema Operacional durante a execução da aplicação de supervisão deve ser manipulado como uma exceção que implica *parar o processador*. Portanto, propõe-se que quando a INSTÂNCIA ATIVA de uma tarefa detetar uma falha, a manipulação da exceção deve cessar não somente as atividades da INSTÂNCIA ATIVA daquela tarefa, mas também todas atividades da UNIDADE. Esta técnica impede que a UNIDADE ATIVA prossiga a execução errônea.

Vê-se claramente que esta técnica é totalmente dependente dos mecanismos de detecção de erro providos pela Interface Interpretativa. Porém, *asserções* (Seção 2.5) introduzidas no "software" (no caso, nas próprias INSTÂNCIAS ATIVAS do Sistema de Supervisão) podem ser utilizadas para detetar erros específicos da aplicação e até mesmo contorná-los sem ter que parar a UNIDADE ATIVA. É o caso, por exemplo, de tarefas críticas de *atuação*. Elas podem implementar *testes de consistência* através de asserções antes de efetuarem a atuação. Caso estas tarefas detetem algum *erro previsto*, o manipulador de exceções associado poderia fazer um ajuste nesta saída antes que a tarefa efetive a atuação. Recentemente asserções foram propostas como uma maneira de confinar erros dentro de uma tarefa em sistemas que se comunicam por troca de mensagens em aplicações de controle de processos (Kopetz, 1982). Nesta proposta, uma asserção é avaliada justamente antes que uma mensagem seja enviada pela tarefa. Se um er

ro for detetado, ou o manipulador de exceção associado contorna o problema (erro previsto) ou pára a execução da tarefa.

Com a adoção desta técnica para Confinamento de Erro, tem-se a vantagem de não haver dependência de apoio de "hardware" para a detecção de erros. A própria interface interpretativa, no caso o Sistema Operacional iRMX86, pode se encarregar de algumas destas deteções, especificadas na Tabela 3.3, e apoiar várias outras, através do desenvolvimento de asserções e expansão do sistema. Porém, nada impede que interfaces de aquisição e atuação sejam implementadas com circuitos específicos para deteção de erro. Neste caso, o único requisito feito é que a manipulação do erro, quer por "hardware" quer por "software" provoque uma parada na UNIDADE ATIVA.

Cabe lembrar (Seção 3.2.1) que a falha de um processador é sentida pelo outro por ocasião de ausência do sinal enviado periodicamente de um processador para o outro, por uma linha de controle específica do barramento que os une. Caso este sinal não seja recebido por uma unidade, então o CÃO DE GUARDA daquela unidade, que é um *mecanismo da interface interpretativa*, sinalizará uma exceção especial. A manipulação desta exceção deverá então reconfigurar o sistema de modo que a unidade sem defeito possa dar continuidade a operação de supervisão, utilizando para isto as INSTÂNCIAS PASSIVAS das tarefas que estavam sendo executadas naquela unidade que falhou.

A técnica Confinamento de Erro proposta visa simplicidade de implementação e de análise. Com esta técnica tem-se que: qualquer falha detetada pela INSTÂNCIA ATIVA de uma tarefa desvia o processador para um manipulador de exceção que parará completamente todas as atividades da UNIDADE ATIVA, inclusive a geração do sinal "EU ESTOU BEM" enviado para a UNIDADE PASSIVA. A ausência deste sinal faz com que a UNIDADE PASSIVA detete tal situação e tome as devidas providências.

Falhas não-detetadas pela INSTÂNCIA ATIVA, tipo falhas transitórias, por exemplo, problemas na geração do sinal "EU ESTOU BEM",

poderiam provocar uma operação paralela das atividades de supervisão nas duas UNIDADES. Para garantir que a UNIDADE com falha não funcionará em paralelo com a outra UNIDADE, a sinalização feita pelo CÃO DE GUARDA, além de desencadear o processo de recuperação do sistema, deve, assim que ocorrida, desativar a UNIDADE com falha. Isto é bastante simples e pode ser feito através de uma linha de controle do barramento que une as 2 UNIDADES (sinal "HOLD" na UCP da UNIDADE com defeito).

Em termos de Confinamento de Erro, uma alternativa mais abrangente do que a adotada seria considerar que um defeito pudesse afetar somente algumas INSTÂNCIAS ATIVAS, não necessariamente todas executadas pela UNIDADE ATIVA. Isto é, aquelas INSTÂNCIAS ATIVAS que, de alguma forma, estivessem envolvidas com aquele tipo de defeito. Neste caso, apenas as tarefas afetadas seriam substituídas pelas suas respectivas INSTÂNCIAS PASSIVAS, residentes na UNIDADE PASSIVA. Isto implicaria processamento distribuído nas duas UNIDADES, o que foge do escorpo deste trabalho, visto que o Sistema Operacional iRMX86 não permite multiprocessamento. Além do mais seria necessário um controle bastante complexo que exigiria um conhecimento detalhado das tarefas afetadas, para cada tipo de defeito.

### 4.2.3 - TÉCNICA DE RECUPERAÇÃO

A técnica de recuperação de erro adotada para tornar este Sistema de Supervisão tolerante a falhas é a *Recuperação Retroativa*. Ela depende da provisão de *pontos de recuperação* (Seção 2.6) para preservar as informações suficientes sobre a situação de um sistema, de modo que ele possa ser restaurado depois da ocorrência de uma falha.

Para implementação desta técnica será utilizada a estratégia *ponto de teste ("checkpoint")* (Seção 2.6) para registrar e preservar dados nos pontos de recurperação.

De acordo com Loques (1984a), em sistemas de um único processador esta estratégia tem sido muito usada pelo próprio sistema para prover tolerância a falhas. No entanto, em sistemas distribuídos, os *"checkpoints"* são caracterizados nas tarefas em lugares relevantes de suas operações. Porém, a implementação da operação de "checkpoint" deve ser independente da tarefa que a invocou. Portanto, deve existir um mecanismo que apoia esta operação.

Na realidade a técnica *recuperação retroativa* foi proposta devido a sua capacidade de: (1) fornecer recuperação diante de defeitos não-previstos; (2) independência da aplicação e; (3) principalmente, pela facilidade de ser implementada como um mecanismo de recuperação de erro.

Com relação ao estabelecimento de pontos de recuperação, a forma proposta foi considerá-los nos aplicativos do sistema (Seção 2.6). Para tanto, a interface interpretativa, isto é, o Sistema Operacional deverá ser especialmente dotado para prover o programador da aplicação com uma chamada ao sistema, denominada "SAVE" específica para o estabelecimento de um ponto de recuperação. Maiores detalhes da primitiva "SAVE" serão dados na Seção 4.2.1.2.

Com a utilização da técnica *confinamento de erro* a implementação de recuperação etroativa torna-se consideravelmente simples.

### 4.1.4 - PROBLEMAS DE CONSISTÊNCIA

É importante ressaltar, no entanto, que dependendo da forma adotada para a implementação dos mecanismos de recuperação de erro, problemas de consistência poderão surgir. Assim sendo, se faz necessária uma escolha criteriosa destes mecanismos visando garantir a *consistência da recuperação* e, portanto, evitar os seguintes problemas:

(I) *Efeito dominó* - Este efeito é devido ao estabelecimento desordenado de pontos de recuperação (Seção 2.6.2). Quando um erro é detetado, as informações armazenadas no ponto de teste ("checkpoint") de cada tarefa devem ser suficientes para o reestabelecimento do "checkpoint". Isto se faz necessário para colocar o sistema de volta a um *estado consistente* que existiu quando o sistema estava funcionando corretamente.

(II) *Repetição de entrada* - Na busca do estado consistente, através da reexecução das tarefas, não é garantido que os dados provenientes do sistema, dados que chegam na interface da tarefa com o sistema, sejam os mesmos, repetidamente. Isto implica restrições na estrutura e na comunicação entre tarefas.

(III) *Geração de saídas* - A propagação do erro além da interface da tarefa com o sistema, resultados errôneos provenientes da tarefa, também é um problema de consistência. Pois, se o erro só for detetado depois de a saída ter sido liberada ela não poderá ser cancelada.

(IV) *Transferência para a memória estável* - Preocupa-se com a consistência da recuperação, caso um erro ocorra durante a transferência da situação da tarefa para a memória estável. É preciso garantir que se um erro ocorrer durante a atualização da situação, esta atualização será desconsiderada permanecendo a situação anterior.

Ainda, com relação a consistência, a técnica para confinamento de erro proposta garante que a falha de um tarefa é vista consistentemente pelas outras tarefas do sistema, o que significa que a falha de uma tarefa não causa falha de outras através da propagação e erro. A validade desta afirmação está diretamente vinculada à eficiência do processo de deteção de erro e conseqüente parada da UNIDADE ATIVA.

## 4.2 - EXTENSÃO DO iRMX86 PARA APOIAR TOLERÂNCIA A FALHAS

Considerando que o Sistema de Supervisão é constituído por tarefas concorrentes que implementam atividades de aquisição, processamento e atuação, e que o Sistema Operacional adotado para apoiar a execução destas tarefas é o iRMX86, a forma escolhida para dotar a supervisão de tolerância a falhas consite em *expandir o iRMX86* com recursos que possibilitam tornar as *tarefas confiáveis* e a *recuperação* do sistema *transparente* à aplicação. Para tanto, propõe-se a incorporação ao Sistema Operacional iRMX86 de mecanismos de tolerância a falhas para apoio às tarefas de supervisão. Isto é possível dada a facilidade de expansão provida pelo iRMX86 (Seção 3.3.2), que permite a definição e a adição de novos objetos e chamadas ao sistema (também referenciadas neste trabalho por primitivas).

Os mecanismos propostos para apoio a tolerância a falhas constituem duas novas camadas que devem ser incorporadas na Arquitetura do Sistema Operacional iRMX86, apresentada na Figura 3.2. Estas camadas são denominadas:

- RECURSO PARA TOLERÂNCIA A FALHAS e

- GERENTE DE RECUPERAÇÃO.

A Figura 4.1 apresenta a arquitetura do Sistema Operacional iRMX86 estendida para tolerância a falhas. A camada RECURSO PARA TOLERÂNCIA A FALHAS constitui uma extensão do NÚCLEO do iRMX86 e implementa mecanismos de apoio à recuperação de erro.

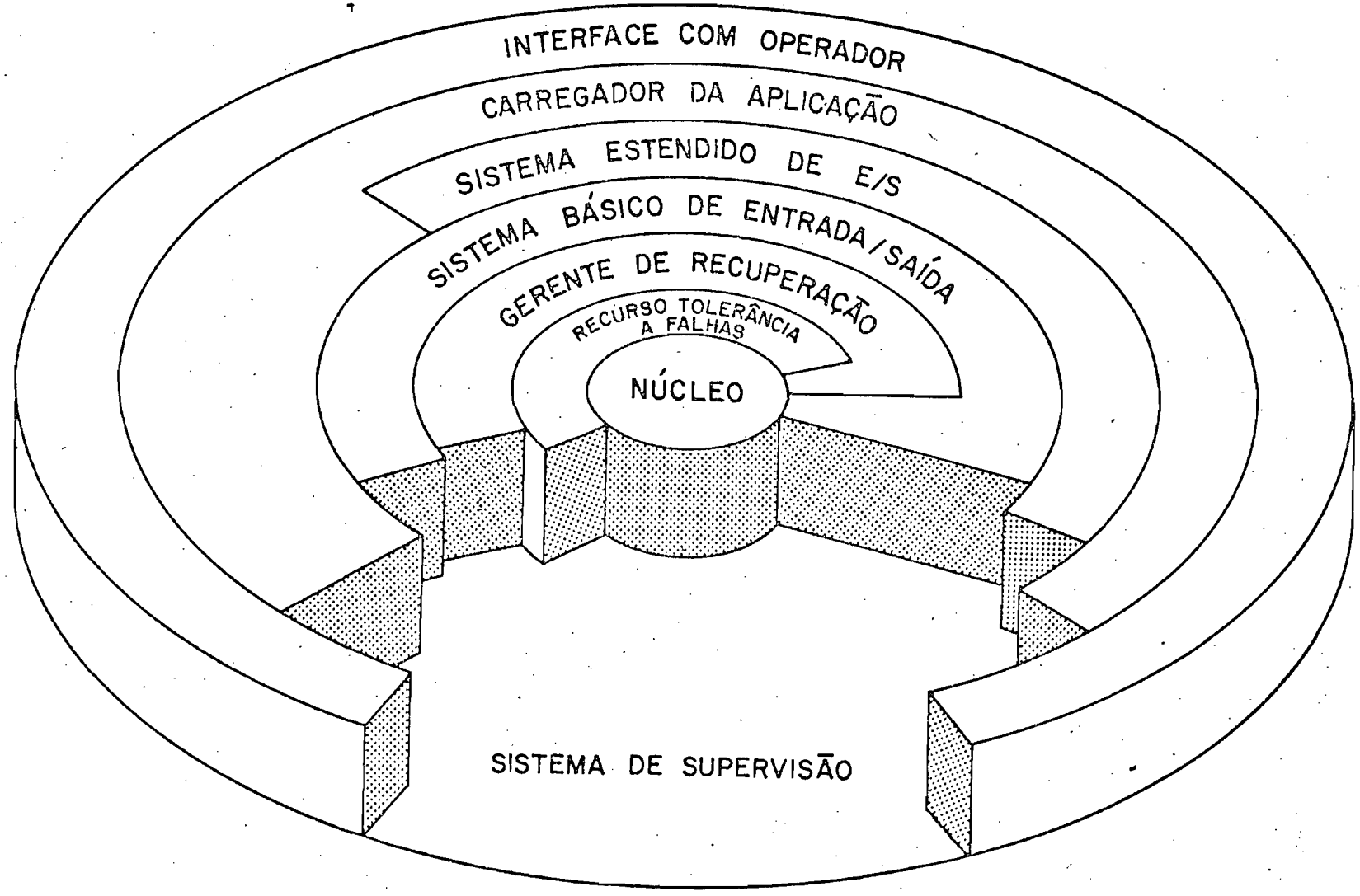


Fig. 4.1 - Arquitetura do Sistema Operacional iRMX86 estendida para tolerância a falhas.

Neste contexto, o GERENTE DE RECUPERAÇÃO constitui a camada responsável pela realização de *recuperação transparente* às atividades da aplicação, permitindo que o sistema seja recuperado de falhas simples sem que as atividades de supervisão tenham de ser reinicializadas. Para isto, o GERENTE DE RECUPERAÇÃO utiliza os serviços oferecidos pelo NÚCLEO do iRMX86 e as facilidades providas pelo RECURSO PARA TOLERÂNCIA A FALHAS.

Convêm ressaltar que o Sistema Operacional iRMX86 por ser padrão, disponível no mercado, é um sistema fechado a modificações. Isto significa que não se tem acesso a sua estrutura de dados e código. Assim, os mecanismos de tolerância a falhas propostos devem utilizar apenas as formas permitidas pelo iRMX86 para serem a ele incorporados.

### 4.2.1 - RECURSO PARA TOLERÂNCIA A FALHAS

Esta camada utiliza de forma simplificada o conceito de *memória estável* introduzido por Loques (1984a).

#### 4.2.1.1 - MEMÓRIA ESTÁVEL

O conceito de memória estável envolve a especificação de um recurso (memória compatilhada disponível na Arquitetura Básica do Sistema de Supervisão - Seção 3.2) e de um mecanismo (primitiva "SAVE) para apoiar a recuperação de erro retroativa adotada.

O recurso *memória compartilhada* deve manter as informações da situação do Sistema de Supervisão atualizadas nos pontos de recuperação estabelecidos pela aplicação, através da invocação do "SAVE" incorporada ao Sistema Operacional iRMX86 como uma nova chamada ao sistema (primitiva).

Propõem-se duas ALTERNATIVAS para implementação da memória estável. A primeira delas, ALTERNATIVA 1, caracteriza-se pela *transferência total* do estado do sistema a cada operação "SAVE", implementando as

sim a estratégia ponto de teste ("checkpoint"), definida na Seção 2.6, para preservar dados nos pontos de recuperação. A segunda, ALTERNATVA 2, restringe-se a uma *transferência parcial* do estado do sistema na operação "SAVE", implementando a estratégia "recovery cache" (Seção 2.6).

ALTERNATIVA 1 - "CHECKPOINT"

A *memória estável* é constituída por dois MÓDULOS de memórias RAM do tipo portas duais, providos pela memória compatilhada, os quais devem possuir tamanho idêntico ao das memórias internas (memórias principais) das UNIDADES ATIVA e PASSIVA. A Figura 4.2 apresenta a idéia da memória estável proposta nesta alternativa. Nesta figura cada MÓDULO é denominado MEMRAM.

A lógica de controle desta memória estável deve implementar a seguinte filosofia de operação: um MÓDULO (MEMRAM) da Memória Estável pode se encontrar em apenas um de dois estados possíveis: CORRENTE ou ÚLTIMO.

Diz-se que um MÓDULO é CORRENTE quando ele estiver disponível à UNIDADE ATIVA para escrita e inacessível pela UNIDADE PASSIVA. Por outro lado, ele se encontra no estado ÚLTIMO quando estiver inacessível pela UNIDADE ATIVA e disponível à PASSIVA apenas para leitura.

É importante salientar que quando um MÓDULO MEMRAM estiver no estado CORRENTE o outro necessariamente se encontra no estado ÚLTIMO e vice versa. Assim sendo, enquanto um MÓDULO (CORRENTE) estiver disponível à UNIDADE ATIVA para escrita, ele não pode ser lido pela UNIDADE PASSIVA. Se ocorrer uma falha na UNIDADE ATIVA, a UNIDADE PASSIVA utiliza as informações contidas no outro MÓDULO (ÚLTIMO) para continuar a operação do sistema.

Toda escrita na memória interna da UNIDADE ATIVA é ao mesmo tempo feita no MÓDULO (MEMRAM) da Memória Estável que se encontra no estado CORRENTE.

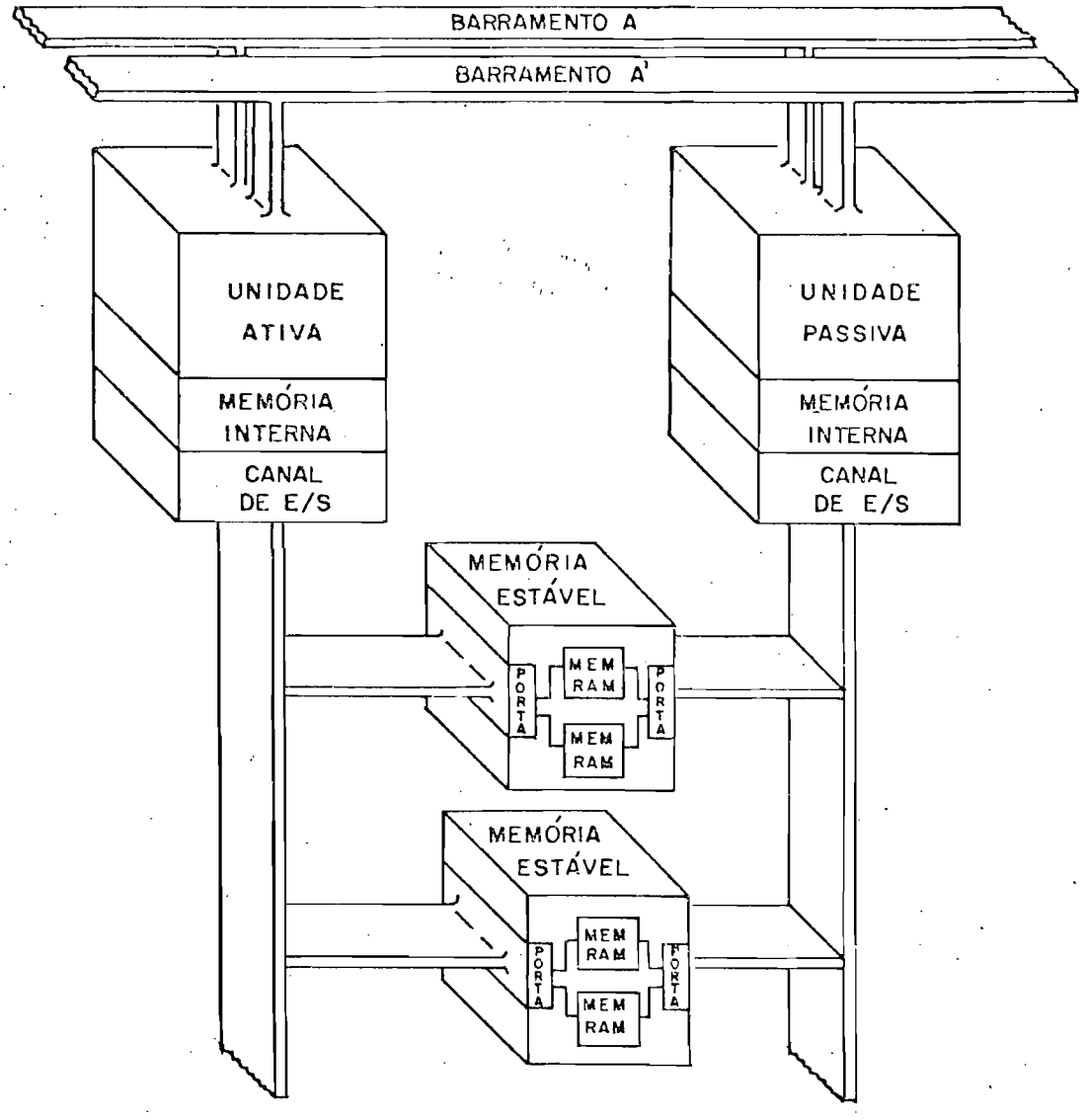


Fig. 4.2 - Memória Estável - ALTERNATIVA 1

Toda leitura de memória realizada pela UNIDADE ATIVA é feita apenas sob sua memória interna (memória principal da UNIDADE ATIVA).

O chaveamento de estado dos MÓDULOS da Memória Estável é estimulado por um sinal *"OUT na porta da memória estável"* conectada à UNIDADE ATIVA. A Figura 4.3 apresenta um esquema do circuito de controle para cheveamento dos dois MÓDULOS (MEMRAMs) da Memóra Estável.

A cada sinal "OUT", enviado pela UNIDADE ATIVA via linha de controle OUT1 da Figura 4.3, o antigo MÓDULO (MEMRAM) que estava no estado CORRENTE é chaveado para o estado ÚLTIMO cujo acesso é permitido apenas para leitura da UNIDADE PASSIVA. Simultaneamente, o outro MÓDULO, antigo ÚLTIMO, é chaveado para o estado CORRENTE e deverá ser utilizado, a partir de então, para escrita pela UNIDADE ATIVA.

Seguindo o chaveamento de estado dos MÓDULOS da Memória Estável, é então executada uma cópia das áreas de dados da Memória Interna da UNIDADE ATIVA para o atual MÓDULO CORRENTE da Memória Estável. Esta transferência de dados é feita para tornar o antigo MÓDULO ÚLTIMO atualizado.

Para permitir a recuperação retroativa do sistema, em caso de falha, a Memória Estável deve conter no mínimo:

- toda a *área de dados do NÚCLEO do Sistema operacional iRMX86* adotado (configurado nas posições 1533H e 15FEH como é mostrado no Apêndice B);

- toda a *área de dados das INSTÂNCIAS ATIVAS* (alocadas a partir da posição 1600H). Estas áreas podem ser especificadas pela aplicação se a alocação das tarefas na memória for estática e feita pelo usuário na inicialização do sistema, através da criação das áreas e dos "JOBs" aos quais elas pertencem (Seção 3.3.2).

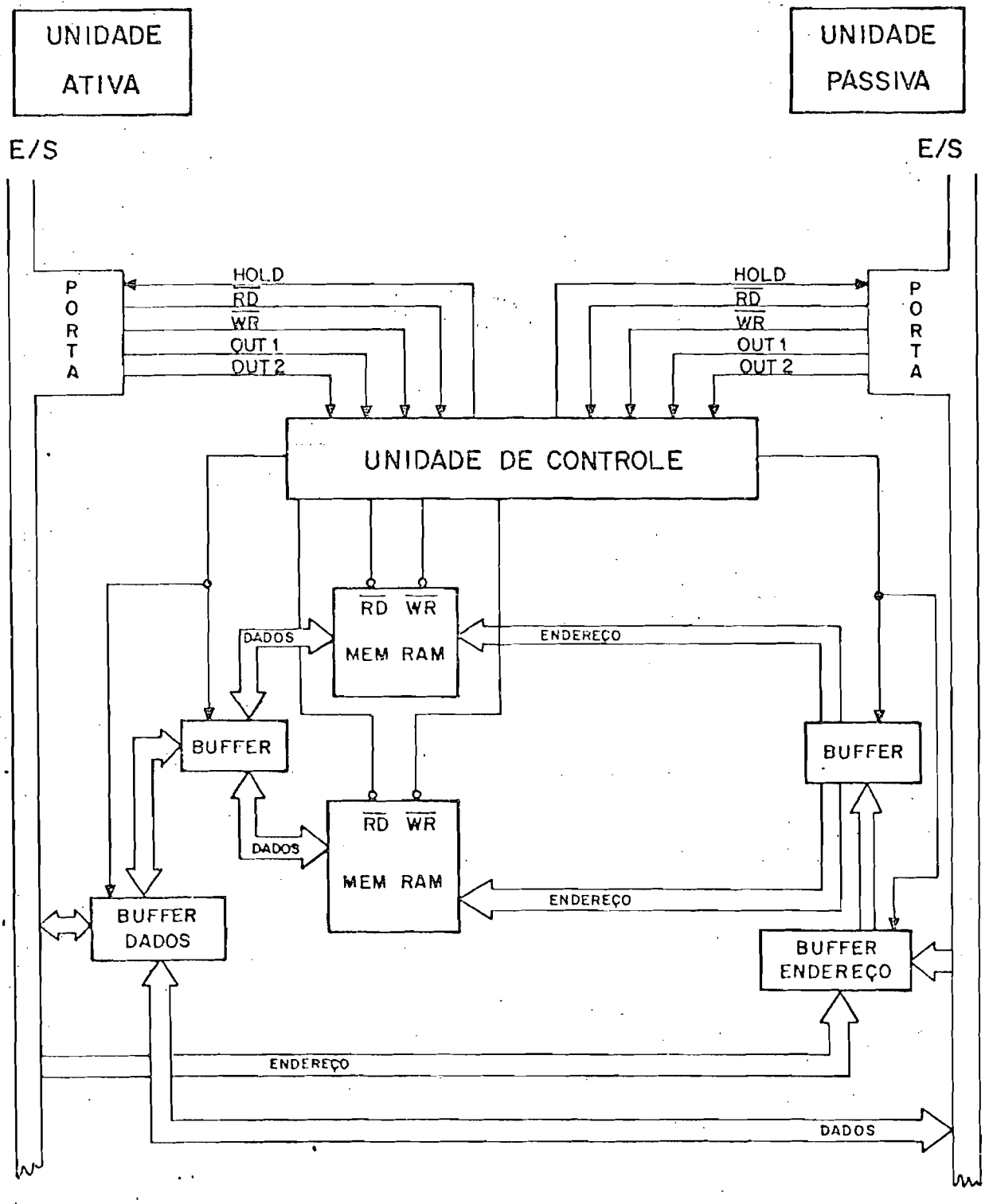


Fig. 4.3 - Esquema do circuito de controle da Memória Estável - ALTERNATIVA 1.

Os endereços absolutos das áreas a serem transferidas devem constar de uma *TABELA DE TRANSFERÊNCIA* criada automaticamente na inicialização do sistema quando ocorre a alocação estática dos "JOBs" e TAREFAS da aplicação (vide Apêndice C).

A construção da TABELA DE TRANSFERÊNCIA é feita com a invocação de duas novas chamadas ao sistema, incorporadas ao iRMX86 como um mecanismo de apoio à tolerância a falhas. São elas:

CRIAJOB - Tem a função de criar um "JOB" numa área de memória especificada pelo usuário. É uma extensão da primitiva CREATE $JOB provida pelo iRMX86.

CRIATAREFA - Tem a função de criar uma TAREFA numa área de memória especificada pelo usuário. É uma extensão da primitiva CREATE$TASK provida pelo iRMX86.

No processo de inicialização do sistema estas *novas chamadas* devem ser utilizadas pelo usuário como substitutas de CREATE$JOB e CREATE$TASK, respectivamente, para que a TABELA DE TRANSFERÊNCIA possa ser construída. Estas novas chamadas ao sistema estão descritas no Apêndice C, assim como a TABELA DE TRANSFERÊNCIA.

Após a inicialização do sistema a TABELA DE TRANSFERÊNCIA não deve ser alterada.

O conteúdo dos endereços especificados nesta tabela são copiados da memória interna da UNIDADE ATIVA para a MEMÓRIA ESTÁVEL, a cada ponto de recuperação (invocação da primitiva "SAVE") definido pela INSTÂNCIA ATIVA em execução.

*Primitiva "SAVE"*

Nesta ALTERNATIVA 1 o "SAVE" tem a função de transferir para a MEMÓRIA ESTÁVEL a área de dados do NÚCLEO do Sistema Operacional iRMX86 e as áreas de dados de cada INSTÂNCIA ATIVA.

O "SAVE" é adicionado ao Sistema Operacional iRMX86 como uma nova chamada ao sistema (primitiva). Assim, a cada invocação desta primitiva por uma INSTÂNCIA ATIVA (estabelecimento de um ponto de recuperação) não só a área de dados da INSTÂNCIA que a invocou será transferida para a Memória Estável, mas também a área de dados de todas as outras INSTÂNCIAS da UNIDADE ATIVA. Isto se deve ao Sistema Operacional adotado (iRMX86) ser fechado a modificações do usuário.

Nota-se, por exemplo, o que acontece se o "SAVE" invocado por uma INSTÂNCIA ATIVA transferisse para a Memória Estável apenas a área de dados daquela tarefa. Suponha-se que a INSTÂNCIA ATIVA A invocou um "SAVE" e então continuou a utilizar o processador pedindo ao NÚCLEO o uso de um semáforo, o qual estava disponível e foi entregue a A. Então, logo a seguir, A perdeu o controle do processador que foi entregue à INSTÂNCIA ATIVA B. Esta por sua vez, invocou um "SAVE" e continuou sua execução até que foi detetada uma falha (Figura 4.4).

Neste caso a recuperação da falha deverá fazer a INSTÂNCIA A retornar ao instante t e a B ao instante t+1. Isto não teria problema algum se A não tivesse invocado o NÚCLEO do Sistema Operacional ( pedido do semáforo) cuja estrutura de dados interna foi modificada (valor do semáforo alterado), e se o "SAVE" dado por B, em t+1, não tivesse salvo este novo estado do NÚCLEO (semáforo alterado). Porém, após estes fatos tem-se que na recuperação a INSTÂNCIA A deve retornar com sua área de dados compatível com o instante t, entretanto com a área de dados do NÚCLEO em t+1 (semáforo já entregue a A e ainda não devolvido por esta INSTÂNCIA). Quando A é executada novamente, ela deve repetir o pedido do semáforo que então nunca lhe será dado, visto que ele está preso por uma A antiga que nunca o liberará ("deadlock").

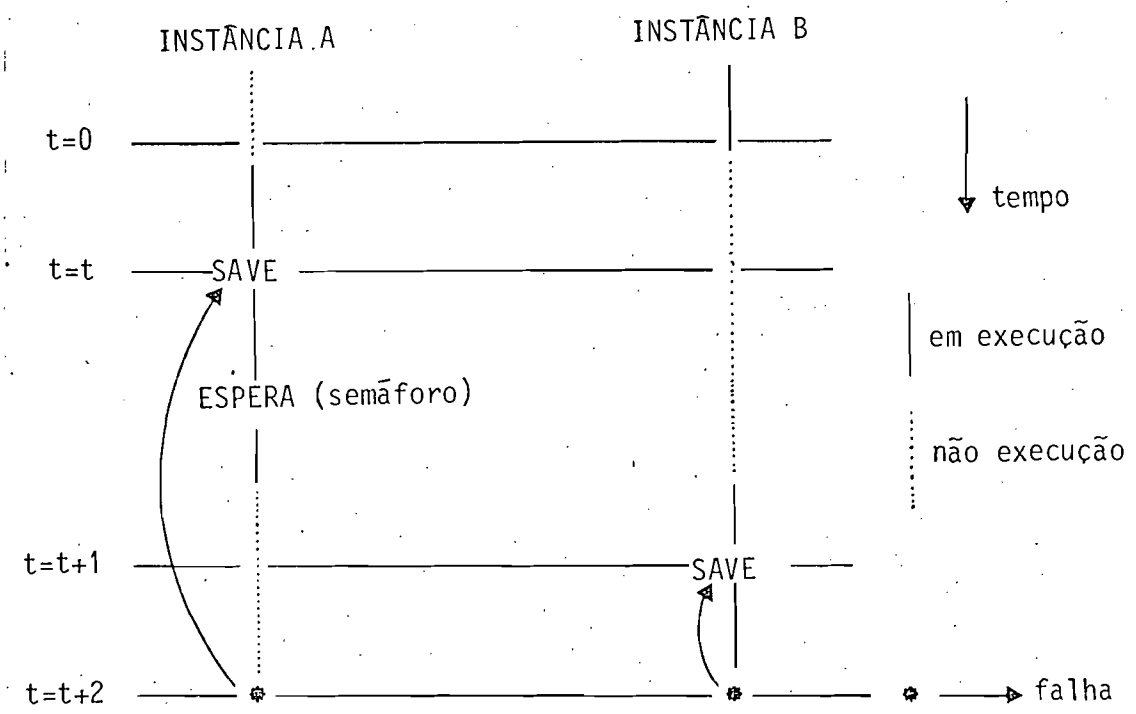


Fig. 4.4 - Um exemplo de aplicação do "SAVE".

Esta situação torna a recuperação inconsistente. Considerando que o NÚCLEO do iRMX86 não é aberto a modificações e que nada se tem a fazer neste sentido, a solução encontrada foi: o "SAVE" invocado por uma INSTÂNCIA ATIVA transfere não somente a área de dados do NÚCLEO do iRMX86 e a área de dados desta INSTÂNCIA, mas também a área de dados das outras INSTÂNCIAS da UNIDADE ATIVA. Isto é, o "SAVE" invocado por uma INSTÂNCIA ATIVA causa um *SAVE VIRTUAL* nas outras INSTÂNCIAS da UNIDADE".

Assim, cabe à função "SAVE":

1 - executar o chaveamento dos MÓDULOS (MEMRAMs) da Memória Estável de modo que o CORRENTE passe a ser ÚLTIMO e este torne-se CORRENTE. Este chaveamento é desencadeado por um sinal *"OUT porta memória estável"* pela linha de controle OUT1 da Figura 4.3;

2 - efetuar a transferência das áreas de dados acima descritas para o MÓDULO CORRENTE, utilizando para isto a TABELA DE TRANSFERÊNCIA. Esta transferência pode ser executada por "software" ou por um controlador de Acesso Direto à Memória (ADM), com a cópia da memória interna da UNIDADE ATIVA no MÓDULO CORRENTE.

No caso de a transferência ser feita por ADM, após o passo 1, o processador que enviou o "OUT" deve ficar suspenso (entrar no estado "HOLD"). Para que isto ocorra a linha de controle "HOLD" da Figura 4.3 deve ser ativada. No caso de a transferência ser por "software", esta linha de controle não é necessária.

Convém notar que o chaveamento dos MÓDULOS da Memória Estável realizado antes de executar a transferência (passo 1), visa garantir a *atomicidade* desta função. Isto é, se uma falha ocorrer durante a transferência, o sistema é retornado pela recuperação a um estado anterior, obtido daquele MÓDULO que se encontra no estado ÚLTIMO. Desta forma, para o sistema, tudo se passa como se a operação "SAVE" que falhou nem tivesse sido invocada.

ALTERNATIVA 2 - "RECOVERY CACHE"

Nesta ALTERNATIVA, a *memória estável* é análoga à proposta na ALTERNATIVA 1 (Figura 4.2), porém é adicionado à memória compartilhada uma *pilha* (Figura 4.5).

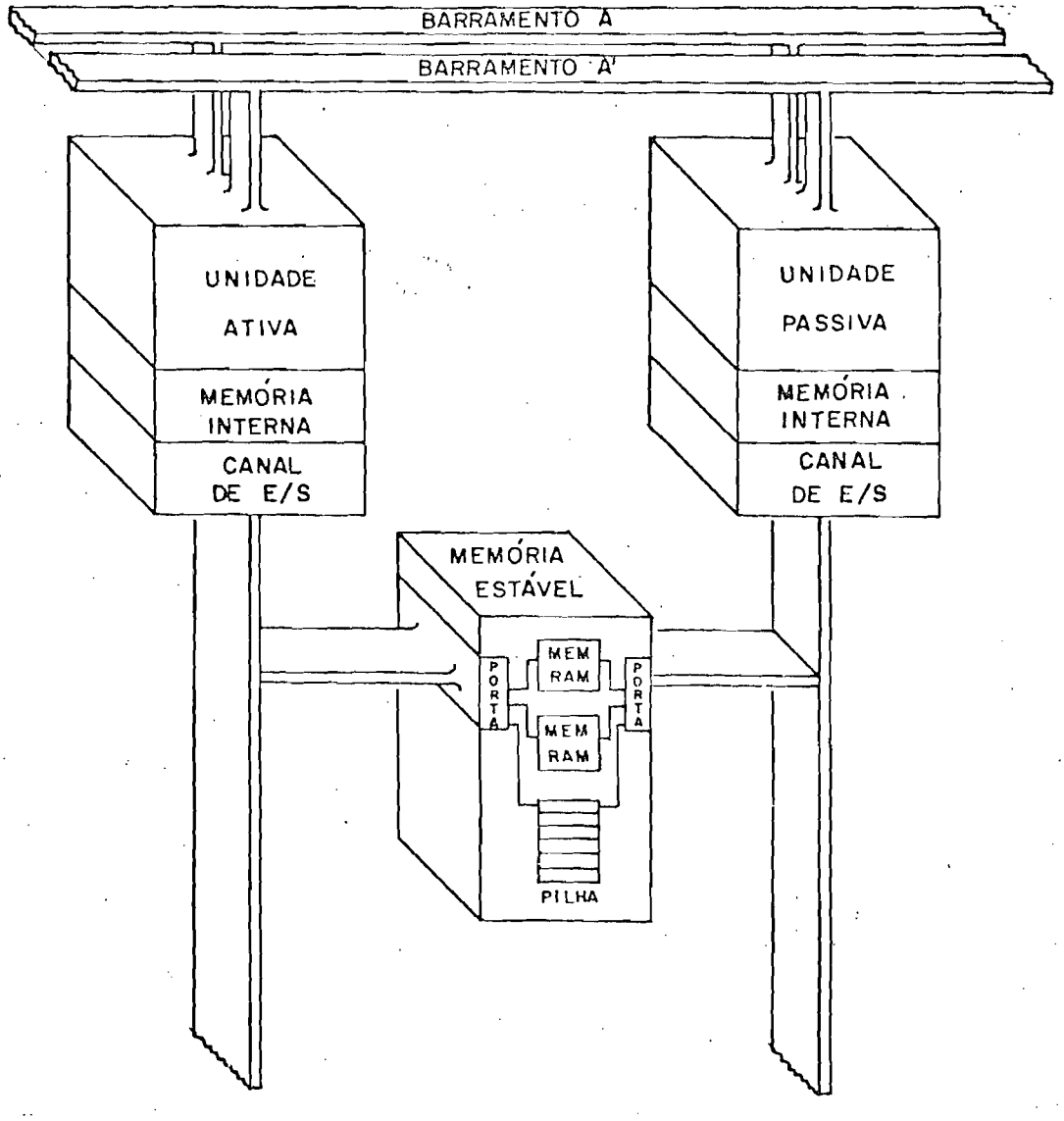


Fig. 4.5 - Memória Estável - ALTERNATIVA 2

A lógica de controle desta memória estável deve implementar a filosofia de operação descrita a seguir e ilustrada na Figura 4.6:

- Um MÓDULO (MEMRAM) da Memória Estável deve se encontrar no estado CORRENTE e o outro no estado ÚLTIMO.

- Toda escrita na memória interna da UNIDADE ATIVA é ao mesmo tempo feita no MÓDULO CORRENTE da memória estável, para manter este MÓDULO atualizado. Simultaneamente, o endereço, onde é realizada esta escrita, deve ser armazenado na *pilha*. Desta forma, todas as áreas de memória que estão sendo *modificadas* têm seus endereços armazenados na pilha.

- Na operação de leitura pelo processador da UNIDADE ATIVA, o acesso é permitido apenas para a memória interna desta UNIDADE.

- Para o chaveamento dos MÓDULOS (MEMRAMs) CORRENTE e ÚLTIMO deve ser enviado um sinal *"OUT porta memória estável"* pela PORTA conectada à UNIDADE ATIVA (linha de controle OUT 1 da Figura 4.6). A partir deste instante, o processador que enviou o "OUT" ficará suspenso (sinal da "WAIT" ativo) até que todas posições do atual MÓDULO CORRENTE, cujos endereços estão na pilha, sejam atualizados de acordo com o conteúdo do atual MÓDULO ÚLTIMO da Memória Estável.

Da mesma forma que na ALTERNATIVA 1, a MEMÓRIA ESTÁVEL deve conter no mínimo:

- Toda a *área de dados do NÚCLEO do Sistema Operacional iRMX86* (configurado nas posições 1533H e 15FEH como é mostrado no Apêndice B).

- Toda a *área de dados das INSTÂNCIAS ATIVAS* (alocadas a partir da posição 1600H). Estas áreas, diferentemente da ALTERNATIVA 1, podem ser alocadas dinamicamente, utilizando as facilidades de Gerenciamento de Memória providas pelo iRMX86.

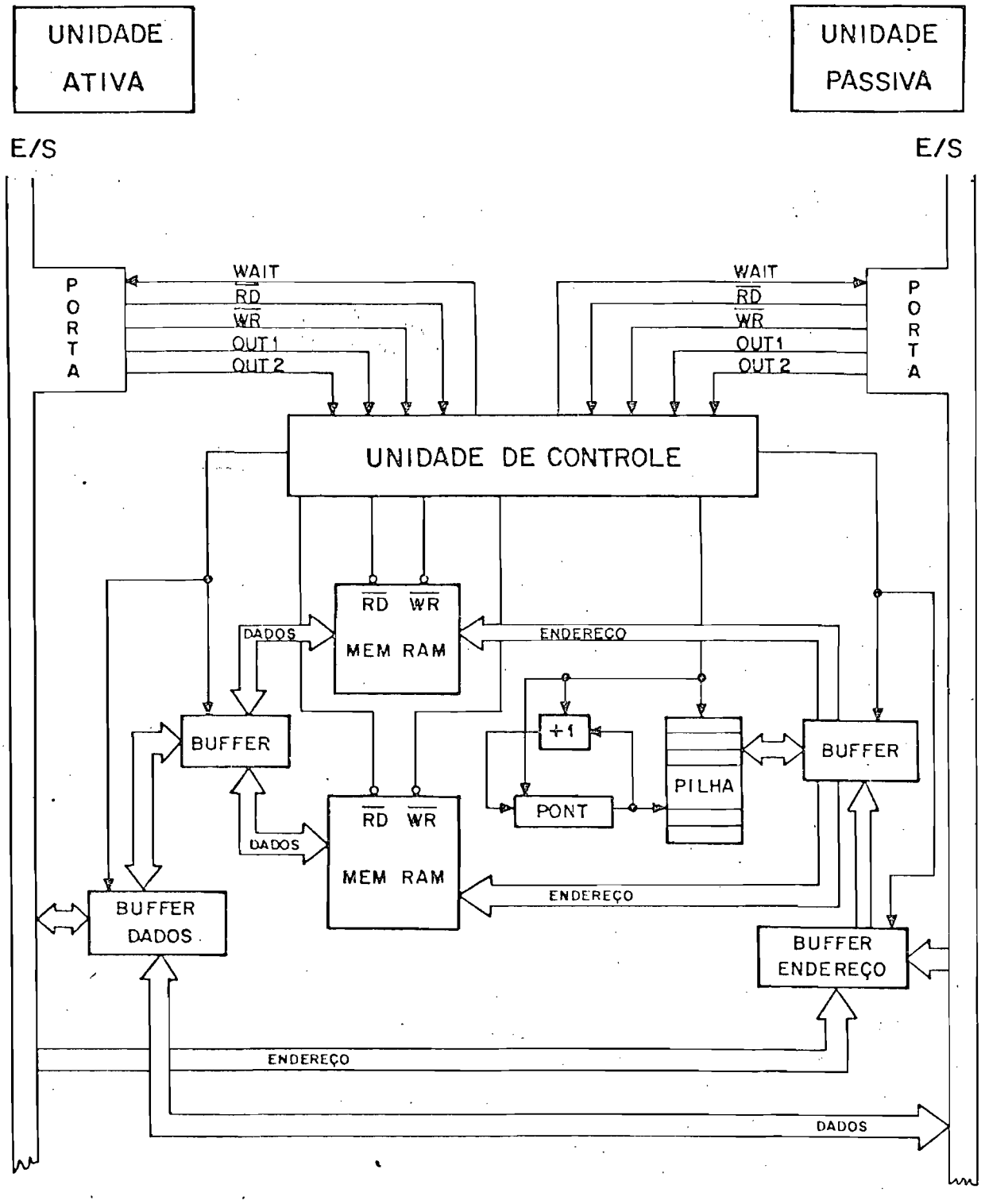


Fig. 4.6 - Esquema do circuito de controle da Memória Estável - ALTERNATIVA 2.

*Primitiva "SAVE"*

Na ALTERNATIVA 2, o "SAVE" deve transferir entre memórias apenas as modificações feitas depois do "SAVE" anterior. Assim, no momento que um "SAVE" é invocado por uma INSTÂNCIA ATIVA é desencadeado o seguinte procedimento:

1 - Execução do chaveamento automático dos MÓDULOS da memória estável para que o CORRENTE, que estava sendo atualizado simultaneamente com a memória interna da UNIDADE ATIVA, torne-se ÚLTIMO. Este chaveamento é desencadeado por um sinal *"OUT porta memória estável"* (linha de controle OUT 1 da Figura 4.6).

2 - Envio de um sinal de "WAIT" para o processador da UNIDADE ATIVA.

3 - Transferência das posições do MÓDULO ÚLTIMO (que se encontra idêntico à memória interna da UNIDADE ATIVA), cujos endereços estão especificados na *pilha de modificações*, deste MÓDULO ÚLTIMO para o MÓDULO CORRENTE da Memória Estável. Isto é feito para atualizar o MÓDULO CORRENTE, de modo que depois do "SAVE" ele se encontre idêntico às outras memórias (MÓDULO ÚLTIMO e memória interna da UNIDADE ATIVA). Para tanto, executa-se o seguinte procedimento: enquanto a *pilha de modificações* não estiver vazia

3.1 - retira da pilha o próximo endereço modificado;

3.2 - copia o conteúdo deste endereço, do MÓDULO ÚLTIMO para o MÓDULO CORRENTE da memória estável.

Cabe notar que um código que deteção de erro pode ser implementado no circuito da ALTERNATIVA 2, de modo a detetar erro na transferência memória a memória. Neste caso, o processador da UNIDADE ATIVA não é tirado do estado "WAIT" o que implica a parada da UNIDADE ATIVA e conseqüente recuperação do sistema pela UNIDADE PASSIVA.

Outro fato importante é a ocorrência de sucessivas modificações num mesmo endereço de memória. Isto faz com que a *pilha* contenha mais de uma vez o mesmo endereço o que resulta na transferência repetida de uma mesma posição de memória durante a *atualização* do MÓDULO CORRENTE. Existem várias maneiras de se evitar tal repetição, porém a escolha e implementação da melhor delas foge ao escopo deste trabalho.

Ainda com respeito à *pilha de modificações* cabe ressaltar que o seu tamanho (limite da pilha) é muito importante. Porém este dimensionamento deve ser estimado pelo projetista de Sistema de Supervisão, depois de uma avaliação da freqüência com que a primitiva "SAVE" é invocada na aplicação. Uma solução simples para este problema seria a implementação de um "SAVE" automático decorrente de *pilha cheia*.

A ALTERNATIVA 2 não utiliza a TABELA DE TRANSFERÊNCIA proposta na ALTERNATIVA 1. Portanto, nada impede que as tarefas da aplicação sejam alocadas dinamicamente já que este tipo de alocação é provido pelo iRMX86 (Seção 3.3.2). Neste caso, o único requisito é que a memória interna à UNIDADE ATIVA e os MÓDULOS da Memória Estável sejam carregados, na inicialização do sistema, de forma idêntica.

Igualmente à ALTERNATIVA 1, o chaveamento dos MÓDULOS (CORRENTE e ÚLTIMO) executado antes da trasferência visa garantir a *atomicidade da operação "SAVE"*.

### 4.2.2 - GERENTE DE RECUPERAÇÃO

Esta camada, incorporada ao Sistema Operacional iRMX86, tem por objetivo prover a *recuperação automática de falhas de forma transparente à aplicação*, isto é, sem qualquer interrupção aparente do processamento do Sistema de Supervisão.

O GERENTE DE RECUPERAÇÃO é uma tarefa que reside em ambas as UNIDADES ATIVA e PASSIVA. Caracteriza-se como uma tarefa de interrupção (Seção 3.3.2) ativada pela interface interpretativa (Sistema Opera

cional iRMX86) por ocasião da manipulação da exceção correspondente ao CÃO DE GUARDA.

O GERENTE DE RECUPERAÇÃO tem por função efetuar, o mais rápido possível, a recuperação do sistema, sendo para tanto a tarefa de mais alta prioridade, já que o Sistema Operacional iRMX86 é preemptivo por prioridade. Assim, garante-se que esta *tarefa de interrupção* quando tornada PRONTA receberá imediatamente o controle do processador, permanecendo neste estado (EM EXECUÇÃO) até que termine a recuperação.

Quando EM EXECUÇÃO, cabe ao *GERENTE DE RECUPERAÇÃO:*

1 - Desativar a UNIDADE com falha ("HOLD" da UCP) através do envio de um sinal por uma linha de controle específica do barramento que une as duas UNIDADES.

2 - Verificar o MODO DE OPERAÇÃO (ATIVO ou PASSIVO) em que se encontra a UNIDADE onde ele reside.

2.1 - Caso a UNIDADE esteja no MODO DE OPERAÇÃO PASSIVO, a sinalização é considerada como uma falha na UNIDADE ATIVA. Isto faz com que a PASSIVA recupere o sistema. Neste caso, o GERENTE DE RECUPERAÇÃO:

2.1.1 - Modifica o MODO DE OPERAÇÃO da UNIDADE PASSIVA para ATIVO através de uma sinalização pela linha de controle OUT 2 da Figura 4.3, quando implementada a ALTERNATIVA 1, e pela linha de controle OUT 2 da Figura 4.6, quando implementada a ALTERNATIVA 2;

2.1.2 - Transfere o conteúdo do MÓDULO ÚLTIMO da memória estável para a memória interna da recente UNIDADE ATIVA, via um controlador de Acesso Direto à Memória (ADM).

- Se a Memória Estável utilizada for aquela proposta na ALTERNATIVA 1, então a área de código das INSTÂNCIAS não precisa ser transferida pois a alocação de memória foi estática e feita na inicialização do sistema. Nada foi modificado em termos de código de programa. Basta que sejam transferidas as áreas de dados das INSTÂNCIAS. Para tanto, o GERENTE DE RECUPERAÇÃO utiliza a TABELA DE TRANSFERÊNCIA (recurso de apoio à tolerância a falhas).

- Se a Memória Estável utilizada for a proposta na ALTERNATIVA 2, torna-se mais simples transferir toda a memória, pois existe a possibilidade de as INSTÂNCIAS terem sido alocadas dinamicamente.
  Cabe notar que a simples transferência da MEMÓRIA ESTÁVEL para a memória interna da UNIDADE PASSIVA faz com que as INSTÂNCIAS PASSIVAS desta UNIDADE tornem-se ATIVAS e continuem a operação do sistema, a partir do último "SAVE" executado completamente.

2.2 - Caso a UNIDADE esteja no MODO DE OPERAÇÃO ATIVA, a sinalização é considerada como uma falha detetada na UNIDADE PASSIVA. Isto não deve acarretar nenhuma modificação na operação da UNIDADE ATIVA. Assim, cabe ao GERENTE DE RECUPERAÇÃO, simplesmente, avisar o operador do Sistema de Supervisão da ocorrência deste fato para que ele providencie a manutenção desta UNIDADE.

A seguir são apresentados dois procedimentos para implementação do GERENTE DE RECUPERAÇÃO, um para cada ALTERNATIVA proposta para a Memória Estável.

GERENTE DE RECUPERAÇÃO - ALTERNATIVA 1

0 - Início

1 - Desativa a UNIDADE com efeito e avisa o operador.

2 - *SE* esta UNIDADE estiver no MODO DE OPERAÇÃO ATIVA *ENTÃO* Fim.

3 - *SENÃO* faz o MODO DE OPERAÇÃO se tornar ATIVO.

4 - Copia as áreas de memória cujos endereços estão especificados na TABELA DE TRANSFERÊNCIA, transferindo estas áreas do MÓDULO ÚLTIMO da Memória Estável para a memória interna da atual UNIDADE ATIVA e para o MÓDULO CORRENTE da Memória Estável.

5 - Fim.

GERENTE DE RECUPERAÇÃO - ALTERNATIVA 2

0 - Início

1 - Desativa a UNIDADE com efeito e avisa o operador.

2 - *SE* esta UNIDADE estiver no MODO DE OPERAÇÃO ATIVA *ENTÃO* Fim.

3 - *SENÃO* faz o MODO DE OPERAÇÃO se tornar ATIVO.

4 - Copia todo o MÓDULO ÚLTIMO da Memória Estável para a memória interna da atual UNIDADE ATIVA e para o MÓDULO CORRENTE, tornando todas as memórias do sistema idênticas.

5 - Inicializa a *pilha de modificação*.

6 - Fim.

CÃO DE GUARDA

Apesar de o CÃO DE GUARDA não ser um recurso provido por esta camada (GERENTE DE RECUPERAÇÃO), e sim pelo "hardware" ele é utilizado como apoio à recuperação. Sendo assim, cabe apresentar aqui alguns detalhes de sua operação.

O CÃO DE GUARDA ("watch dog timer") é um mecanismo baseado em um temporizador implementado por "hardware" que gera uma interrupção na UNIDADE à qual se conecta, caso se complete um período de tempo. Este período de tempo só é atingido, ou seja, a interrupção só ocorre nas seguintes situações:

1 - a outra UNIDADE parou;

2 - foi detectado algum tipo de defeito de "hardware" na outra UNIDADE.

Em operação normal, quando não ocorrem as situações 1 e 2 mencionadas acima, o temporizador CÃO DE GUARDA deve ser reinicializado antes que se complete o período de tempo em questão, evitando que seja gerada a interrupção.

A reinicialização do temporizador, por exemplo, conectado à UNIDADE PASSIVA, é disparada pela UNIDADE ATIVA através do envio de um sinal pela linha de controle do barramento que interliga as duas UNIDADES (Seção 3.2.1). Este sinal simula a mensagem "EU ESTOU BEM" emitida pela UNIDADE ATIVA. A geração deste sinal é feita pela própria UNIDADE ATIVA através do atendimento de uma INTERRUPÇÃO DIAGNOSE que deve ocorrer em intervalos de tempo menores que o período de tempo associado ao temporizador que implementa o CÃO DE GUARDA. A rotina de atendimento desta INTERRUPÇÃO DIAGNOSE, como o próprio nome sugere, pode implementar alguns testes diagnósticos na UNIDADE ATIVA, antes de gerar o sinal "EU ESTOU BEM" que reinicializa o CÃO DE GUARDA da UNIDADE PASSIVA. Caso o resultado destes testes indique uma falha em alguma parte da UNIDADE ATIVA, o sinal de reinicialização ("EU ESTOU BEM") simplesmente não é gerado, o que implementa o *confinamento de erro* proposto na Seção 4.1.2. A parada do próprio processador da UNIDADE ATIVA enquadra-se como o caso trivial, pois o sinal de reinicialização não pode ser gerado.

É importante notar que, se houver falha na linha de controle do barramento, o sinal de controle "EU ESTOU BEM" enviado pelo processador ATIVO ao CÃO DE GUARDA da UNIDADE PASSIVA não chegará por esta linha, mas deverá chegar pela análoga do barramento redundante. Assim sendo, a falha de uma linha de controle não implicará recuperação do sistema.

A operação de reinicialização do temporizador CÃO DE GUARDA conectado à UNIDADE ATIVA é análoga ao exemplo da UNIDADE PASSIVA descrito acima, o que significa que a UNIDADE PASSIVA pode ficar executando AUTODIAGNOSE enquanto atua como reserva.

### 4.2.3 - ASPECTOS DE IMPLEMENTAÇÃO DOS MECANISMOS PROPOSTOS

As camadas propostas para *extensão do iRMX86* prover tolerância a falhas preservam as características multiníveis (conceito apresentado na Seção 2.4) do Sistema de Supervisão.

Como pode ser observado na Figura 4.7, o sistema como um todo foi estruturado em níveis de abstração (estruturação vertical). Cada camada do Sistema de Supervisão Tolerante a Falhas implementa um nível de abstração.

A interação entre os diversos níveis é implementada pelas chamadas ao sistema providas pelo iRMX86 (vide o Apêndice A) acrescidas das novas chamadas que compõem as extensões propostas. As chamadas ao sistema funciona como *interpretadores* entre níveis.

Nota-se por exemplo, a chamada ao sistema "SAVE". Ela é invocada pela aplicação, porém é um mecanismo implementado pelo nível RECURSO PARA TOLERÂNCIA A FALHAS que por sua vez utiliza a memória estável provida pelo nível de "hardware" para armazenar o estado do sistema.

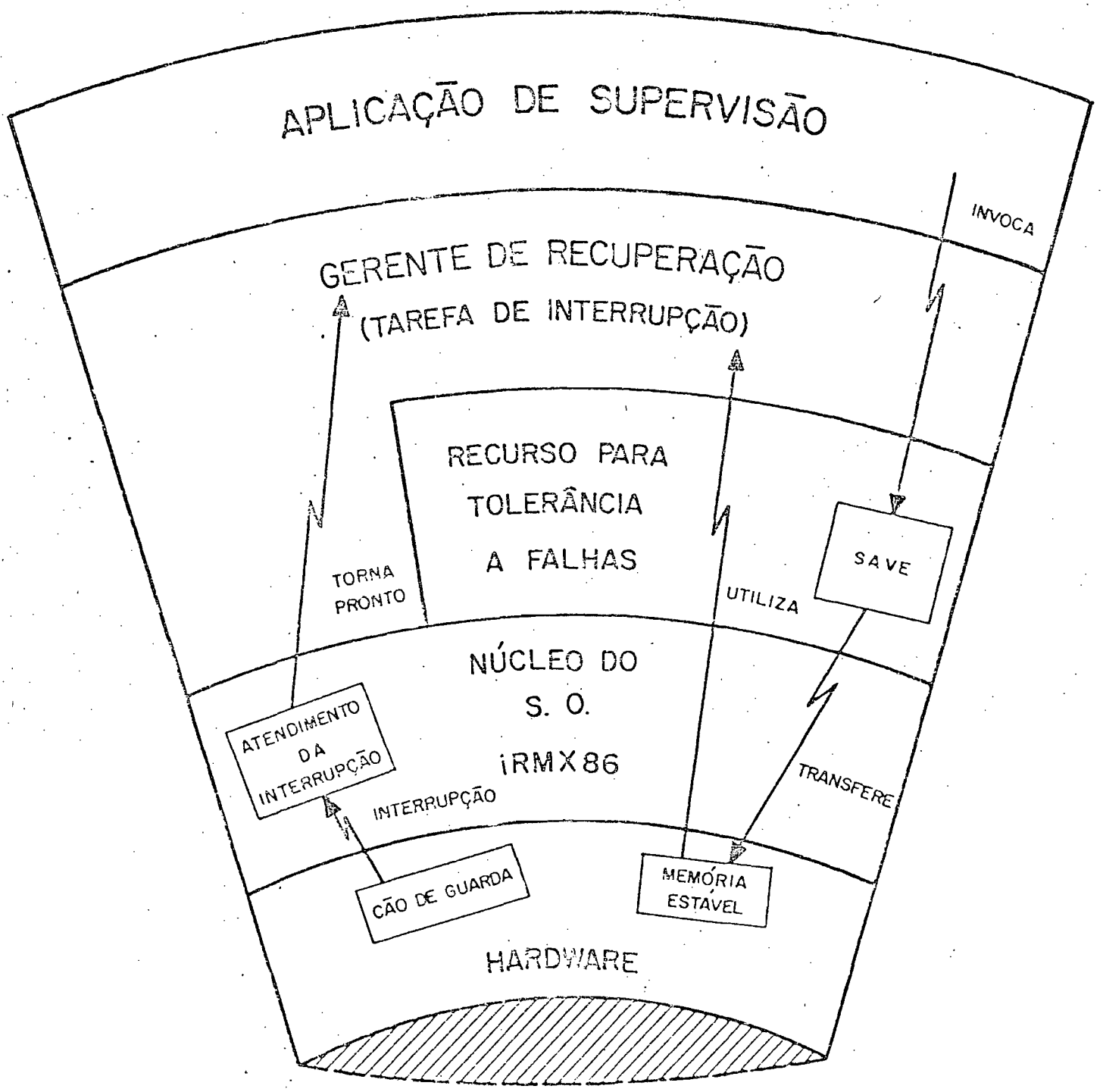


Fig. 4.7 - Sistema de Supervisão multinîveis tolerante a falhas.

Na ALTERNATIVA 1 devem ser implementadas, ainda neste nível, outras duas novas chamadas ao sistema: CRIAJOB e CRIATAREFA que permitem a geração da TABELA DE TRANSFERÊNCIA.

No caso do nível do GERENTE DE RECUPERAÇÃO, esta camada é essencialmente implementada por uma *Tarefa de Interrupção* tornada PRONTA por ocasião da ocorrência da interrupção do CÃO DE GUARDA (nível de "hardware"), através do atendimento desta interrupção, feito pelo NÚCLEO (interface interpretativa).

Cabe citar, com respeito à ALTERNATIVA 1, que o nível do GERENTE DE RECUPERAÇÃO faz uso da TABELA DE TRANSFERÊNCIA provida pelo nível inferior (RECURSO PARA TOLERÂNCIA A FALHAS) preservando assim a característica multinível, mesmo em termos das camadas adicionais propostas nesta dissertação.

A incorporação destes mecanismos ao iRMX86 de modo a estender este Sistema Operacional para Tolerância a Falhas é bastante simples pois consiste em escrever apenas a *Parte de Serviço* de um *Gerente de Tipo* apresentada na Seção 3.4.2.

# CAPÍTULO 5

## EFICIÊNCIA E CONSISTÊNCIA DA RECUPERAÇÃO

Neste capítulo é feita uma análise dos mecanismos propostos no Capítulo 4 no que consiste a eficiência deles. Salienta-se uma comparação entre as duas ALTERNATIVAS para implementação da Memória Estável. Além disso, mostra-se que, com a técnica de recuperação de erro adotada e a observação de algumas regras simples para estabelecimento dos pontos de recuperação, a consistência da recuperação do Sistema de Supervisão pode ser assegurada.

### 5.1 - ANÁLISE DE EFICIÊNCIA DA MEMÓRIA ESTÁVEL

Os mecanismos de apoio à tolerância a falhas propostos no Capítulo 4 compreendem basicamente a memória estável, provida da chamada ao sistema *"SAVE"*, e o GERENTE DE RECUPERAÇÃO.

Em termos de GERENTE DE RECUPERAÇÃO como já mencionado, ele é implementado por uma *tarefa de interrupção*, aquela de maior prioridade no Sistema de Supervisão. Porém, com respeito à MEMÓRIA ESTÁVEL torna-se conveniente fazer uma análise, em termos de eficiência, das ALTERNATIVAS propostas na Seção 4.2.1.1.

Cabe lembrar que a ALTERNATIVA 1 implementa a estratégia *"ponto de teste"* essencialmente por "software". A outra, ALTERNATIVA 2, implementa um caso especial da estratégia de ponto de teste, denominada *"recovery cache"*, porém com maior apoio de recursos de "hardware".

As duas ALTERNATIVAS permitem a recuperação do Sistema de Supervisão em caso de falhas simples de "hardware". Entretanto, diferenças significativas em termos de eficiência podem ser constatadas.

Como argumentado na Seção 4.2.1.1, ALTERNATIVAs 1 e 2, o fato de o iRMX86 ser um Sistema Operacional fechado à aplicação exige que, a cada "SAVE" dado por uma INSTÂNCIA ATIVA, um "SAVE VIRTUAL" nas

demais INSTÂNCIAS da UNIDADE ATIVA seja realizado. Isto implica uma transferência de dados bastante significativa a cada "SAVE", que é proporcional ao número de tarefas executadas pelo sistema. Como isto, a invocação de um "SAVE" por uma INSTÂNCIA ATIVA deixa de ser independente da invocação de um "SAVE" por outra INSTÂNCIA ATIVA. A freqüência com que esta função é invocada pelas tarefas é um parâmetro crítico, pois quanto maior a ocorrência de "SAVE" num intervalo de tempo, menor o tempo restante deste intervalo para a execução das atividades de supervisão. Este fato já seria crítico se o "SAVE" dado por uma tarefa transferida apenas as informações de estado daquela tarefa, quanto mais no caso do "SAVE VIRTUAL". Isto pode ser observado analiticamente supondo:

$t_i$ - intervalo de tempo necessário para a transferência de estado de uma *tarefa*;

M - número total de tarefas do sistema;

$t_n$ - intervalo de tempo necessário para a transferência da área de dados do NÚCLEO para a Memória Estável.

Desta forma,

- se o "SAVE" não for virtual, então o tempo total gasto em transferências num "SAVE" dado por uma $tarefa_i$ será:

(1) $(t_i + t_n)$;

- se o "SAVE" for "VIRTUAL", então o tempo total gasto com transferências num "SAVE" dado por uma $tarefa_i$ será:

(2) $\left[ \sum_{i=1}^{M} (t_i) + t_n \right] = \left[ t_n + t_i + \sum_{\substack{j=1 \\ j \neq i}}^{M} (t_j) \right]$.

Como $t_i$ e $t_n$ são constantes tanto em (1) quanto em (2), então:

$$(2) - (1) = t_n + t_i + \sum_{\substack{j=1 \\ j \neq i}}^{M} (t_j) - t_i - t_n = \sum_{\substack{j=1 \\ j \neq i}}^{M} (t_j) ,$$

o que significa um acréscimo de tempo de

$$\sum_{\substack{j=1 \\ j\neq i}}^{M} (t_j),$$

por "SAVE", proporcional ao número total de tarefas do sistema.

Supondo que num intervalo T sejam executadas K tarefas concorrentemente e que cada tarefa$_i$ invoca neste intervalo $S_i$ "SAVES", então:

- se o "SAVE" não for virtual, o tempo total gasto com transferencias será:

$$(1') \quad \left[\sum_{i=1}^{K} S_i \, (t_i + t_n)\right];$$

- se o "SAVE" for "VIRTUAL", o tempo total gasto em transferências será:

$$(2') \quad \left[\sum_{i=1}^{K} S_i \left[\sum_{i=1}^{M} (t_i) + t_n\right]\right].$$

Como $t_i$ e $t_n$ são constantes tanto em (1') quanto em (2') então:

$$(2') - (1') = \sum_{i=1}^{K} S_i \sum_{i=1}^{M} (t_i) + \sum_{i=1}^{K} S_i \, t_n - \sum_{i=1}^{K} S_i \, t_i - \sum_{i=1}^{K} S_i \, t_n$$

$$= \left(\sum_{i=1}^{K} S_i \cdot \sum_{j=1}^{M} t_j\right) - \left(\sum_{i=1}^{K} S_i \, t_i\right)$$

$$= \sum_{i=1}^{K} S_i \left(\sum_{j=1}^{M} t_j - t_i\right)$$

$$= \left[\sum_{i=1}^{K} S_i \cdot \left(\sum_{\substack{j=1 \\ j\neq i}}^{M} t_j\right)\right], \quad (3)$$

onde

$\sum_{i=1}^{K} S_i$ = número total de "SAVES" dado por K tarefas no intervalo T.

Desta forma fica demonstrado que o tempo gasto em T com a transferência de "SAVEs", no caso "SAVE VIRTUAL", além de crescer com o número de tarefas do sistema, aumenta proporcionalmente com o número de "SAVEs" executados neste intervalo. Vê-se que a adoção do "SAVE VIRTUAL" acarreta um acréscimo igual a (3), em termos de tempo, em relação ao "SAVE" simples.

Considerando que um Sistema de Supervisão é caracterizado como um sistema de controle de processos em tempo real, não se admite um gasto de processamento tão grande para prover a disponibilidade ao sistema.

Neste contexto, torna-se interessante a adoção da ALTERNATIVA 2, pois, apesar de a restrição do "SAVE VIRTUAL" também se fazer presente nesta ALTERNATIVA, executa-se neste caso apenas a *transferência das áreas de memória que foram modificadas depois do último "SAVE"*, o que naturalmente significa bem menos transferências que no caso anterior (ALTERNATIVA 1). Além disso, a freqüência com que o "SAVE" é invocado pelas INSTÂNCIAS ATIVAS pode ser considerado um parâmetro menos crítico, pois quanto mais frequente o "SAVE", menos modificações foram efetuadas e portanto menor o número de transferências a serem feitas. Assim, pode-se afirmar que não há dependência entre as tarefas na invocação da chamada "SAVE". Cada INSTÂNCIA ATIVA pode invocar a primitiva "SAVE" independentemente das outras INSTÂNCIAS da UNIDADE ATIVA.

## 5.2 - ESTABELECIMENTO DE PONTOS DE RECUPERAÇÃO

A definição dos pontos de recuperação pelo projetista do Sistema de Supervisão, apesar de ser bastante adequada para sistemas do tipo controle de processos, pode acarretar muitas interrupções no processamento de supervisão. O tempo gasto para transferência das informa

ções de estado do sistema, mesmo no caso da transferência de poucas modificações, não deve ser significativo a ponto de comprometer o desempenho de tempo real do Sistema de Supervisão. Isto deve ser cuidadosamente analisado.

Considerando que o projetista de um Sistema de Supervisão tem conhecimento total da aplicação, torna-se possível e muito conveniente explorar este fato no sentido de *otimizar o uso dos pontos de recuperação*.

Em Sistemas de Supervisão a maioria das tarefas têm suas execuções ciclicamente repetidas. Assim, na maior parte dos casos, estas tarefas da aplicação têm uma *freqüência natural de repetição* determinada, por exemplo, por um dispositivo supervisionado ou mesmo por uma seqüência de atividades que devem ser realizadas periodicamente.

Esta característica dos Sistemas de Supervisão permite considerar que a maioria dos processos (ou tarefas) da aplicação de supervisão possuem um *ciclo de controle natural*. Desta forma, poucas outras tarefas do sistema são aleatórias e neste conjunto se enquadram, por exemplo, as tarefas executadas quando um erro é detetado.

Dada esta característica, a otimização no uso dos pontos de recuperação pode ser feita, por exemplo, através da aplicação do conceito de ciclos de controle, introduzido na Seção 2.6 e detalhado no decorrer deste capítulo.

Outro aspecto relevante quanto ao estabelecimento de pontos de recuperação diz respeito à comunicação. Num sistema de controle de processos, particularmente num Sistema de Supervisão, a comunicação entre processos ou mesmo comunicações entre os processos dos sistema e as INTERFACES de AQUISIÇÃO e ATUAÇÃO são bastante freqüentes.

Neste sentido alguns casos especiais serão levantados e será definida uma regra para o estabelecimento de pontos de recuperação mais adequados. A observação desta regra pelo projetista do Sistema de Supervisão visa assegurar ao sistema o que se denomina aqui por *comunicação confiável*.

### 5.2.1 - CICLOS DE CONTROLE

Sistemas de Supervisão por serem do tipo controle de processos têm um comportamento bem determinado e possuem um ciclo de controle natural. Geralmente são constituídos por conjuntos de *processos cíclicos* (definidos na Seção 2.6) sendo referenciados por sistemas cíclicos.

Assim, torna-se simples e aplicável na prática o estabelecimento periódico de pontos de recuperação de acordo com o ciclo de controle do sistema. Para tanto, o projetista do Sistema de Supervisão teria de estruturar a execução dos seus processos em ciclos de controle e determinar o *período do ciclo de controle do Sistema de Supervisão*.

Anderson e Knight (1983) apresentam um modelo denominado GRAFO DE SINCRONIZAÇÃO que pode ser utilizado para estruturação de um sistema de tempo real. Este modelo não está limitado a projetos específicos, podendo ser adotado em quaisquer *aplicações cíclicas de tempo real*. Este modelo é apresentado a seguir.

Sistemas de tempo real são modelados como um conjunto de processos seqüenciais cooperantes com restrições em termos de tempo de execução. Processos são usualmente executados periodicamente e, para um dado processo, os intervalos entre iniciação duram quase sempre o mesmo tempo.

GRAFO DE SINCRONIZAÇÃO

Seja um conjunto de processos denominados $P_1$, $P_2$, ..., $P_n$ e um conjunto de instantes de tempos distintos $T_1$, $T_2$, ..., $T_m$, os quais representam restrições de tempo em alguma escala apropriada.

Seja G um grafo dirigido (dígrafo) finito e acíclico com exatamente *um nó de entrada* de grau zero e exatamente *um nó de saída* de grau zero.

Cada nó de G é rotulado com um nome de processo ($P_i$) ou uma restrição de tempo ($T_i$) e mais que um nó pode ser rotulado com o mesmo nome de processo. Porém, o mesmo não se aplica às restrições de tempo.

Um arco de G pode conectar apenas dois nós com diferentes tipos de rótulos, ou seja, G é um bígrafo.

Os nós de entrada e saída de G são rotulados com restrições de tempo.

Se houver um caminho em G que liga um nó rotulado $T_i$ a um nó rotulado com $T_j$, então $T_i$ deve ser menor, em termos de índice que $T_j$.

Um nó rotulado com um nome de processo ($P_i$) indica a execução daquele processo e deve ter graus de entrada e saída iguais à unidade. Este nó é conectado a dois nós rotulados com restrições de tempo, as quais são características de tempo real do processo ($P_i$). Um nó rotulado com restrição de tempo é um *ponto de sincronização do processo* (com outros processos e/ou com o mundo exterior) e o rótulo do nó especifica o tempo decorrido para que esta sincronização ocorra.

No grafo, um processo nomeado ($P_i$) deve iniciar execução no tempo especificado pelo nó de entrada para $P_i$ e deve completar a execução antes ou no instante de tempo especificado pelo nó de saída de $P_i$.

Desta forma G é referido como um GRAFO DE SINCRONIZAÇÃO.

Sistemas de tempo real arbitrariamente complexos podem ser modelados usando tais grafos, pois, em geral, estes sistemas exibem uma modularidade natural que pode ser representada com o aninhamento de grafos de sincronização. Uma parte do sistema pode ser representada com um grafo separado e este usado como um nó do grafo que representa o sistema completo.

A Figura 5.1 mostra um exemplo de grafo de sincronização. Nesta $P_1$, $P_2$, $P_3$, ..., $P_8$ são nomes de processos e $T_1$, $T_2$, ..., $T_7$ são restrições de tempo.

O significado do grafo de sincronização apresentado em termos de processamento é o seguinte:

1 - $P_1$, $P_2$, $P_3$ e $P_4$ iniciam execução concorrente no instante $T_1$, e $P_4$ deve terminar sua execução em $T_2$.

2 - $P_5$ inicia execução em $T_2$ e juntamente com $P_3$ deve terminar sua execução em $T_3$.

3 - Em $T_3$ os processos $P_6$, $P_7$ e $P_8$ iniciam execução concorrente e juntamente com $P_2$ devem completar em $T_4$.

4 - Todos os processos, exceto $P_1$, repetem esta sequência de execução com novas restrições de tempo: $T_5$, $T_6$ e $T_7$.

5 - $T_7$ é a restrição de tempo final. Neste instante $P_6$, $P_7$ e $P_8$ devem terminar suas execuções bem como $P_1$ e $P_2$.

6 - A execução completa da seqüência pode ser repetida tantas vezes quanto se queira. Porém, as restrições de tempo devem ser incrementadas por $T_7 - T_1$ no final de cada iteração.

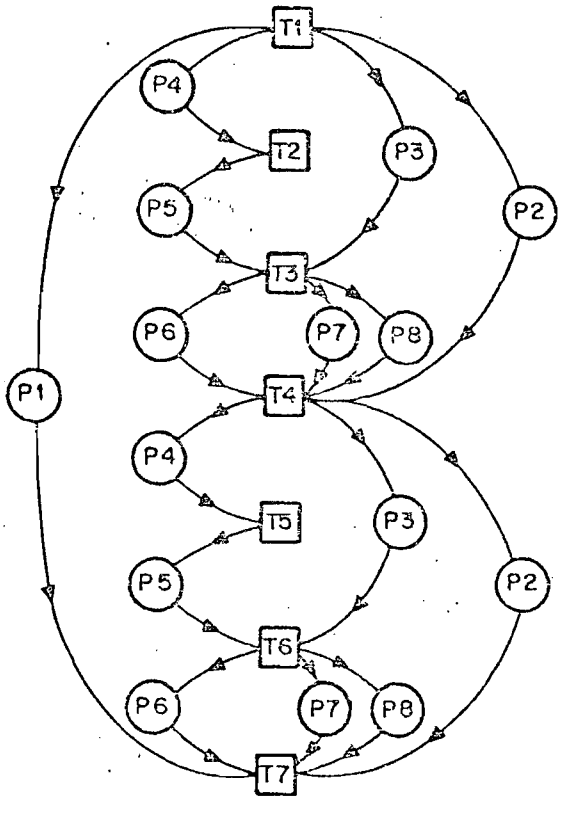


Fig. 5.1 - Um exemplo de grafo de sincronização (Anderson and Knight, 1983).

O intervalo (T) = $T_7 - T_1$ do GRADO DE SINCRONIZAÇÃO apresentado no exemplo seria um possível ciclo de controle do sistema.

De posse do *ciclo de controle do Sistema de Supervisão*, que não necessariamente seria obtido a partir de um GRAFO DE SINCRONIZAÇÃO, o projetista teria condições de aplicar a seguinte regra.

REGRA I - Um "SAVE" deve ser invocado no início de cada ciclo de controle do sistema.

Esta regra garante que *não mais que uma operação "SAVE" seja executada por ciclo de controle*. Porém, a invocação do "SAVE" não poderia ser através dos processos do sistema. Uma tarefa de interrupção associada a um nível de interrupção especificado para o ciclo de controle do sistema implementaria a operação "SAVE".

Desta forma, a função "SAVE" não seria mais incorporada ao Sistema Operacional iRMX86 como uma nova primitiva, mas sim como uma tarefa de interrupção. Neste caso, caberia ao projetista do Sistema de Supervisão apenas a especificação do intervalo de tempo de ocorrência da *interrupção do ciclo de controle do sistema*.

O estabelecimento de pontos de recuperação do INÍCIO dos ciclos de controle é uma regra bastante simples que garante um limite em termos de tempo, previamente conhecido, na extensão de retorno do sistema a um estado anterior, em casos de falhas.

A adoção desta solução (ciclo de controle), embora muito adaptável à aplicação, faz com que o estabelecimento efetivo dos pontos de recuperação fique a cargo da *interface interpretativa* (no caso, a EXTENSÃO do Sistema Operacional iRMX86 que provê a tolerância a falha - nível do RECURSO PARA TOLERÂNCIA A FALHAS), e não mais ao nível da APLICAÇÃO (invocação da primitiva "SAVE" pelos processos de supervisão). Porém, cabe notar que em termos de *otimização* no estabelecimento dos pontos de recuperação do sistema esta solução é bastante atraente. E mais, esta solução não fere os princípios da hierarquia dos sistemas multiníveis até então utilizados neste trabalho.

Apesar da adoção da ALTERNATIVA 2 (proposta na Seção 4.2.1.1 desta dissertação) para implementação da Memória Estável do Sistema de Supervisão Tolerante a Falhas, a existência do "SAVE VIRTUAL" não pode ser esquecida. Mesmo nesta ALTERNATIVA, a invocação de um "SAVE" por uma INSTÂNCIA ATIVA não salva apenas *as modificações* sofridas por esta INSTÂNCIA depois do "SAVE" anterior, mas salva todas as modificações feitas no sistema depois do último "SAVE" dado por qualquer INSTÂNCIA ATIVA. Portanto o conceito de "SAVE VIRTUAL" permanece.

Assim sendo, é importante salientar que o estabelecimento de um ponto de recuperação no início de um ciclo de controle não se limita ao armazenamento de estado dos processos envolvidos no ciclo, mas de todos os processos do Sistema de Supervisão, inclusive aqueles de execução aleatória.

### 5.2.2 - COMUNICAÇÃO CONFIÁVEL

Entende-se por *comunicação confiável* aquela provida pelo Sistema Operacional iRMX86, que pode ser síncrona ou assíncrona, porém acompanhada do estabelecimento de um ponto de recuperação (invocação da primitiva "SAVE" imediatamente antes ou após a invocação das chamadas ao sistema comunicação, providas pelo iRMX86).

A comunicação confiável é considerada muito útil em Sistemas de Supervisão Tolerantes a Falhas, pois existem situações em que a comunicação simples entre processos e mesmo do sistema com o meio exterior pode ser prejudicada, na ocorrência de falhas e conseqüente recuperação retroativa do sistema. A seguir estas situações são salientadas e duas regras simples podem ser observadas para que elas sejam parcialmente evitadas.

Considerando as características de um Sistema de Supervisão, especialmente das suas INTERFACES de AQUISIÇÃO, certos dados de entrada não podem ser *mantidos* por esta INTERFACE. Assim, no caso de ocorrência de falha logo após uma AQUISIÇÃO, o retorno do processo de aquisição para antes da aquisição, quando reexecutado, não mais poderia dispor *repetidamente* destes mesmos dados. Neste caso seria interessante que o projetista do Sistema de Supervisão dispusesse de um tipo de comunicação confiável que evitasse este problema. Para tanto define-se:

REGRA II - Um "SAVE" deve ser executado por uma INSTÂNCIA ATIVA imediatamente *após* uma requisição de dados da INTERFACE de AQUISIÇÃO.

A aplicação desta REGRA significa o estabelecimento de um ponto de recuperação imediatamente após uma comunicação do processo de aquisição com a INTERFACE de AQUISIÇÃO. Assim, garante-se uma *comunicação confiável* mesmo na ocorrência de uma falha depois da aquisição. Porém, mesmo assim, nada se pode garantir em termos de aquisições dos mesmos dados, se a falha ocorrer antes da invocação do "SAVE".

Cabe notar que se a freqüência dos dados adquiridos pela INTERFACE de AQUISIÇÃO for muito alta, em caso de falha de UNIDADE ATIVA, certos dados certamente serão perdidos. Neste caso a única solução seria prover a INTERFACE de AQUISIÇÃO de memória ("buffer") para armazenar internamente um certo número de dados.

Assim como a aquisição de dados citada acima, a comunicação para atualização de dados no disco do tipo portas duais, provido pela arquitetura básica proposta na Seção 3.2, também é crítica. Imagine-se, por exemplo, a seguinte situação:

1 - uma INSTÂNCIA ATIVA A leu certas informações do disco;

2 - em seguida esta INSTÂNCIA processou as informações lidas e obteve novos resultados que foram atualizados no disco;

3 - ocorreu uma falha na UNIDADE ATIVA;

Suponha-se que na recuperação a INSTÂNCIA PASSIVA volte para antes do passo 1. Neste caso tem-se que a INSTÂNCIA A não contará mais com as mesmas informações antigas e sim com os novos resultados, o que significa que sua reexecução será uma nova iteração.

Fazendo uma analogia com a situação da INTERFACE de AQUISIÇÃO apresentada vê-se que o problema é o mesmo: os dados a serem adquiridos do disco não são mais os mesmos.

Também neste caso a aplicação da REGRA II pelo projetista do sistema (logo após a leitura das informações do disco - passo 1) provê uma comunicação confiável. Assim, tem-se garantido que os dados antigos, lidos do disco, encontram-se na memória estável e que a recuperação os utilizará, evitando uma nova iteração.

Com respeito às INTERFACES DE ATUAÇÃO de um sistema de supervisão pode-se levantar o problema da *repetição* de uma atuação no caso de ocorrência de falha logo após uma atuação ter sido realizada e a conseqüente recuperação de falha retornasse a um estado anterior à atuação. Nestes casos seria aconselhável que o projetista do sistema de supervisão fizesse uso da comunicação confiável aplicando a seguinte regra:

REGRA III - Um "SAVE" deve ser executado por uma INSTÂNCIA ATIVA imediatamente *após* uma saída de dados pela INTERFACE DE ATUAÇÃO.

Assim, a aplicação desta regra evita a *repetição* indesejável de certas *atuações*. Contudo, deve ser lembrado que as limitações intrínsecas à técnica de *recuperação retroativa* adotada não podem ser evitadas. Naturalmente, na recuperação de falhas, certas atividades já executadas pelo sistema deverão se repetir.

## 5.3 - ANÁLISE DE CONSISTÊNCIA

Como citado na Seção 4.1.4, a forma adotada para implementação dos mecanismos de recuperação de erro pode gerar problemas de consistência.

Garantir a consistência implica, entre outros, assegurar que o *efeito dominó* não ocorre.

A observação da definição de *consistência* apresentada na Seção 2.6 garante a recuperação consistente do sistema em caso de falha.

A prova de que esta definição é observada pelas técnicas propostas nesta dissertação é bastante simples. Ela é apresentanda a seguir:

Considerando que a utilização de um Sistema Operacional padrão fechado à aplicação implicou a necessidade do "SAVE VIRTUAL", então, pela definição de consistência, tem-se que:

(1) o subconjunto de processos $P_1$, $P_2$, $P_3$, ..., $P_n$ compreende sempre o total de INSTÂNCIAS ATIVAS do Sistema de Supervisão;

(2) os pontos de recuperação estabelecidos pelos processos $P$ , $P$ , ..., $P_n$, respectivamente em instantes $T_1$, $T_2$, ..., $T_n$ distintos, neste caso são idênticos: $T_1 = T_2 = \ldots T_n$.

Portanto, demonstra-se que são satisfeitas:

- a condição (i) da definição de consistência, pois:

  para todo $T_i$, $T_j \xrightarrow{(2)} T_j = T_i \longrightarrow T_j - T_i = 0$, o que garante a inexistência de quaisquer tipo de comunicação entre $P_i$ e $P_j$ neste intervalo;

- a condição (ii) da definição de consistência, pois:

  no período $T_i$ a 'T', para 'T' > $T_i$, $P_i$ não se comunica com nenhum outro processo $P_k$ que não estiver no subconjunto devido (1) à não-existência de processos fora do subconjunto.

Como a técnica de recuperação retroativa adotada satisfaz a definição de Anderson e Lee (1981) de consistência, então tem-se garantida a não-ocorrência do efeito dominó.

Ainda, com respeito a esta técnica, cabe citar que ela implementa o método que limita as interações entre processos numa *conversação* (Randell, 1975) apresentado na Seção 2.6. O "SAVE VIRTUAL" im

plementa o caso mais simples de conversação, aquele em que todos os processos do sistema estão engajados na conversação e a entrada na conversação é simultânea. Esta entrada é renovada a cada estabelecimento de um ponto de recuperação por qualquer uma das INSTÂNCIAS ATIVAS do Sistema de Supervisão.

Existem outros três problemas de consistência que foram levantados na Seção 4.1.4:

- repetição da entrada;
- geração de saídas;
- transferência para a Memória Estável.

Quanto à *repetição da entrada*, preservação dos dados adquiridos, a aplicação da REGRA II (comunicação confiável) garante que no caso de falha, se a falha ocorreu depois de a aquisição ter sido realizada, a sua recuperação estará consistente com os mesmos dados de entrada.

No que diz respeito à *geração de saída*, em termos de consistência, cabe lembrar que este problema já foi discutido na Seção 4.1.2 e pode ser resolvido com a implementação de ASSERÇÕES. Estas ASSERÇÕES visam garantir a consistência dos dados resultantes de algum processamento crítico, evitando assim que certas ATUAÇÕES inconsistentes sejam efetivadas.

Finalmente, o problema da transferência para a memória estável já foi totalmente solucionado com as lógicas de controle propostas, tanto para a ALTERNATIVA 1 quanto para a ALTERNATIVA 2 (Seção 4.2.1.1). O chaveamento dos MÓDULOS CORRENTE e ÚLTIMO é executado antes da transferência para a Memória Estável, exatamente para garantir que se um erro ocorrer durante a atualização da Memória Estável, esta atualização será desconsiderada. A recuperação utilizará o MÓDULO ÚLTIMO que se encontra num estado consistente.

# CAPITULO 6

## CONCLUSÃO

Como apresentado no decorrer desta dissertação, seu principal objetivo foi a incorporação de técnicas de tolerância a falhas a um sistema de controle de processos de tempo real que se destina a atividades de supervisão. Estas técnicas visam prover um Sistema de Supervisão de maior disponibilidade por ocasião da ocorrência de falhas simples de "hardware".

Considerando as características comuns dos Sistemas de Supervisão, foi proposta uma arquitetura básica de "hardware" em torno da qual todo o trabalho foi desenvolvido. Esta arquitetura provê *redundância dupla* ao Sistema de Supervisão.

Em termos de "software" básico teve-se o propósito de utilizar um Sistema Operacional modular, configurável, o iRMX86, disponível no mercado. Isto se deve ao fato de se pretender não apenas tornar um Sistema de Supervisão tolerante a falhas mas, principalmente, de fazê-lo da forma mais simples possível, sem ter de desenvolver um Sistema Operacional Tolerante a Falhas específico.

Apesar de o Sistema Operacional escolhido ser *expansível*, o que permite sua extensão com novos objetos e chamadas ao sistema, algumas restrições nas técnicas de recuperação de erro propostas tiveram de ser impostas.

O fato de o Sistema Operacional ser fechado à modificações do usuário impede que se tenha conhecimento real das áreas de dados utilizadas pelas INSTÂNCIAS ATIVAS individualmente. Conhece-se apenas a área de dados total alocada a cada INSTÂNCIA e isto somente se a alocação de memória for estática e definida pelo usuário na inicialização do sistema. No que diz respeito à área de dados do NÚCLEO do iRMX86 tem-se apenas os seus limites, desconhecendo-se portanto as áreas onde estão descritas cada INSTÂNCIA ATIVA.

Estas limitações implicaram a necessidade de utilizar o conceito de "SAVE VIRTUAL" apresentado no Capítulo 4.

O "SAVE VIRTUAL", apesar de ser uma restrição que garante a consistência da recuperação do Sistema de Supervisão (não-ocorrência do efeito dominó, quando se adota a técnica de recuperação retroativa), causa uma *alta taxa de tranferência de dados para a memória estável* quando implementada a estratégia de "ponto de teste" (ALTERNATIVA 1 proposta no Capítulo 4).

A solução encontrada para minimizar esta taxa de transferência foi a adoção da estratégia "recovery cache" para recuperação retroativa (ALTERNATIVA 2 proposta no Capítulo 4) apoiada por recursos específicos de "hardware".

Assim, o armazenamento do estado do sistema num ponto de recuperação de forma automática por uma *memória estável* com a implementação da estratégia "recovery cache" torna-se mais vantajoso e adequado a aplicações de supervisão. Se o "hardware" utilizado pelo Sistema de Supervisão não possuir tal capacidade, como acontece com certas máquinas, o tempo de processamento dispendido para a provisão de tolerância a falha torna-se inaceitável para a implementação de um caso genérico que utiliza apenas recursos de programação. Nestes casos, as técnicas empregadas para tornar um Sistema de Supervisão tolerante a falhas não devem seguir as propostas neste trabalho.

A solução acima apresentada foi considerada como uma forma viável e consistente para recuperar de maneira retroativa um Sistema de Supervisão, no caso de falhas simples de "hardware", sem ter de modificar seu "software" básico.

Nestas condições é suficiente que sejam incorporadas ao Sistema Operacional iRMX86 apenas duas novas funções. Uma delas, denominada RECURSO PARA TOLERÂNCIA A FALHAS, destina-se a apoiar a recupe

ração retroativa da aplicação de supervisão. A outra, GERENTE DE RECUPERAÇÃO, nada mais é do que uma tarefa de interrupção responsávle pela recuperação do Sistema de Supervisão, depois de detetada uma falha de "hardware".

A *extensão do Sistema Operacional iRMX86* com estas funções é bastante simples, pois nem mesmo exige a definição de novos objetos. A incorporação de uma chamada ao sistema denominada "SAVE" é suficiente.

Como um RECURSO PARA TOLERÂNCIA A FALHAS, a chamada "SAVE" permite o estabelecimento de pontos de recuperação definidos pelo projetista do Sistema de Supervisão.

Assim, a única alteração necessária à aplicação de supervisão para torná-la tolerante a falhas é a invocação da primitiva "SAVE" em locais específicos das INSTÂNCIAS considerados pelo projetista como adequados para o estabelecimento de pontos de recuperação.

No que diz respeito ao estabelecimento adequado destes pontos de recuperação, propõe-se nesta dissertação que o projetista do Sistema de Supervisão explore a característica *cíclica* destas aplicações.

As INTERFACES de AQUISIÇÃO e de ATUAÇÃO, por permitirem uma interação como o meio externo, atribuem uma natureza iterativa a Sistemas de Supervisão de tempo real. Este atributo repetitivo pode estar implícito se os sistemas admitem interrupção (como muitos sistemas de controle de processos) porém, por outro lado é usualmente exibido na forma de operações cíclicas.

Um *grafo de sincronização* é uma ferramenta que pode ser utilizada para representar a estrutura dos processos de sistemas cíclicos de tempo real. Ele mostra quais os processos a serem executados e quando cada um deles deve iniciar e terminar.

Os sistemas de Supervisão operam em tempo real. Nestes sistemas, entradas podem ser esperadas e/ou saídas devem ser geradas de acordo com algum escalonamento de tempo real. Portanto tais sistemas podem ser modelados por grafos de sincronização para identificar seus *ciclos de controle*. Então, a aplicação da REGRA I, definida no Capítulo 5, pode ser efetivada pelo projetista, apesar de esta solução contar com o estabelecimento de pontos de recuperação ao nível de *interface interpretativa* e não mais com a invocação da primitiva "SAVE" ao nível da aplicação.

Desta forma estará garantida ao projetista do Sistema de Supervisão a otimização dos pontos de recuperação, o que significa menor execução da operação "SAVE" durante a operação do sistema e conseqüentemente maior tempo disponível para o processamento das atividades de supervisão.

Uma preocupação bastante considerável em Sistemas de Supervisão é quanto à *repetitividade* (preservação) dos dados a serem adquiridos da INTERFACE de AQUISIÇÃO e quanto à repetição de saída pela INTERFACE de ATUAÇÃO, no caso de falha do sistema e possível reexecução de tais atividades. Neste sentido, o conceito de *comunicação confiável* deve ser aplicado através das REGRAS II e III. Contudo o projetista deve cuidar para não abusar na quantidade dos pontos de recuperação a serem estabelecidos.

É importante salientar que o sistema de Supervisão Tolerante a Falha proposto não está livre das limitações intrínsecas à técnica de recuperação retroativa adotada.

## REFERÊNCIAS BIBLIOGRÁFICAS


ANDERSON, T.; KNIGHT, J.C. A framework for software fault tolerance in real-time systems. *IEEE Transactions on Software Engineering*, SE-9(3):355-364, May, 1983.

ANDERSON, T.; LEE, P.A. *Fault tolerance principles and practice.* Londres Prentice/Hall, 1981.

AVIZIENIS, A. Fault tolerant systems. *IEEE Transactions on Computer*, C-25(12):1304-1312, 1976.

BARTLETT, J.F. *A "Nonstop operating system"*, Cupertino, CA., Tandem Computers, c1977. (Paper].

HEIDER, G. Let operating systems aid in component designs. *Computer Design*, 21(9):151-159, Sept., 1982.

INTEL. *Introduction to the iRMX86 operating system.* Santa Clara, CA., Intel, 1982a. Manual de referência do sistema.

———. *iRMX86 nucleus reference manual.* Santa Clara, CA. Intel, 1982b. Manual de referência do sistema.

———. *iRMX86 operating system.* Santa Clara, CA., Intel, 1982c. Manual de referência do sistema.

KOPETZ, H. *An architecture for a maintainable real time system (MARS).* Berlin, Technische Universitat, Abril, 1982. (Report MA 82/2).

LOQUES FILHO, O.G. *A fault tolerant distributed computer control system.* Doctor of Philosophy. London, University of London. Department of Computing, Feb., 1984a.

———. Uma técnica para a construção de software distribuído tolerante a falhas. In: SIMPÓSIO SOFTWARE BÁSICO, 4., ITA, São José dos Campos, 29-31 out. 1984. *Anais*, São José dos Campos, SBC, 1984b. p.166-172.

———. Uma arquitetura flexível para sistemas distribuídos tolerantes a falhas. In: CONGRESSO DA SOCIEDADE BRASILEIRA DE COMPUTAÇÃO, 5.; CONFERÊNCIA LATINO-AMERICANA DE INFORMÁTICA, 11., Porto Alegre, 20-27 jul. 1985. *Anais*. Porto Alegre, SBC, 1985. p.436-446.

MARTINS, R.C.O.; DE PAULA A.R. *A fault-tolerant 16 bits multiprocessing unit for on-board* satellite applications. Sta. Anne-de-Bellevue, Canadá, Spar Aerospace, Apr., 1984. (SPAR TR ≠ RML - 009 - 84 - 57).

RANDELL, B. System structure for software fault tolerance. *IEEE Transactions Software Engineering*, SE-1(3):220-232, June, 1975.

RANDELL, B.; LEE, P.A.; TRELEAVEN, P.C. Reliability issues in computing system design. *Computing Surveys*, 10(2):123.165, June, 1978.

ROSSI, G.P.; SIMONE, C. A multitasking operating system with explicit treatment of recovery points. *Microprocessing and Microprogramming*, 14(2):55-66, Sept., 1984.

RUSSEL, D.L. State restoration in systems of communicating processes. *IEEE Transactions on Software Engineering*, SE-6(2):183-194, Mar., 1980.

SCHMITTER, E.J.; BAVES, P. The basic fault-tolerant system. *IEEE Micro*, 4(1):66-74, Feb., 1984.

SERLIN, O. Fault-tolerant systems. in commercial applications, *IEEE Computer*, 17(8):19-30, Aug., 1984.

SIEWIOREK, D.P.; SWARZ, R.S. *The theory and practice of reliable system design*. Bedford, M.A., Digital Equipment, 1982.

TOY, W.N. Fault-tolerant design of local ESS Processors. *Proceedings of IEEE*, 66(10):1126-1145, Oct., 1978.

# APÊNDICE A

## LISTA DE CHAMADAS AO NÚCLEO DO iRMX86

Este apêndice lista as chamadas ao sistema que o NÚCLEO do Sistema Operacional iRMX86 reconhece. É dada também uma breve descrição de cada chamada. Maiores detalhes destas primitivas poderão ser encontradas nos Manuais de Referência INTEL (1982a, 1982b, 1982c).

*ACCEPT$CONTROL* - obtém o controle de uma região somente se a região estiver disponível imediatamente.

*CATALOG$OBJECT* - cataloga um nome e um sinal para um objeto no diretório de objetos de um "job".

*CREATE$JOB* - cria um ambiente para um número de tarefas e outros objetos, assim como cria uma tarefa inicial e sua pilha.

*CREATE$MAILBOX* - cria uma caixa postal com fila de tarefas e fila de objetos, organizadas por prioridade ou FIFO.

*CREATE$SEGMENT* - aloca dinamicamente um número de parágrafos de 16-bytes específico.

*CREATE$SEMAPHORE* - cria um semáforo para sincronização de acesso a recursos.

*CREATE$TASK* - cria uma tarefa com uma prioridade específica e área de pilha.

*DELETE$JOB* - descarta um "job" e todos os objetos pertencentes a ele somente se este "job" não contém nenhum outro "job" ("job" filho). Toda a memória usada é devolvida para o "job" que o contém ("job" pai).

*DELETE$MAILBOX* - descarta uma caixa postal.

*DELETE$SEGMENT* - descarta um segmento específico desalocando a memória.

*DELETE$SEMAPHORE* - descarta um semáforo.

*DELETE$TASK* - descarta uma tarefa do sistema e remove-a das filas em que ela pode estar esperando por objetos.

*DISABLE* - desabilita o "hardware" de aceitar interrupções abaixo de um nível específico, inclusive este nível.

*ENABLE* - habilita o "hardware" de aceitar interrupções a partir de um nível específico.

*EXIT$INTERRUPT* - usada por um manipulador de interrupção para liberar o controle do sistema.

*GET$LEVEL* - retorna o número do nível de interrupção de mais alta prioridade que está sendo processado.

*GET$POOL$ATTRIBUTES* - retorna atributos tais como: tamanho de memória da tarefa (mínimo, máximo e corrente) no ambiente do "job" ao qual a tarefa pertence.

*GET$PRIORITY* - obtém a prioridade atual de uma tarefa específica.

*GET$SIZE* - retorna o tamanho (em bytes) de um segmento.

*GET$TASK$TOKENS* - obtém o *sinal* (token) da tarefa chamante ou os *sinais* dos objetos associados à tarefa.

*GET$TYPE* - retorna um código para o tipo de objeto referenciado pelo sinal do objeto.

*LOOKUP$OBJECT* - retorna um sinal para o objeto cujo nome especificado foi encontrado no diretório de objetos do "job" especificado.

*OFFSPRING* - provê uma lista de todos "jobs" correntes criados pelo "job" especificado.

*RECEIVE$MESSAGE* - cuida de receber um objeto de uma caixa postal específica. Se o objeto não estiver disponível, a tarefa chamante pode escolher esperar por ele um número de unidades de tempo do sistema.

*RECEIVE$UNITS* - cuida de obter um número específico de unidades de um semáforo. Se as unidades não estão disponíveis imediatamente, então a tarefa chamante pode escolher esperar.

*RESET$INTERRUPT* - desabilita um nível de interrupção e cancela o manipulador de interrupção daquele nível. Se uma tarefa de interrupção estiver associada, ela é descartada.

*RESUME$TASK* - continua a execução de uma tarefa. Se a tarefa foi suspensa vária vezes, a profundidade de suspensão é reduzida de um, e ela permanece suspensa.

*SEND$CONTROL* - libera o controle de uma região.

*SEND$MESSAGE* - envia um objeto para uma caixa postal especificada. Se uma tarefa estiver esperando por este objeto, ele é passado para a tarefa apropriada de acordo com a disciplina da fila de tarefas. Se nenhuma tarefa estiver aguardando por ele, o objeto é enfileirado na caixa postal.

*SEND$UNITS* - incrementa o contador do semáforo de um número específico de unidades.

*SET$INTERRUPT* - associa um manipulador de interrupção e, se desejado, uma tarefa de interrupção a um nível de interrupção específico. Geralmente a tarefa chamante torna-se a tarefa de interrupção.

*SET$POOL$MIN* - muda dinamicamente os requisitos mínimos de memória do "job" que contém a tarefa chamante.

*SET$PRIORITY* - altera dinamicamente a prioridade da tarefa especificada.

*SIGNAL$INTERRUPT* - usada por um manipulador de interrupção para sinalizar a tarefa de interrupção associada da ocorrência de uma interrupção.

*SLEEP* - faz com que uma tarefa entre no estado DORMENTE durante um número de unidades de tempo do sistema.

*SUSPEND$TASK* - suspende a operação de uma tarefa. Se a tarefa já foi suspensa, sua profundidade de suspensão é aumentada de um.

*UNCATALOG$OBJECT* - remove um objeto e seu nome do diretório de objetos de um "job".

*WAIT$INTERRUPT* - usado por uma tarefa de interrupção para manter-se no estado DORMENTE até que o manipulador de interrupção associado sinalize a ocorrência de uma interrupção.

As chamadas ao sistema apresentadas a seguir são consideradas chamadas do programador, pois causam um efeito global no sistema:

*A$GET$EXTENSION$DATA* - retorna para o Sistema de e/s os dados de extensão armazenados num arquivo.

*A$PHYSICAL$ATTACH$DEVICE* - conecta um dispositivo ao Sistema Básico de e/s.

*A$PHYSICAL$DETACH$DEVICE* - desconecta um dispositivo do Sistema Básico de e/s.

*A$SET$EXTENSION$DATA* - coloca os dados de extensão num arquivo do Sistema de e/s.

*ACCEPT$CONTROL* - requisita acesso aos dados protegidos por uma região, somente se o acesso estiver disponível imediatamente.

*ALTER$COMPOSITE* - altera a lista de componentes de um objeto composto.

*CREATE$COMPOSITE* - cria um objeto composto.

*CREATE$EXTENSION* - cria uma nova extensão de tipo de objeto.

*CREATE$REGION* - cria uma região.

*CREATE$USER* - cria um objeto de usuário.

*DELETE$COMPOSITE* - descarta um objeto composto.

*DELETE$REGION* - descarta uma região.

*DELETE$USER* - descarta um objeto de usuário especificado.

*DELETE$DELETION* - incrementa de um a profundidade de descarte desabilitado de um objeto.

*ENABLE$DELETION* - decrementa de um a profundidade de descarte desabilitado de um objeto.

*FORCE$DELETE* - força o descarte de um objeto mesmo se ele já teve seu descarte desabilitado uma vez.

*INSPECT$COMPOSITE* - retorna uma lista dos objetos componentes de um objeto composto.

*INSPECT$USER* - retorna uma lista dos identificadores de um objeto de usuário.

*LOGICAL$ATTACH$DEVICE* - conecta um dispositivo ao Sistema Estendido de e/s.

*LOGICAL$DETACH$DEVICE* - desconecta um dispositivo de um Sistema Estendido de e/s.

*RECEIVE$CONTROL* - requisita acesso eventual aos dados protegidos por uma região.

*SEND$CONTROL* - libera acesso aos dados protegidos por uma região.

*SET$OS$EXTENSION* - aloca e desaloca entradas de extensão na tabela do vetor de interrupção.

*SET$PRIORITY* - muda dinamicamente a prioridade de uma tarefa.

*SET$TIME* - coloca o tempo e os dados.

*SIGNAL$EXCEPTION* - sinaliza a ocorrência de uma condição excepcional.

# APÊNDICE B

# CONFIGURAÇÃO DE MEMÓRIA

Neste trabalho a memória interna (principal) de cada UNIDADE do sistema e os MÓDULOS da Memória Estável devem ser configurados como mostra a Figura B1.

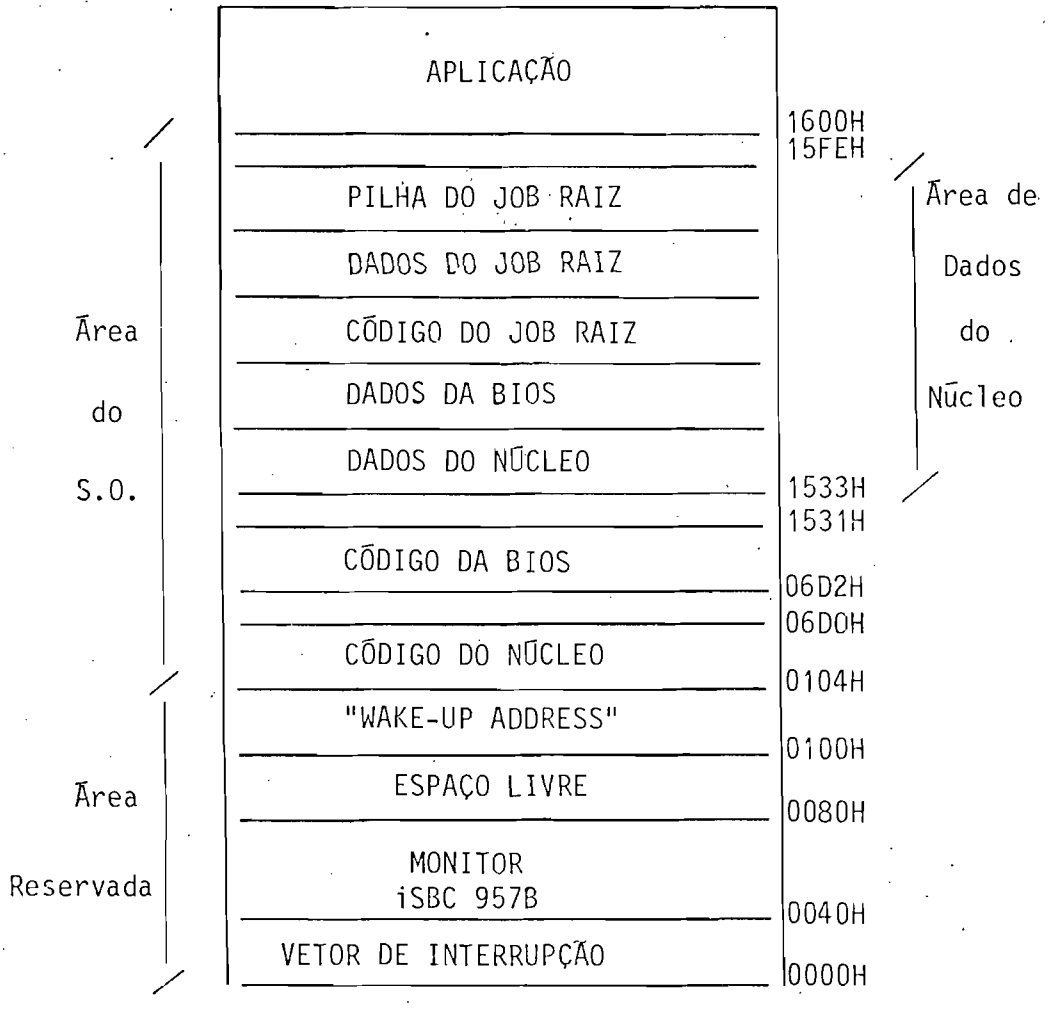


Fig. B1 - Configuração de Memória.

# APÊNDICE C

## TABELA DE TRANSFERÊNCIA

## TABELA DE TRANSFERÊNCIA

A TABELA DE TRANSFERÊNCIA contém os endereços absolutos a serem transferidos para a Memória Estável. Esta tabela é utilizada na ALTERNATIVA 1 proposta para a Memória Estável. Ela deve ser gerada na inicialização do Sistema de Supervisão, no momento que o usuário cria seus "jobs" e tarefas. Como nesta ALTERNATIVA, a alocação da memória é estática, então a TABELA DE TRANSFERÊNCIA é chamada assim que terminada a tarefa de inicialização do sistema.

O procedimento para geração dos dados contidos na TABELA DE TRANSFERÊNCIA é bastante simples e descrito a seguir.

Utilizando o Interactive Configuration Utility (ICU), que é um programa utilitário oferecido pelo Sistema Operacional iRMX86 para facilitar a configuração do sistema, o usuário pode definir sua aplicação e atribuir valores para:

- base de segmentos de dados de cada tarefa e/ou "job";
- endereço base da pilha de cada tarefa;
- tamanho da pilha de cada tarefa.

## ALOCAÇÃO DOS SEGMENTOS DE DADOS

O valor que o usuário especifica depende do *modelo de segmentação* utilizado. Sugere-se que seja o PL/M-86 MEDIUM AND COMPACT MODEL PROCEDURES cujos procedimentos possuem *segmentos de dados* alocados estaticamente. Para este modelo de segmentação o compilador não inicializa automaticamente o registro de segmentos de dados. Neste caso, a especificação da *base do segmento de dados de uma tarefa/"job"* deve ser feita na especificação do "job" de primeiro nível que contém esta tarefa/"job". Isto é possível através da chamada ao sistema CREATE$TASK/CREATE$JOB.

ALOCAÇÃO DE PILHA

Utilizando o PL/M-86 MEDIUM MODELS o usuário pode especificar precisamente o *endereço base da pilha* de cada tarefa e também o *tamanho da pilha* de cada tarefa a partir do "job" de primeiro nível que contém a tarefa. Isto também é feito a partir de invocações à primitiva CREATE$TASK oferecida pelo NÚCLEO do iRMX86.

Cabe aqui algumas considerações quanto à INICIALIZAÇÃO DO SISTEMA DE SUPERVISÃO:

1 - O Sistema Operacional iRMX86 oferece um "job" (raiz) com uma *tarefa para inicialização.*

2 - O usuário, na geração de seu sistema, deve determinar a *área alocada para este "job" (raiz).*

3 - Na inicialização, utilizando a *tarefa para inicialização,* o usuário deve criar todos os "jobs" do seu sistema com suas respectivas tarefas, a partir do "job" (raiz). Isto é feito através da invocação das primitivas CREATE$JOB e CREATE$TASK.

Em ambas primitivas citadas, quatro parâmetros de entrada, definidos pelo usuário, são considerados relevantes para a geração da TABELA DE TRANSFERÊNCIA. São eles:

1 - start$address - endereço de início do segmento que contém a tarefa/"job" (área de código mais área de dados);

2 - data$segment - endereço do início do *segmento de dados* utilizado pela tarefa/"job";

3 - stack$ptr - endereço do início da pilha da tarefa/"job";

4 - stack$size - tamanho, em bytes, da pilha.

Dispondo destes parâmetros o sistema tem condições de saber onde estão alocadas as áreas de dados dos JOBs e das TAREFAS na memória. E, portanto tem-se condições de gerar a TABELA DE TRANSFERÊNCIA.

Para tanto propõe-se a extensão do NÚCLEO do iRMX86 com duas novas primitivas: *CRIATAREFA* e *CRIAJOB*. Estas primitivas do ponto de vista do usuário nada mais são do que substitutas das chamadas ao sistema CREATE$TASK e CREATE$JOB providas pelo NÚCLEO do iRMX86. Porém, elas são imprescindíveis na geração da TABELA DE TRANSFERÊNCIA, pois só assim o nível RECURSO PARA TOLERÂNCIA A FALHAS tem condições de conhecer os parâmetros acima citados e montar a TABELA DE TRANSFERÊNCIA descrita a seguir.

Na realiadade, as informações que uma TABELA DE TRANSFERÊNCIA deve conter, para cada tarefa do Sistema de Supervisão, são:

- identificador ao JOB ao qual a TAREFA pertence;
- identificador da TAREFA;
- início da área de dados da TAREFA;
- final da área de dados da TAREFA.

Cabe lembrar que criar um JOB significa criar a *tarefa inicial* do JOB.

Para cada JOB, o usuário deve organizar a alocação de memória como mostra a Figura C1.

Assim, no preenchimento da TABELA DE TRANSFERÊNCIA serão utilizados os parâmetros:

INCDAD = início da área de dados:= data$segment e

INCPIL = final da área de dados:= stack$ptr.

Para que se possa estabelecer o final da área de dados da TAREFA impõe-se que o parâmetro stack$ptr, que é do tipo POINTER (32 bits), não poderá ter sua parte BASE (primeiros 16 bits) igual a zero, isto é, não é permitida deixar ao NÚCLEO a alocação da pilha da nova TAREFA.

Uma proposta de implementação das primitivas CRIAJOB e CRIATAREFA é dada a seguir, em uma linguagem estruturada do tipo ALGOL.

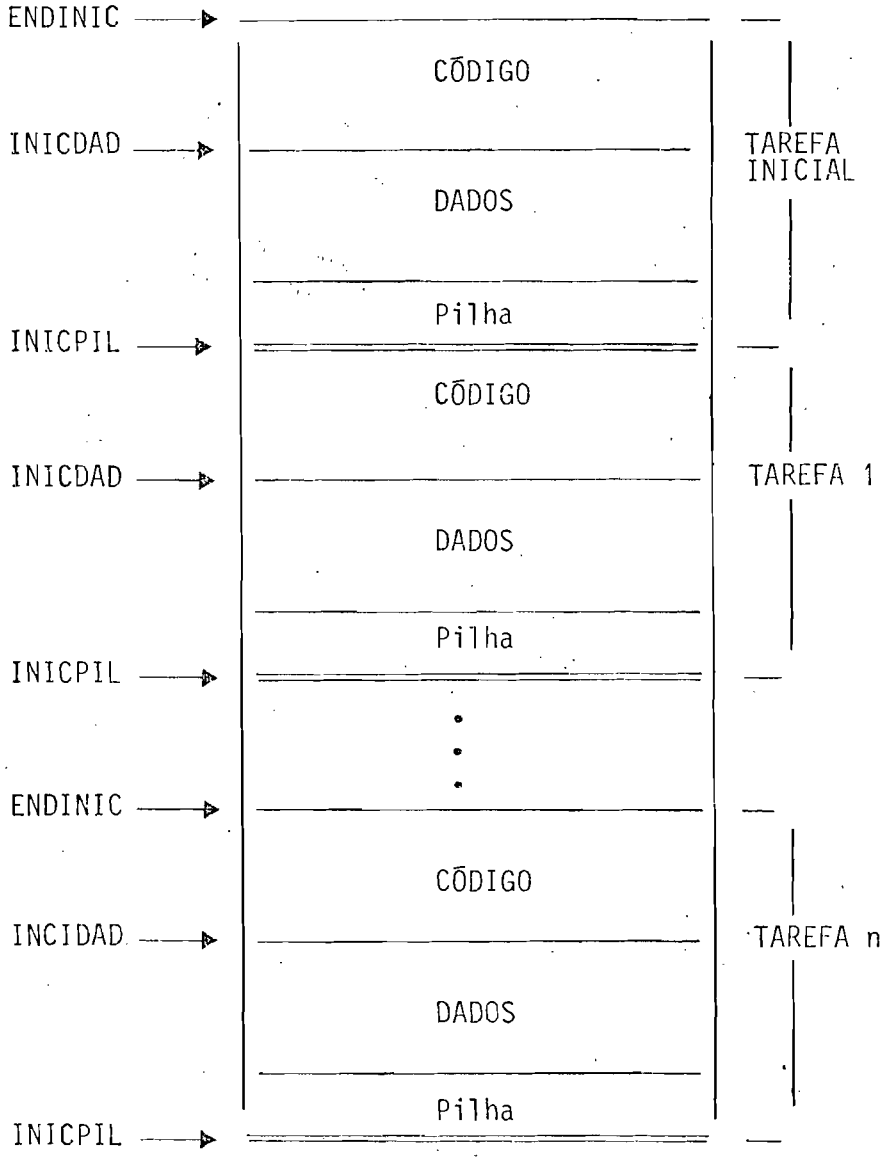


Fig. C1 - Organização do poço máximo por JOB.

Primitiva CRIAJOB

```
JOB = CRIAJOB  (directory$size,  param$obj,  pool$min,  pool$max,
               max$object,       max$tasks,      max$priority,
               except$handler,   job$flags,      task$priority,
               start$address,  data$seg,  stack$ptr, stack$size,
               task$flags, except$ptr);

DEFINE TABELA: ARRAY of IDJOB: token, IDTAR: token, INÍCIO: word,
                        FIM: word;

BEGIN
     INICDAD:= data$seg;

     INICPIL:= stack$ptr;

     JOB:= CREATE$JOB (directory$size,      param$obj, pool$min,
          pool$max,    max$object,     max$tasks, max$priority,
          except$handler,       job$flags,       task$priority,
          start$address,    data$seg,  stack$ptr,   stack$size,
          tasks$flags, except$ptr);

     IF except$ptr=E$OK THEN ; nenhuma condição excepcional
     BEGIN                   ;preencha a TABELA DE TRANFERÊNCIA
          IDJOB:= JOB;

          IDTAR:= TAREFA INICIAL;

          INICIO:= INCDAD;

          FIM:= INCPIL;

     END;

END.
```

Primitiva CRIATAREFA

```
TAREFA = CRIATAREFA (priority, star$address, data$seg,
                     stack$ptr, stack$size, task$flags,
                     except$ptr);

DEFINE TABELA: ARRAY of IDJOB: token, IDTAR: token, INICIO: word,
                        FIM: word;

BEGIN
     INICDAD:= data$seg;

     INICPIL:= stack$ptr;

     TAREFA:= CREATE$TASK (priority, start$address, data$seg,
              stack$ptr, stack$size, tasks$flags, except$ptr);

     IF except$ptr=E$OK THEN ; nenhuma condição excepcional
     BEGIN                     ;preencha a TABELA DE TRANSFERÊNCIA

          JOB:= GET$TASK$TOKENS (TAREFA, except$ptr);

          IDJOB:= JOB;

          IDTAR:= TAREFA;

          INICIO:= INCDAD;

          FIM:= INCPIL;

     END;

END.
```